THESIS

Evolution of artificial neural networks using a two-dimensional representation

by

João Carlos Figueira Pujol

Thesis submitted to the School of Computer Science
of the University of Birmingham for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
The University of Birmingham
April 1999

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Riccardo Poli, for his permanent guidance throughout my research, and also for revising the manuscript of the thesis.

I also thank Dr. Peter Hancox and Dr. Manfred Kerber, members of my thesis group, for useful discussions during the several meetings we had in the course of three years.

I wish to thank my colleagues Jonathan Page, Ahmed Badawy and Amr Radi for helping me identify some points in the thesis which were not sufficiently clear.

I would like to thank the support team of the computer system, for their constant technical advice.

Abstract

The design of artificial neural networks is still largely performed by an expert, with only a few heuristics to guide a trial-and-error search. Recently, new methods based on evolutionary computation (EC) have been applied to the synthesis of artificial neural networks with modest results. The basic limitation of EC-based methods is that they do not take into account the fact that artificial neural networks are two-dimensional structures, and do not use specialized evolutionary operators. In this work, a new method based on a special form of evolutionary computation called genetic algorithms is proposed for the evolution of artificial neural networks. The method is a general purpose procedure able to evolve feed-forward and recurrent architectures. It is based on a two-dimensional representation, and includes operators to evolve the architecture and the connection weights simultaneously. The new approach has shown promising results, and has fared better than previous methods in a number of applications, including: binary classification problems, design of neural controllers and a complex navigation task of traversing a trail. An extension of the two-dimensional representation is also presented, which can be combined with other methods, providing them with an alternative procedure to evolve the weights of the connections.

Chapter 1

Introduction

Artificial neural networks (ANNs) are a class of computational tools inspired by the biological nervous system [77]. They consist of partially or fully interconnected simple processing units, called *neurons*. ANNs derive their power from a parallel distributed structure, and from their ability to learn underlying relations from a given set of representative examples, instead of following a predefined set of rules. The connections between units have weights, which must be adjusted through a training process, to solve a particular problem.

Artificial neural networks have been successfully applied to a number of problems in many areas including science and engineering [49]. However, the training process as well as the performance of the artificial neural network are strongly influenced by its architecture. Unfortunately, the space of ANNs to solve a specific problem is infinite and complex, and there is no general purpose, reliable, automatic method to search this space. This task is still largely performed by an expert, in a laborious process of trial-and-error definition of the architecture, combined with a training process to adjust the weights.

Evolutionary computation is a class of global optimization techniques also inspired by biology [50]. Evolutionary algorithms are computational tools based on the collective learning process of a population of individuals, where each individual is a trial solution to a given problem. This population is randomly initialized, and then evolved, by means of the

repeated application of evolutionary operators. These operators are biased to favor individuals with better performance in the task at hand. As a result, a new population with better performance is created at each generation, to explore the space of trial solutions. Ideally, this process will eventually converge to a solution.

The ability of evolutionary algorithms to search large and complex spaces has been demonstrated in a wide range of tasks [11]. As a consequence, they seem to be natural candidates to methods for designing artificial neural networks.

Genetic algorithms (GAs) are a form of evolutionary computation which strongly emphasizes recombination as the main drive of evolution [80, 63]. GAs have been combined with ANNs in three major forms: for adjusting the weights, for optimizing the architecture, and for performing both simultaneously.

In the first case, the role of GAs is to adjust the weights of a given architecture, replacing a conventional training procedure [77]. Many times conventional training algorithms can not be applied or, due to the peculiarities of the search space or of the architecture, training by GAs may be more efficient. The problem of using GAs to optimize the weights is that the topology must be known in advance or must be constructed using other methods.

In the second case, genetic algorithms are used to evolve a suitable architecture to the problem at hand. This is carried out by using a conventional procedure for training the architectures at each generation. This combination has been fairly successful. Unfortunately, the computational cost involved in training an entire population of architectures at each generation is prohibitive.

The most ambitious combination attempts to evolve the architecture and the weights simultaneously without a separate training process. However, until now, this combination has yielded modest results. The main reason for this is that the efficiency of genetic algorithms is considerably affected by the mechanism used to encode the individuals of the population. To be efficient, it is of paramount importance to use a suitable representation for the architecture and the weights. Most previous methods based on genetic algorithms do not

take into account the fact that artificial neural networks are two-dimensional structures, and do not use evolutionary operators specifically designed to evolve ANNs. They either convert the artificial neural network into a form suitable for the application of standard GAs operators, or transfer the search into a space of rules for constructing ANNs.

In this work, a new method is presented, which is based on a two-dimensional representation very similar to the artificial neural network itself. The method has considerable advantages over previous ones, as it allows the definition of evolutionary operators specialized to evolve the architecture and the weights concurrently. The thesis is organized as follows.

In Chapters 2 and 3, an introduction to artificial neural networks and evolutionary computation is given in order to introduce the problem to be addressed, and to provide the basic terminology for the method proposed.

In Chapter 4, a discussion of representative achievements of genetic algorithms applied to the evolution of artificial neural networks is carried out in the form of a literature review. The drawbacks of previous methods as well as the lessons which can be learned from them are discussed.

In Chapter 5, on the grounds of the lessons learned in Chapter 4, a new method is proposed to the synthesis of artificial neural networks based on genetic algorithms. The new representation and a new specialized crossover operator are introduced.

In Chapter 6, experimental results on the application of the method to standard benchmark problems are reported, and its efficiency is compared to other approaches.

In Chapter 7, an extension of the method is proposed, which uses genetic programming (a special form of genetic algorithms) to evolve the weights. Results are reported on the application of the approach to the same problems discussed in Chapter 6.

Finally, in Chapter 8, a summary of what has been achieved in the thesis is presented, and outlines for future research are discussed, including the combination of the approach proposed in Chapter 7 with other methods to evolve artificial neural networks.

Chapter 2

Artificial neural networks

2.1 Introduction

Artificial neural networks, or simply networks for brevity, are computational tools inspired by the biological nervous system. They are systems of partially or fully interconnected simple processing units, called neurons. [78, 77, 177, 103].

The neuron is usually a nonlinear unit that receives input signals from other units or from the environment (the space of input data relevant to the problem at hand), yielding an output. The signals received by a neuron are modulated by real numbers called *connection weights* (or simply weights). The total input of a neuron is obtained by adding an activation threshold (*bias*) to the weighted sum of the signals received. The output signal of a neuron is computed by the application of an *activation function* to its total input. Typical activation functions include: a step function, a sigmoid, a Gaussian function, or even more than one type of activation function can be used in the network [77, 168].

Artificial neural networks derive their power from a parallel distributed structure, capable of generalization and learning by examples, instead of following a predefined set of rules. They can learn underlying rules (like input/output relations) from a given collection of representative examples. ANNs may also be visualized as oriented graphs, where the nodes

of the graph are occupied by neurons, and the links between nodes represent connections between neurons.

According to the connectivity pattern and the direction of signal propagation, artificial neural networks can be grouped into two classes [85]:

- *feedforward*, in which the neurons are evaluated in order, and there are no connections from higher order neurons to lower order ones;
- *recurrent*, in which feedback connections may be present, allowing the formation of loops.

In feedforward networks the neurons are usually arranged in layers: input and output layers, which interact with the environment, and one or more layers of hidden neurons, which do not have contact with the environment. There is a definite order of evaluation of the neurons. The network receives the input signals, propagates them through all the layers, and returns signals to the environment through the output neurons. This means that, for the same input, the network always returns the same output, independent of any previous state, i.e. the network has no memory.

In recurrent networks there may be separate input and output neurons, or all of them may be considered input as well as output ones. Differently from their feedforward counterparts, recurrent networks have memory, as previous states of the neurons may be used to update their current state, through feedback connections. They are dynamical systems able to produce a sequence of different outputs from the same input. The order of neuron evaluation may be: synchronous, where the output of all neurons are computed in parallel, asynchronous, where only part of the neurons are randomly selected for updating, or from input to output [77].

Artificial neural networks have been successfully applied to a number of problems in a broad range of different fields [49, 111, 29, 135]. Despite their success, the design of artificial neural networks is still largely performed through a tedious process of trial-and-

error definition of the *architecture* (number of neurons and connectivity), combined with a training process to adjust the weights.

2.2 Training

No matter what architecture is selected to solve a particular problem, the weights must be determined before the network can be used. Although in some cases the weights can be calculated analytically (e.g. Hopfield networks [77]), they are usually randomly initialized with raw values within a specified range, and subsequently adjusted by a *training* (a.k.a. *learning*) algorithm.

In a large class of problems, a set of input patterns is presented to the network and, for each pattern, the output of the network is computed and compared to an expected response, yielding an *error signal*. The error signal defines an *error surface* as a function of the weights, and can be used to adjust them. This optimization process is carried out iteratively, aiming at eventually making the network reproduce the desired output patterns within an acceptable error [77, 78]. There is no unique learning paradigm for artificial neural networks. Depending on the application and the architecture of the network, different training techniques can be used, with their own limitations and advantages [77, 79, 49, 197].

For example, a widely used training procedure for multilayer feedforward networks is *back-propagation*. This learning paradigm propagates the error signal associated to the output of the network, from the output layer to the previous layers. In this process, the weights are adjusted by a steepest-descent strategy based on local information about the gradient of the error signal as a function of the weights. By interpreting the weights as components of a vector, the gradient defines a *directional vector* along which the weight vector is updated [181, 77, 139, 79, 160, 16, 197]. Although simple to implement, this learning paradigm has considerable shortcomings:

- If the error surface is relatively flat in some regions or directions, a small value for the gradient will lead to a slow convergence rate [77, 168].

- The error surface associated with the problem is often not convex and, depending on weight initialization, backpropagation can be trapped in local minima. To reduce this problem backpropagation can be combined with statistical methods (e.g. *simulated annealing* [77]). However, this sort of approach is computationally costly, and the outcome is still not guaranteed [37].

- The desired output of the network must be known for each input pattern, and the activation function as well as the output error must be differentiable [77, 168]. In many problems a target response for each input pattern is not available, but only a measure of the overall performance of the network after many action steps (e.g. in *reinforcement learning* [77, 29]). Sometimes, threshold functions are the best option to solve a problem, but the requirement of differentiability of the activation function prevents them from being used.

- The efficiency of the algorithm depends on learning parameters, whose values may lead to slow convergence or oscillatory behavior [77]. The *Delta-Bar-Delta* (DBD) learning rule [77] and the *resilient propagation* (RPROP) learning algorithm [157] attempt to bypass this problem in the course of the training process, by using the sign of the partial derivative of the error signal with respect to the weights in the previous and current iterations, to adapt the learning parameters (DBD) and to determine the sign of the update of the weights (RPROP). *Conjugate-gradient* [77] is another attempt to address the problem, which uses an intricate combination of the gradient of the error signal in the previous and current iterations, to define the directional vector for updating the weights. Conjugate-gradient also performs a search at each iteration, for the optimal value of the learning parameters to minimize the error signal. Although more efficient, these methods are computationally more expensive than the standard backpropagation algorithm.

- Depending on the peculiarities of the error surface, the gradient may simply point away from the minimum on the error surface, leading the algorithm to move in the wrong direction [77].

The application of training paradigms based on gradient information to recurrent architectures is even more difficult, as the recurrent versions are considerably more complex than the feedforward counterparts [77, 16, 78, 104, 83]. For example, the operation of any recurrent network can be simulated by a feedforward network for a finite period of time [119, 160, 29]. This idea is explored in the *backpropagation-through-time* algorithm [160], which is simply standard backpropagation applied to the substitute feedforward network. The problem of this approach is that the computation and memory requirements increase with the size of the training sequences, due to the manifold duplication of units in the equivalent feedforward network, to simulate the dynamics of the recurrent network [78]. An alternative is provided by the *real-time recurrent learning* algorithm [199]. In this case, at each evaluation of the recurrent network, an error signal is generated, and used to compute the gradient for continually updating the weights. There is no need for duplication of units, and training sequences of arbitrary length are allowed. However, for a fully connected network with *N* neurons, the number of computations for each update of the weights scales as N^4 [78, 1] (to contrast with N^2 for standard backpropagation applied to a fully connected feedforward network with *N* neurons). Recurrent networks which are expected to stabilize before their output is read, can be trained by *recurrent backpropagation* [138]. The algorithm uses two networks with the same architecture: the original network to be trained, and another one to compute gradient information to adjust the weights. In this case, instead of one, two networks have to relax to a stable state at each iteration of the learning process.

No matter what procedure is used for training, its application still requires the previous specification of the architecture, often referred to as topology, configuration or structure of the network. To address this issue, some solutions have been proposed.

2.3 Architecture definition

In principle, it is always possible to select an architecture complex enough, and train it to solve a particular problem. However, this may unnecessarily slow down the training process and, foremost, a large topology may lead to overfitting of the training set. The training set is presumably a representative sample of the universe the network is supposed to face. After training, the network should be able to return reasonable outputs for inputs which were not included in the training set. This ability is called generalization, and an overfitted network is unable to do that, as it was adjusted to fine details in the input data, instead of having learned general features. Also, a smaller network makes more amenable the analysis of the internal representations encoded in the hidden neurons [168] (nobody likes ”black boxes”). Moreover, the computation of the output of a large network simulated via software as well as its hardware implementation are more expensive. Therefore, it is important to find a solution with a small topology [176].

Some heuristics to estimate the number of hidden neurons (the number of input or output neurons is constrained by the problem at hand) can help constrain the topology of the network [165, 82, 19]. These methods are based on the fact that multilayer feedforward networks with a single hidden layer using a squashing activation function (e.g. a hyperbolic tangent) are universal approximators, in that any continuous function can be approximated to any degree of accuracy if a sufficient number of hidden neurons is provided [77, 81, 20, 67, 128, 40]. However, these constraints overestimate the size of the network, and do not provide a complete description of the architecture, making the arbitrary choice of the connectivity still necessary. Moreover, they are only applicable to feedforward networks.

To bypass this limitation, *destructive* and *constructive* methods have been proposed. Destructive and constructive algorithms attempt to automatically build artificial neural networks by combining training with adaptation of the topology. The idea is to remove or add features from or to an unsatisfactory architecture according to deterministic rules, and then retrain it, partially or totally.

2.3.1 Destructive methods

Destructive algorithms start with a large network, which is expected to be sufficiently rich to solve the problem, and gradually remove unnecessary elements during or after the training process [155, 38, 60, 168].

To perform pruning during training, a *penalty term* (a term proportional to the sum of the square of the weights, for example) can be added to the error function. When used with gradient-descent-based learning methods, this tend to reduce the value of the weights and, as a consequence, unimportant weights are driven to zero, allowing the corresponding connections to be removed [155, 174, 190, 38]. Eventually, neurons with no output connections as well as neurons with no input connections and constant output can be removed, by adding their output signals to the bias of other neurons.

The other alternative is to prune the network after a solution to the problem has been found, by eliminating unimportant connections (eventually neurons as in the previous case), and then retrain the network. This process is repeated until an architecture sufficiently small is achieved, or the network fails to converge to a solution. For example, Cun at al. [39] use the second order partial derivative of the error signal with respect to the weights to estimate the relative importance of a weight. Those weights whose estimated relative importance are below a specified threshold are deleted, and the network is retrained. Giles *et al*. [62] eliminate the neuron with the smallest input connection weights and retrain the network. Pelillo and Fanelli [137] remove a neuron from a layer, and readjust the weights to minimize the effect on the input to the next layer.

Although attractive at first sight, destructive approaches present a series of drawbacks. Firstly, it is necessary to start with an oversized network, slowing down the training process. Secondly, the fact that a weight is small does not necessarily mean that it is not important, and its elimination may even lead to an untrainable architecture. Thirdly, to favor solutions with small weights constrains the training process, preventing the search for solutions with better performance. Fourthly, penalty terms require the specification of additional parameters, whose values are problem dependent, and which may substantially

influence the learning process and the quality of the solution.

2.3.2 Constructive methods

Constructive algorithms start with a small network and, during the training process, new features (neurons or connections) are added to the architecture if the training process stops converging. It seems that such an approach is more promising than that offered by destructive methods. However, constructive procedures face the problem of which features to include, and how to include them in the topology of the network. To cope with this, considerable bias is introduced in the architectures.

For example, in the cascade correlation method introduced by Fahlman and Lebieri [46], a gradient-descent-based algorithm is used to train an initial feedforward network without hidden neurons. If the learning process stagnates, a hidden neuron is created fully connected to the existent ones, to reduce the residual error, and training is resumed. This procedure is repeated by adding new hidden neurons if necessary. The advantage of this algorithm is that it uses incremental training, whereby only the connections of the new added neuron are trained. However, this procedure does not ensure that a small architecture or even a solution will be found. Moreover, there is the problem of why the training process stagnates. It may be due to an inappropriate architecture, or because the training process got trapped in a local minimum due to bad weight initialization. To address this problem Fahlman and Lebieri use a pool of neurons with different sets of connection weights. Each neuron in the pool is provisionally incorporated to the network separately. The network is trained, and the neuron yielding the best network performance is then chosen to be permanently incorporated to the architecture. Obviously, depending on the size of the pool, this procedure can be computationally very expensive.

Decision trees have also been used to build artificial neural networks [17, 112]. The idea is to generate a decision tree to classify a particular data set. The tree is then traversed to produce a *disjunctive normal form* formula [161] for each class present in the data set. Afterwards, a network is built with two hidden layers: one with as many neurons as the

number of distinct literals of the form *attribute ¿ value* present in the formulae, and a second with as many neurons as the disjuncts present in the formulas. The number of units in the input layer is equal to the number of *attributes*, and the number of neurons in the output layer equals the number of classes. Each layer is fully connected to the previous one, with an adequate set of weights and biases that enable the network to make the classification correctly. The idea is attractive, since the architecture is automatically built and even values for the weights and biases are produced (actually the network still has to be trained as its generalization power with the weights estimated using the decision tree is poor). However, the method is only applicable to classification problems, and the networks developed may be unnecessarily big, as the number of hidden neurons grows linearly with the number of internal nodes of the decision tree, and each layer is fully connected to the previous one. Despite these limitations, the method can be used to estimate the maximum size of the network.

The situation is not better for recurrent networks. For example, Fahlman and Lebieri [47] extended cascade correlation to a limited form of recurrent networks, where only self-loops are allowed. Giles *et al.* [61] start training a small fully recurrent architecture. If it is not satisfactory, a new neuron fully connected to the previous ones is created and the training process is resumed, not from scratch, but with the adapted weights of the previous step. Although the authors claim that the procedure reduces the number of training iterations, this is little more than trial and error.

Another limitation of constructive approaches is that once introduced, a bad structural modification can not be automatically removed. In principle, it would be possible to keep track of the sequence of topologies generated, and go back to one of them, should the training process stagnate. But it would be necessary to know at which point in the sequence of architectures, a new feature was wrongly introduced, otherwise it would be a shot in the dark. The situation would be different, if the sequence of architectures could be treated as a population of topologies, and if their different performances in the task being tackled could be used to decide which structural modification should be implemented.

Other constructive algorithms have their applicability limited to binary inputs and binary neurons only (*upstart* [55] and *tiling* [116] algorithms). A general discussion about constructive algorithms and their limitations can be found in [178, 113].

2.4 Summary

In addition to the issues regarding the training process itself, the topology of the network must be defined before any learning procedure can be applied. The alternative provided by destructive and constructive methods constrain the architectures achieved, either from the beginning, or through the structural modifications introduced. The alternative of trial-and-error search blindly generates and tests architectures randomly, and nothing is learned from their relative performance. A more interesting approach would be to test a population of randomly generated topologies, and from these experiments to build different networks, neither randomly nor through the straitjacket of deterministic heuristics, but by taking into account the relative performance of the networks. If an architecture has a good performance (not necessarily a solution to the problem being tackled), it must possess some good features, and it seems reasonable to create new architectures by combining topologies which have already proved their worth. Although it is unknown which are the good features to pick up from the different networks (this might be carried out randomly), there is still a chance that the outcome has an improved performance. This is exactly the idea of evolutionary computation, to be discussed in the next chapter.

Chapter 3

Evolutionary computation

3.1 Introduction

Evolutionary computation is another class of global optimization techniques inspired by biology [14, 88, 13, 36, 50, 11, 9, 8, 53, 179]. Evolutionary algorithms try to mimic the process of biological evolution, by applying evolutionary operators of *selection*,*recombination* and *mutation*, to individuals in a population of potential solutions to a given problem. These operators are stochastically applied to favor individuals of better performance in the task at hand. By repeatedly applying the evolutionary operators to the current population at each generation, a new population of individuals with better performance is created to search the space of potential solutions. Ideally, this process eventually converges to a solution to the problem being tackled.

The evolutionary process is based on the *fitness* of the individual, as measured by its performance in the task at hand. The fitness can be measured by a simple objective function (e.g. the error signal used for training artificial neural networks discussed in Chapter 2), or can result from a performance evaluation in a complex simulation task.

Initially, a population is randomly created. Selection decides which individuals (*parents*) will breed to generate *offspring*. The generated offspring are evaluated in the task at hand,

and then replace the current population or, in some cases, compete with the current population to form the next generation. The basic idea is that the fitness of an individual represents a local evaluation of the search space. Evolutionary algorithms use the information gathered from different points to move the population to a better region of this space, by favoring the best individuals in the reproduction process.

The method proposed in this work is based on a form of evolutionary computation called genetic algorithms, which are now introduced.

3.2 Genetic algorithms

Genetic algorithms (GAs) introduced by John Holland [80] is a form of evolutionary computation which strongly stresses recombination as the driving force of evolution [45, 63, 66, 120, 117, 43, 22, 184, 54]. The structure of a typical genetic algorithm can be described as follows [179]:

0-¿t;
initialize population(s) → P(t);
evaluate(P(t));
REPEAT until solution is found
{
t+1 → t;
selection(P(t)) → B(t);
breeding(B(t)) → R(t);
mutation(R(t)) → M(t);
evaluate(M(t));
survival(M(t),P(t-1)) → P(t);
}
END REPEAT;

where

s is a random generator seed;
t represents the generation;
P(t) is the population at generation *t*;
B(t) is the buffer of parents at generation *t*;
R(t) are the offspring generated by recombining or cloning *B(t)*;
M(t) are the offspring created by mutating *R(t)*

After creating the buffer of parents by a selection procedure, members of the buffer are randomly selected for breeding. Breeding may be carried out by *recombination*, when genetic material of two individuals is exchanged, or by *cloning*, when an individual is simply copied. In the sequence, the generated offspring are mutated and evaluated. Finally, a *survival* step may be implemented to replace the current population with the offspring.

3.2.1 Terminology

As GAs are inspired by natural evolution, specialized terms borrowed from biology are used [120]. In the context of genetic algorithms, individual structures in the population are called *genotypes*. The smallest units of information in the genotype that can be individually inherited by the offspring in the evolutionary process are called *genes*. For example, the genotype may be represented by strings of bits (*bitstrings*), where a *gene* is encoded by a single bit. In the case of parametric optimization using vectors of real-valued parameters as genotype, each parameter may be considered a *gene*. The different values a *gene* can take are called *alleles*. Very often the term *chromosome* is applied to the individuals of the population. This may be confusing, as in living organisms, the total genetic content of an individual is distributed over more than one *chromosome*. Consequently, in this work the term genotype is preferred. Moreover, the genetic content of living organisms is referred to as the *genome* (e.g. the human *genome*), whereas the specific instance of the genetic material of an individual (the *alleles* actually present) is called its genotype. To avoid being pedantic, in this work, the term genotype is used for both, the general genetic structure of the individuals of the population and the individual itself, the context will make clear the distinction.

Before the fitness of an individual can be evaluated, the information encoded in the genotype has to be decoded to build a complete *phenotype*. For example, in the case of the evolution of artificial neural networks, the structure used to encode the elements of the network (neurons, connections, weights) is the genotype, whereas the network itself is the phenotype. In other words, the evolutionary process takes place in a genotype space,

whereas the evaluation process is carried out in a phenotype space. In some representations to be discussed in Chapter 4, a complex decoding process is necessary to transform the genotype into the phenotype.

3.2.2 Selection

Selection is the competition among individuals of the population to become parents of the next generation. Selection procedures are designed to favor those individuals with better performance. Most selection schemes use an intermediate buffer (*mating pool*), whereby fitter individuals have higher probability of being copied to this buffer. Subsequently, pairs of individuals are selected at random from this buffer for recombination or cloning [22, 120].

A good selection method should exert sufficient selective pressure to boost evolution [64]. This is more noticeable later in the search, when the fitness variance in the population is small and low selective pressure may lead to *stagnation* in the evolutionary process. On the other hand, it should not favor highly fit individuals excessively, since early in the search the fitness variance is high, and a group of super individuals may quickly mate and multiply, preventing an adequate exploration of the search space (a problem known as *premature convergence*). A variety of different selection procedures have been used by the evolutionary computation community [120, 11, 63, 117].

For example, a straightforward form of selection is *fitness proportionate*. In this selection procedure, individuals are selected with a probability proportional to their fitness divided by the average fitness of the population. This is the method originally used by John Holland [80]. It is easy to implement, but it does not address the aforementioned issues of stagnation and premature convergence.

An alternative method is *rank selection*, whereby individuals are ranked according to their fitness, and the probability of selection is taken proportionally to the rank rather than to the raw fitness. The method avoids the premature convergence and the stagnation problems.

However, it requires sorting of the entire population at each generation.

To avoid this drawback, *tournament selection* can be used. This method can be implemented by taking a small random sample (the size of the sample is the *tournament size*) of individuals of the population. The fittest individual in the sample is selected and inserted in the buffer of parents, and the sample is returned to the population. The process is repeated until the mating pool is full (the mating pool and the population are of the same size). This selection procedure was adopted in all experiments carried out in this work.

3.2.3 Recombination makes genetic algorithms work

Recombination is the distinguishing feature of genetic algorithms. It is the mechanism by which genetic material of different individuals is combined to create offspring. The theoretical foundation of GAs is based on the assumption that highly fit individuals can be built by assembling good small blocks of alleles (*building blocks* [80, 63]). The objective of recombination is to allow the exchange of genetic material between parents, in order to exchange building blocks, create new ones, and possibly not destroy good ones in the process. Recombination is usually carried out by a *crossover* operator. For example, in fixed length linear representations, *one-point* crossover can be carried out by selecting two parents, choosing a common crossover point in both parents, and swapping the right ends of both genotypes. With a variable length linear genotype, one-point crossover can be implemented by randomly selecting two crossover points, one in each parent, and swapping the left ends of both parents.

In the canonical form introduced by John Holland [80], GAs use bitstrings as representation. In this case, building blocks are interpreted as patterns of bit values in a specified position in the bitstring (the same bit pattern in a different position has a different meaning) which contribute to a higher fitness of the individual. The pattern can be formed by a small group of adjacent bits, or by a big group of bits separated by many bit positions. To describe bit patterns, it is useful to use a template made up of ones, zeros and asterisks, called a *schema*. For example, in the bitstring S = 1010100110, the bit pattern formed by

the 1st, 5th, 7th and 9th bits can be represented by the template A = 1***1*0*1*, where the asterisks indicate that the value of the bits in those positions do not matter. The distance between the two fixed bits in the schema which are farthest apart is the *defining length* of the building block, and the number of bits in the schema is the *order* of the building block. In this case, schema A represents a 4th order building block of length 8.

Any bitstring that fits a schema in its fixed positions is an instance of the schema (e.g. 1101100010 and 1000110111 fit schema A). As a consequence, a population of bitstrings may contain different instances of the same schema, and the average fitness of the individuals in the population which fit a particular schema, may be used to estimate its fitness. On the other hand, a bitstring is an instance of several different schemas (or *schemata*). For example, the bitstring S is an instance of schemas ***01**110, 1***1**110, 1******11*, and many others. Consequently, when a population of bitstrings is evaluated, the fitness of a much larger number of building blocks is actually estimated. This property of GAs is called *implicit parallelism*.

Intuitively, the bigger is the building block and the higher is its order, the more difficult it is to keep its integrity when bits are exchanged between parents to generate the offspring. This suggests that the number of short, low-order building blocks with above average fitness should increase in subsequent generations (this intuitive idea is formalized in the *schema theorem* [80]). This is also valid for other non-binary alphabets. However, its extension to representations other than linear is still a subject for research [117, 142]. In spite of this, it gives support to the idea that genetic algorithms search for ever better individuals by sampling and combining highly fit, short, low-order building blocks to form larger ones in the course of evolution. This is the *building block hypothesis* [64]. Although it is only a hypothesis, accumulated experimental evidence indicates that it is true, and it emphasizes the idea that good evolutionary operators which work on meaningful short building blocks, are critical for the performance of GAs [117].

For example, as long as the representation uses small building blocks, the genetic algorithm will be able to exploit this information by recombining them without disruption. On the

other hand, if good building blocks have their information widely spread over the genotype, they are likely to be disrupted by recombination. Other difficulties may also arise when trying to blindly encode the potential solutions to a problem in a representation. In the ideal case, recombination takes the ”best” part of one parent and combines it with the ”best” part of the other parent, creating a new individual that would be superior to its parents. But there is no way to be sure about that. If the effect of one gene depends on the value of other genes, there may be severe disruptive effects when these are far apart within the genotype. This is called the *epistasis* problem. One may try to put these genes close to each other in the genotype to reduce the problem. But to do so, one must know their effect beforehand, which is seldom the case. Another problem occurs if a building block A has a higher fitness than a building block B, but B contains a low-order building block C with higher fitness than any low-order one in A. In this case, although A is better than B, the search will be misled by an increase in the number of the low-order building block C in the population. This problem is called *deception* [64]. In general, any situation where the simple manipulation of low-order building blocks may mislead the search away from the optimum solution may be called *deceptive* [45]. These issues again highlight the importance of a good representation to solve a particular problem.

One-point crossover is often used in theoretical analysis, but it has the disadvantage that given two individuals, some combinations of their buildings blocks can not be achieved in the offspring. A more flexible option is *two-point* crossover, whereby the linear genotype is considered to be circular, and two segments, one in each parent, are selected and swapped. An extreme case is *uniform* crossover [182], whereby the value of each gene in the offspring is randomly selected from each parent. Uniform crossover is more flexible, in that any combination of genes can be achieved. On the other hand, it is the most disruptive to the building blocks. Several other forms of crossover have been investigated involving linear and non-linear representations (see [117, 27] for an extensive discussion of the subject). However, their advantages and disadvantages depend on the representation and the application. The subject is further discussed in Chapter 4, in the scope of the evolution of artificial neural networks.

The number of offspring created by crossover is calculated proportionally to the size of the population, as a function of a *crossover probability*. The remaining offspring to build an entire new population are created by randomly cloning individuals of the mating pool produced by selection.

3.2.4 Mutation

In genetic algorithms mutation is a secondary operator, used to introduce lost or new genetic material, and to keep genetic diversity in the population [12]. Mutation implements a random change in the value of one or more genes. For example, in a bitstring mutation can be implemented by flipping a bit. For real-valued representations, Gaussian noise may be added to the encoded parameters. Although not biologically inspired, mutation can also be implemented by crossover with a randomly created individual (other forms of mutation in connection with the evolution of artificial neural networks are discussed in Chapter 4). Mutation is applied to the offspring created by recombination and, similarly to crossover, the number of mutation operations is determined by a *mutation probability*.

3.2.5 Survival policy

Some implementations of genetic algorithms introduce a policy for replacing the current population. Usually, the population size is constant and a new population is created at each generation to replace the current one. When the current population is simply replaced by the offspring (even if some of the offspring are simply copies of the parents), the approach is called *generational*. On the other hand, if only part of the population is replaced by the offspring at each generation, the approach is called *steady-state* [120, 164]. In the latter case, usually a small number of offspring is created at each generation to replace the worst individuals in the population, or a competition for survival may be implemented between the offspring and the current population [59, 28, 30, 122, 185].

Steady-state approaches may present higher convergence rates, but the additional selective pressure introduced may result in a rapid loss of genetic diversity in the population, leading the evolutionary process to get trapped at local optima [120, 164]. With generational approaches this risk is reduced, but it may happen that good individuals of the current population are replaced, and simply lost forever. Sometimes, this issue is addressed by implementing an *elitist* strategy, whereby the best individual of the current population is always copied to the next generation.

Which of these approaches is the best for a given problem is an open question, as the overall efficiency of a genetic algorithm is the result of the combined effect of all its mechanisms, and their application [120, 164]. In all experiments carried out in this work, a generational approach has been used, whereby a new population, formed by applying mutation to the individuals generated by crossover and cloning, replaces the entire current population.

3.2.6 Performance evaluation

A necessary instrument for any evolutionary algorithm to work, is a measure of the relative performance of the individuals of the population in the task at hand, in the form of a fitness function. In the case of artificial neural networks, the standard mean square error of the difference between the target output and the output produced by the network is commonly used to evaluate the fitness of the individual. Other alternatives include: a number of successful steps in a control task [193, 124], the output error of the network after training for a specified number of iterations by a learning procedure at each generation [90, 201], the number of relaxation cycles until a recurrent network reaches a stable state [28], etc. In addition, similarly to the destructive methods discussed in Section 2.3.1, penalty terms for the complexity of the network can also be included in the fitness function to favor more parsimonious architectures [202, 41, 30, 7, 28, 166].

3.3 Genetic programming

Any introduction to genetic algorithms would be incomplete without mentioning *genetic programming* (GP). Genetic programming [95, 18, 4] is a special form of genetic algorithm, originally developed to evolve computer programs, which uses structures called *parse trees* as representation. The nodes of the parse tree are the genes, which may be of two types: *functions* or *terminals*. Functions are the *internal* nodes of the parse tree which receive input from other nodes. Terminals are the external nodes (*leaves*), and may be variables or numerical constants representing input. The parse tree is evaluated by assigning values to the terminals and evaluating the nodes of the parse tree in a bottom-up sequence until an output is returned by the root node.

These structures are evolved by a specialized crossover operator, which randomly selects two nodes, one in each parent, and swaps the subtrees which have the selected nodes as roots.

Genetic programming is able to handle variable size structures, although in practical applications, the size of the parse tree is usually bounded to avoid excessive growth [99, 5]. The application of genetic programming to the evolution of artificial neural networks is discussed in Section 4.5.

3.4 Summary

In this chapter, general features of genetic algorithms have been presented. There are several variations on the basic evolutionary operators that can be explored [11, 63, 120, 117]. Nonetheless, the conclusion can be drawn that, as long as a measure of the relative performance of the individuals is provided to guide the optimization process, no constraints are imposed either on the individuals of the population or on the problems that can be addressed. This flexibility makes genetic algorithms attractive for the design of artificial neural networks. However, as recombination is the fundamental feature of GAs, the issue

of how to encode the network in order to allow crossover to preserve and create new good building blocks still remains. It is true that bitstrings are the canonical representation for GAs. However, many different representations have been used in different applications, with encoding mechanisms and operators more natural to the problem being addressed. To discuss this subject in the case of the evolution of artificial neural networks, a literature review is presented in the next chapter.

Chapter 4

Integrating neural networks and genetic algorithms

4.1 Introduction

Genetic algorithms have been used in combination with artificial neural networks in many different ways [191, 187, 98, 52, 200, 143]: for finding new learning rules [24, 32, 154, 153, 126], for optimizing an initial set of weights to be used by a training procedure [23, 133, 91], for evolving the activation function [129], for creating a suitable training data set [173], for pruning the network [73, 196], and for optimizing parameters of learning procedures [23, 156]. Although interesting, these applications do not address the problem of designing a complete network, since they still depend on other methods to build the architecture as well as to adjust the weights.

Genetic algorithms can also be used to optimize either the weights or the architecture exclusively or, more ambitiously, GAs can evolve the architecture and the weights simultaneously, without resorting to a separate learning procedure. Although the subject of the present work is the latter combination of GAs and ANNs, critical issues involving a good representation for evolving the architecture and the weights concurrently also arise when

evolving them separately. Therefore, a literature review on these three combinations of genetic algorithms and artificial neural networks is presented in the next sections. From this literature review, a few lessons are drawn about which conditions the evolutionary operators and the mechanisms for encoding ANNs should meet.

4.2 Genetic algorithms for training

Genetic algorithms have been used for training artificial neural networks by replacing conventional learning algorithms such as backpropagation. As the architecture is fixed, no architecture information has to be included in the genotype; only the weights of the neural network have to be properly encoded (biases can also be represented as weights by including an extra unit of constant output, and connecting it to all neurons of the network). The fitness of the individuals is obtained by assigning the weights to the given architecture, and assessing the performance of the resulting network in the task at hand.

Bitstrings are the canonical representation for genetic algorithms. Bitstrings are sometimes preferred because they are believed to maximize the implicit parallelism of GAs (see Section 3.2.3), allowing more building blocks to be simultaneously evaluated [65]. Binary representations also allow standard crossover and mutation operators to be applied, without the need for the design of tailored genetic operators. A straightforward procedure to encode the connection weights is to assign a specified number of bits to each weight, and to place them in a vector, in a predefined order. The connection weights of a neuron are usually placed next to each other in the genotype, so that they constitute functional units to be evolved by the genetic algorithm [86, 195, 171, 196, 87, 198, 101, 31].

However, there has been considerable debate concerning the appropriateness of using binary encoding in optimization tasks where the search space of the genotype is quite different from that of the phenotype [10, 51, 65, 136]. In the case of encoding the weights of artificial neural networks, this encoding scheme presents several drawbacks:

- A binary substring that represents a weight is highly epistatic (see Section 3.2.3). This makes it difficult to gather useful information about the fitness of groups of bits in a substring, or across the border of weight descriptions, as the fitness of the group of bits is strongly dependent on the values of the other bits in the substring. This undermines the fitness evaluation of small low-order building blocks and may mislead the search by the genetic algorithm. This is an extreme example, where a group of genes is so inextricably interconnected, that they should be treated as a single unit, instead of individually [44].

- Crossover and mutation can introduce big jumps in the search by changing a single bit in the value of the weights (Gray coding can reduce this problem, but does not eliminate it). This reduces the fitness correlation between parents and offspring, making the search harder.

- If few bits are used to encode the weights, the genetic algorithm may be prevented from finding a solution, as some combinations of real values may be impossible (or difficult) to achieve with sufficient precision [65].

- For large networks, the exponential increase in the size of the search space, may render the task intractable, as genetic algorithms are inefficient for manipulating bitstrings with thousands of bits [117]. Successful experiments to train artificial neural networks with long bitstrings (14,036 bits) have been reported by Korning [93], but it is difficult to assess these results without knowing the complexity of the search space (actually a comparison with backpropagation performed by Korning suggests that the task investigated was not really difficult). As a consequence, it is arguable that binary coding does not scale up well with the size of the network [41, 195, 196], and that the approach is impractical, but for small network configurations.

An alternative is to use real-valued weights. In this case, each weight is a gene, and crossover takes place only between weights and not within the weights. This eliminates disruptive effects caused by crossover at the binary level. It also reduces the opportunity for deception, as fewer low-order building blocks are available for manipulation [65].

Additionally, alternative forms of recombination can be used (e.g. linear combination of weights between parents), and no conversion of the binary code to floating point numbers is required for each fitness evaluation. On the other hand, by reducing the dimensionality of the problem, the implicit parallelism provided by the binary representation is reduced, i.e. less information is being obtained to guide the search by the genetic algorithm at each generation. Moreover, if standard crossover is used, those weights present in the population can be recombined, but new values are not created, i.e. the search space is reduced (depending on the problem, this can be an advantage or disadvantage). Although the situation is not clear-cut, the accumulated experimental evidence suggests that this form of encoding is more effective than bitstrings [121, 193, 192, 159, 186, 84, 115, 91, 172, 25].

Genetic algorithms have also been used in combination with constructive algorithms to build the network [144, 145, 134, 2]. In this form, GAs are used for training the architectures created by each structural modification introduced, replacing conventional learning methods, and removing the limitations inherent in them (see Section 2.2). However, this procedure still presents the limitations introduced by constructive methods discussed in Section 2.3.2. Moreover, the repeated training by genetic algorithms after each structural modification introduced by constructive methods is computationally very expensive.

Lesson 1 *Real-valued weights should be preferred.*

4.3 Genetic algorithms for building the architecture

Before any training process can be carried out, the topology of the network must be defined. The definition of the architecture is of paramount importance, as it has great influence on the network performance, as well as on the effectiveness and efficiency of the learning process. As discussed in Sections 2.3.1 and 2.3.2, the alternative provided by destructive and constructive techniques is not satisfactory.

The design of the network architecture can be interpreted as a search in the architecture space, where each point represents a different topology. Searching this space is a very difficult task. Even with a bounded number of neurons, and a constrained connectivity (e.g. feedforward), the search space is huge. Moreover, other characteristics of the search space make things even more difficult. For example, on the one hand, networks with similar structures may have quite different performances, on the other hand, networks with quite different topologies may show similar learning and generalization abilities. Moreover, the performance evaluation is indirect and noisy, since it depends on the training method and on the initial conditions (weight initialization) [118, 200].

The ability of genetic algorithms to search large and complex spaces has been demonstrated in a wide range of tasks. As consequence, they seem to be natural candidates to artificial neural networks design methods. When using genetic algorithms for optimizing the architecture, the connection weights are obtained by training. At each generation the genotype is decoded into the corresponding architecture, weights are randomly assigned to the connections, and the resulting network is trained by a conventional learning algorithm.

The efficiency of the process of building the architectures using GAs is strongly dependent on how the features of the network are encoded in the genotype. In principle, a bitstring representation would be enough for any application. However, it is not necessarily the best approach to evolve the architecture of artificial neural networks. Consequently, a decision has to be made regarding how the information about the architecture should be encoded in the genotype.

According to the degree of network specification, encoding schemes can be broadly classified as *direct* and *indirect*. In *direct* representations (a.k.a. *strong* or *low-level* specification) every component of the network (neurons, layers, connections) corresponds to a specific element in the genotype [158, 118, 108] (the contrary is not necessarily true, as there may be unused parts or redundancies in the genotype). They are very powerful since, within predefined limits, in principle all architectures can be achieved. *Indirect* representations (a.k.a. *weak* or *high-level* specification) encode general features of the architecture, like

number of layers, number of neurons per layer and connectivity pattern (e.g. full connectivity between adjacent layers). Fine details of the architectures are predefined or have some degree of regularity.

In general, the encoding scheme should have enough expressive power to capture the potentially interesting architectures, while excluding flawed or meaningless ones [108]. Unfortunately, both features are usually mutually incompatible or impossible to achieve. On the one hand, to constrain the search, previous knowledge about the potential architectures is required. On the other hand, by excessively constraining the search, interesting solutions may be missed. However, in terms of the evolutionary process, some desirable features of the representation and evolutionary operators can be outlined [15]:

closure - A representation is said to be *closed*, if a genotype always decodes into a valid phenotype according to the specified constraints (e.g. feedforward connectivity). This property can be built into the genotype, or introduced in the genotype/phenotype decoding procedure, by excluding or repairing unacceptable traits from the phenotype (e.g. loops in a feedforward architecture). The evolutionary operators designed to operate on the representation should also possess the attribute of *closure*, by only yielding valid offspring.

completeness - A representation should be able to generate all possible solutions to the problem at hand within specified structural constraints (maximum number of neurons and connectivity pattern).

scalability - In general, *scalability* expresses the ability of tackling ever bigger or more difficult problems. For example, *scalability* can be measured by the increase in the size of the representation by n units (genes), when the phenotype (architecture) is increased by one element (a neuron or a connection). Consequently, the greater the value of n, the less scalable would be the representation, as the search for larger networks would be more difficult, due to the increase in the size of the search space. However, it is complicated to compare the scalability of different encoding mechanisms this way. A representation may show a smaller value of n than another, due

to constraints imposed on the achievable architectures. By reducing the search space this way, the number of achievable solutions may decrease, and the difficulty of the search may actually increase instead of decrease. Moreover, more compact representations may require computationally costly decoding procedures to transform the genotype into the phenotype. Therefore, it seems more reasonable to compare the scalability of different encoding schemes, by the difficulty to find a solution, as measured in terms of the increase in processing time, number of fitness evaluations, or any other suitable parameter to measure computational cost.

There has been a lot of research involving genetic algorithms to evolve the architecture of artificial neural networks. Some representative methods are discussed next.

4.3.1 Linear representations

Linear representations are an example of direct encoding scheme, whereby the elements of the network are organized in a vector. The vector is structured, with the neurons functioning as building blocks, and the crossover operator is designed to respect this organization.

A standard example of this approach is provided by linearizing a *connectivity matrix*. The connections of a network with *N* neurons are represented as a $N \times N$ connectivity matrix, where the entries in each row and column represent the *incoming* and *outgoing* connections of each neuron, respectively. Each entry in the matrix may be 0 or 1, indicating absence or presence of connections, respectively. The matrix is linearized by concatenating the rows (it can be done by column as well [48]), yielding a vector. For feedforward networks only half of the matrix is necessary, whereas for recurrent networks the entire matrix has to be used. The linearization procedure for a feedforward network is illustrated in Figure 4.1.

Miller *et al*. [118] explored this procedure by interpreting each row (which represents a neuron with its incoming connections) as a functional building block, and performing crossover by exchanging the same row between parents. This prevents the creation of flawed individuals (e.g, with no connections from the input layer), as long as they are

not present in the initial population. However, this considerably limits the connectivity patterns achievable, since only connections already present in the initial population can be used, i.e. new building blocks are not created (this could be partially solved by mutating individual entries of the matrix). An alternative proposed by Stepniewski and Keane [180] performs one point crossover in the bitstring resulting from the linearized matrix, allowing the creation of new connections not present in the initial population. It should be noted that the matrix linearization only makes sense if the crossover operator takes advantage of this. With the operator used by Miller *et al*. it is a waste of time to linearize the connectivity matrix, as the operation may be carried out on the matrix directly.

Figure 4.1: (a) Network. (b) Corresponding connectivity matrix. (c) Vector obtained by concatenating the rows of the matrix (only entries below the diagonal are relevant for a feedforward connectivity).

The advantages of the linearized connectivity matrix method lie in its simplicity of implementation, and the fact that in principle any architecture can be achieved within predefined limits. On the other hand, for large networks, the size of the connectivity matrix can increase excessively, making the search difficult for genetic algorithms. This is aggravated by the fact that non-existent elements of the network (e.g. a row of zeros) are still encoded in the genotype, wasting memory and computational power.

This form of representation allows a restricted form of encoding artificial neural networks of different sizes in a fixed size genotype, since sometimes elements can be eliminated from the phenotype without affecting its functionality. For example, neurons which do not send signals to other neurons (indicated by no outgoing connections) can be eliminated from the phenotype. Similarly, in feedforward architectures, neurons which do not have incoming connections can also be deleted, as their output is constant, and their effect can be compensated by changing the biases of other neurons. Non-existent connections are not expressed in the phenotype either. However, this pruning procedure is only carried out in the phenotype, the genotype is kept unaltered. Ideally the elimination of network elements should be carried out in the genotype itself, by the genetic operators directly, improving the search by avoiding the accumulation of parts of the genotype which do not contribute to the fitness of the individual.

For example, Sase *et al.* [166] partially address this issue by adopting a bitstring with a hierarchical structure divided into two parts. The first part is a substring indicating the presence or absence of N neurons in the network by the bit values 1 and 0, respectively. The second part is a substring resulting from the linearization of the connectivity matrix of the network. If a neuron is not present in the network (the corresponding bit is zero in the first part of the genotype), its connections encoded in the second part of the genotype are simply ignored when the individual is evaluated. The authors perform one-point crossover separately in both segments of the genotype. The representation allows the explicit elimination of neurons, but their connections are still kept in the genotype even if they do not contribute to the fitness of the individual. Moreover, a neuron can be completely disconnected and still be encoded in the first part of the genotype.

Another approach proposed by Schiffmann and Werner [169, 170] addresses the issue of deleting elements of the network in the genotype by using a variable size representation. The genotype is an ordered list of units, where each unit represents a neuron with its incoming connections. Each connection is represented by an index indicating the position of the connected neuron in the network. The number of units in the genotype is variable, so that networks with arbitrary size can be encoded. Although the length of the individuals may be different, crossover is performed by selecting a common crossover point (the crossover point is selected within the shorter parent), and swapping the right ends of both parents. This crossover operator guarantees that the offspring are not larger than the biggest of the parents and, consequently, constrains the maximum size of the networks attainable, to the size of the biggest individual in the initial population. Only those elements actually active in the network are really encoded, non-existent connections are not encoded in the genotype, and isolated neurons are deleted from the offspring. However, similarly to the method proposed by Miller *et al*. [118], only connections already present in the initial population are explored. Moreover, the elimination of isolated units in the offspring may lead to more deletions of neurons in the next generation. For example, incoming connections of a neuron transferred from one parent to the offspring, may require a non-existent neuron, which is not present in the offspring because it has been deleted from the other parent in a previous generation. This means that further elimination of connections and possibly neurons may be required. In other words, closure is not a characteristic of this method (it is not clear how the authors deal with this problem).

Lesson 2 *Structured linear representations treat neurons as building blocks, and can represent networks of varying sizes. However, it is necessary to design a representation which allows the complete elimination of network elements by the genetic operators directly.*

4.3.2 Blueprint representations

Blueprint representations are an example of an indirect encoding scheme. The networks are divided into *blocks*. Each block represents a group of neurons, which can be as large as a layer or a group of layers. The genotype is a list of blocks (*clusters*) separated by *markers*, where each block includes the number of layers as well as the number of neurons per layer in the cluster, and one or more *projection fields* to specify connections from the block to other clusters (see Figure 4.2). Crossover may be performed at bit level, with each block represented by a bitstring [75, 76, 74, 158], or may be carried out between blocks only [109, 108].

Blueprint representations are based on a modularity interpretation of the network, as expressed by the clusters (although a cluster may in principle represent a single neuron). If all blocks were reduced to single neurons and projection fields were included for each of their connections, the result would be a direct encoding scheme. The idea behind blueprint representations is to assume that the network can be divided into groups of neurons which share the same connectivity pattern. As a consequence, instead of requiring the encoding of individual connections, only connectivity between groups have to be specified. However, this requires some features to be predefined. For example, it may be assumed that layers within a block are fully connected, and that only neurons in the highest layer of a block make connection to other blocks. In addition, layers within a block are of the same size.

Blueprints are very flexible encoding schemes, as the number and size of the blocks are variable, allowing networks of arbitrary complexity to be encoded. Besides, depending on the number of clusters and the number of projection fields, the representation can be fairly compact, although this is obtained at the expense of assuming some regularity in the network. When the blocks consist of more than one layer, the projection fields specify the areas in the target blocks as a rectangle, defined by the number of layers in the area and the number of neurons per layer, and even the position of the rectangle in the target block is specified by an offset parameter [158]. This is a first step in the direction of interpreting neural networks as two-dimensional structures, an idea fully explored in this thesis.

Figure 4.2: (a) Example of blueprint representation (adapted from [74, 158]). (b) Example of block with projections areas to blocks 2 and 3. (c) Connections projected from block 1 onto blocks 2 and 3. All neurons in the last layer of (black squares) block 1 are connected to all neurons in the projected areas (black squares) in blocks 2 and 3.

4.3.3 Methods based on two-dimensional representations

In this work, the term two-dimensional representation applies to those encoding mechanisms which either encode spatially meaningful parameters, or use evolutionary operators which handle artificial neural networks as two-dimensional structures.

In the first case, parameters for growing a network as a physical entity are encoded in a linear genotype. To a certain extent these methods are similar to grammar-models (see Section 4.3.4), in that, at each generation, the network is constructed from a small embryo. However, in contrast to grammar-models, they are based on the process of spatial growth of *axons* and *branches* in biological neurons.

An example of this approach is provided by Nolfi and Parisi [131, 132, 130], who investigated a growing method with development in time. The genotype consists of a list of *blocks*, where each block contains the instructions to develop one neuron, namely: an *expression gene*, the spatial coordinates of the neuron on a *two-dimensional nervous system*, a *branching angle*, a *segment length* and a connection weight (all connections of a neuron have the same weight). The expression gene is a temporal mechanism for controlling the growth of the neuron's axon and branches. The individual is assigned a number of epochs to live, and the expression gene indicates when during its life time, a particular neuron will be activated by the growth of its axon and branches. Using the angle and segment information, a neuron grows segments (axons) on the nervous system (a two-dimensional lattice), and a connection between two neurons is defined when the axon of a neuron reaches the neighborhood of another. Neurons may have different expression genes and, consequently, may be active or not, at different stages of the individual's life time. The idea is that an individual is not fully developed at birth, i.e. the genotype does not determine a complete individual instantly, the complete phenotype is created through a developmental process. During its life time, the individual is evaluated at different stages of development, and this information is used to compute the fitness of the genotype. Methods which separate evolution of the architecture from the optimization of the connection weights also represent an ontogenetic adaptation of the individual (the weights in this case). The difference is that a

growing method develops the whole individual. It is an interesting idea, although fitness measured by repeated evaluations in the course of a developmental process is computationally expensive. Fujita [58] applied a similar model to fully develop the architecture before the evaluation process takes place.

In the second case, the encoding schemes use a genotype with structure similar to the phenotype (network), they not only interpret artificial neural networks as two-dimensional structures, but also implement a crossover operator to exploit this fact. For example, Arena *et al.* [7] proposed a binary matrix representation to encode the hidden layers and neurons of artificial neural networks. Given a maximum number M of hidden layers and a maximum number N of hidden neurons per layer, a $M \times N$ binary matrix is constructed where the rows are the hidden layers of the network, and each entry in a row represents an active (1) or inactive (0) neuron. The matrices are randomly initialized with a number of active neurons distributed over the layers (see Figures 4.3a and b). Connectivity is predefined (e.g. full connectivity between adjacent layers). Crossover is performed by selecting a common rectangular window in both parent matrices and swapping them (see Figures 4.3c, d, e and f). Obviously, this kind of operation could also be carried out in a linear genotype, by linearizing the matrix by rows. Segments of equal length could be taken from each layer in one individual and swapped with corresponding segments in another individual, but it would be difficult to justify the procedure. The operation only makes sense if the network is visualized two-dimensionally.

Sato and Ochiai [167] proposed a two-dimensional crossover operator based on the interpretation of the network as an oriented graph. All individuals in the population have the same graph structure of layers and neurons per layer. Each neuron contains all information about the connections and weights necessary to compute its output. Crossover is performed by selecting a common random neuron in both parents, and swapping the subgraphs represented by all neurons which are directly or indirectly connected to the selected neuron. That is to say, the authors treat the subgraphs as building blocks. However, the crossover operator proposed by Sato and Ochiai is severely limited by the fact that its does not create new substructures, only those already present in the population are manipulated. The authors

Figure 4.3: (a) Network. (b) Corresponding matrix representation of the neurons in the hidden layers (adapted from [7]). Note that only the number of hidden neurons represented in each row matters, not their distribution in the row. (c) Selected window in the first parent. (d) Selected window in the second parent. (e) and (f) Offspring obtained by swapping the selected windows.

suggest the use of large populations to compensate for this limitation, but this introduces obvious computational costs.

The methods proposed by Arena et al. [7] and by Sato and Ochiai [167] constrain the search to those substructures already present in the population. It may be impossible to achieve a connectivity pattern necessary to solve a particular problem, by exclusively exchanging substructures present in the population. By merging existing substructures to produce new ones, arbitrary connectivity can be achieved, increasing the expressive power of the representation. It may also happen that a good neuron has been evolved, but it is simply in the wrong place. It is possible to wait for the evolutionary process to build the same neuron in the proper place, by using standard evolutionary operators such as one-point crossover. However, it would be more efficient to design a crossover operator which includes the possibility of moving neurons around in the genotype.

For example, a more flexible crossover operator than that proposed by Arena *et al*. [7] might be implemented by exchanging windows of equal size, but selected randomly in both parents. This would lead to neurons from different positions to be exchanged. One might also investigate the possibility of randomly merging the two windows into one, to create an offspring. An even better approach might be obtained by replacing the binary matrix with another one, where each entry would still represent an inactive neuron or an active neuron. But the active neurons in this case would be encoded as lists of incoming connections from other neurons. This would allow the crossover operator based on a common window to evolve the network connectivity. Crossover based on random windows could also be applied, but mechanisms for redefining connections would be required, to avoid violating connectivity constraints (e.g. creation of loops in feedforward topologies). These ideas of merging two-dimensional structures and redefining connections are fully explored by the author in this thesis.

Similarly, the selected subgraphs in the crossover operator proposed by Sato and Ochiai could also be randomly merged into a single one, which would replace one of the selected subgraphs in the offspring (a copy of one of the parents). Once more, this alternative

suggests merging of two-dimensional structures as an idea to be further explored in this work.

The method introduced by Nolfi and Parisi can, in principle, generate arbitrary connectivity, but the effect of the parameters encoded in the genotype is hard to predict, due to the developmental process needed to generate the network (a feature also common to grammar-methods, to be discussed in the next section).

Lesson 3 *Crossover operators can be defined to operate with two-dimensional structures, but they should be able to create new substructures by merging existing ones.*

4.3.4 Grammar-based methods

Grammar-based methods are indirect representations based on the idea of building complex structures by repeatedly applying instructions to rewrite elements of a simple initial structure. A common form of this technique is provided by *L-systems*, originally introduced by Lindenmayer as a mechanism to describe the development of plants [102, 147]. L-systems start with an initial character (*axiom*) and, recursively and in parallel (simultaneously), apply instructions (*production rules*) for rewriting characters, for a specified number of cycles. A simple example can be used to illustrate the idea.

Suppose that there are two characters *a* and *b*, and that there are two production rules:

$a \rightarrow ab$
: Meaning that the character *a* is to be replaced with string *ab*.

$b \rightarrow a$
: Meaning that the character *b* is to be replaced with character *a*.

The component of the rule to the left of the arrow is called *predecessor*, and the component to the right of the arrow is called *successor*.

In the hypothesis that the axiom is the character *a*, the following sequence of strings would be produced after three rewriting cycles:

a
ab
aba
abaab

These production rules constitute a *context-free* grammar, as their application is independent of the characters before and after the *predecessor*. A more sophisticated approach takes into account the interaction between neighboring characters in the string by using *context-sensitive* rules.

For example, suppose that there are three characters *a*, *b* and *c*, and that there are three production rules:

$a \rightarrow ab$
: The character *a* is to be replaced with string *ab*.

$ab < b \rightarrow ca$
: The character *b* is to be replaced with the string *ca*, only if it is preceded by the string *ab*. The string *ab* is the *left context*.

$b < c > a \rightarrow aab$
: The character *c* is to be replaced with the string *aab*, only if it is preceded by *b* and followed by a *a*. In this case, a *left* and a *right context* are present, represented by the characters *b* and *a*, respectively.

In this case, the following sequence of strings is obtained after the application of four rewriting cycles to the axiom *a*:

a
ab
abb
abbca
abbcaaabab

These ideas have been applied to the evolution of artificial neural networks [89, 90, 26, 201], by using genetic algorithms to evolve production rules, and using characters with

appropriate semantics to allow the interpretation of the generated string as an artificial neural network. By encoding the rules in the genotype, the search is carried out in the space of transformation rules, instead of in the space of networks. This is biologically appealing, as the development from genotype to phenotype controlled by the transformation rules may vaguely resemble the development of the embryo controlled by the DNA in nature. It is argued that grammar-based methods show better scalability than direct encoding methods, and also that they contribute for the formation of regular patterns of connectivity [89, 90, 201]. To illustrate these and other issues, it is worth discussing a specific example, the method proposed by Kitano [89, 90].

Kitano used a context free grammar to convert characters into square matrices of characters. Figures 4.4a and b show an example of a grammar. The rules in Figure 4.4b are fixed and do not take part in the evolutionary process, whereas the rules in Figure 4.4a are encoded in a linear genotype as shown in Figure 4.4c. The characters in the rules are actually encoded as bit patterns, so that the genotype is a bitstring, and crossover is carried out at bit level. Starting from a single character *S*, each character in the matrix is replaced by a 2×2 matrix according to its rule. This transformation process is shown in Figures 4.4d-f. The rewriting process results in a binary matrix, which is then interpreted as the connectivity matrix of an artificial neural network. In the case of feedforward connectivity, the network corresponding to the matrix in Figure 4.4f is shown in Figure 4.4g.

It is clear from this example, that large networks can be built by simply increasing the number of rewriting cycles. Consequently, large networks can be represented by small genotypes. The ability of grammar-based methods to create large structures by the repeated application of transformation rules is certainly an advantage, since for complex networks, the search space becomes huge and intractable for any search method. This should, in principle, improve scalability in comparison to direct encoding representations. Indeed, Kitano [89] reports on results of experiments with the encoder/decoder problem [160] to compare the performance of direct and indirect approaches, and concluded that on the experiments performed, the matrix rewriting method consistently outperformed the direct encoding method used for comparison. However, Siddiqi and Lucas [175] also investigated

the issue of scalability, and achieved results which contradict those obtained by Kitano in the same task. Moreover, the reduction in the size of the genotype is obtained at the expense of expressive power, i.e. not all architectures can be achieved by using rewriting procedures. As a consequence, small changes in the genotype may lead to large modifications in the phenotype. This adds to the already difficult task of using information collected in the phenotype space, where the evaluation actually takes place, to guide the evolution in the search space of genotypes.

For feedforward artificial neural networks, the generation of a connectivity matrix to represent the network is inefficient, as large portions of the genotype are not expressed in the phenotype. For example, it can be realized from Figure 4.4c, that 28% of the positions in the genotype were not used in the construction of the network in Figure 4.4g. This is due to the fact that only the upper half of the connectivity matrix is actually used to build a feedforward network. However, this drawback is not a general characteristic of grammar-based methods. For example, Boers and Kuiper [26] used a context-sensitive grammar which does not generate a connectivity matrix. The authors used characters which are interpreted as neurons and connections, to build the network directly. Nonetheless, the components of the rules (*left* and *right* contexts, *predecessor* and *successor*) are not of fixed length as in Kitano's case, they are defined by *marker* characters. As the characters are encoded as bit patterns, rules can appear and disappear due to the redefinition of characters by the evolutionary operators. This may leave large segments of the genotype unused, and also leave production rules undefined. In addition, with Kitanos's method as well as with Boers and Kuiper's, there may be duplicated or unused rules (unused because they are not activated by any *predecessor*) in the genotype.

Another issue sometimes neglected is the complicated developmental process from genotype to phenotype which has to be executed before the fitness of the individual can be evaluated. This may correspond to a great fraction of the time consumed in the fitness evaluation. It also makes the connection between elements in the genotype and features in the phenotype obscure. Also, it is virtually impossible to impose constraints to the architecture if desired.

Figure 4.4: Illustration of Kitano's grammar-based method (adapted from [89, 120]). (a) Evolvable transformation rules to rewrite characters into square matrices. (b) Fixed transformation rules. (c) Evolvable rules encoded in a linear genotype. (d) Result of the first rewriting cycle. (e) Result of the second rewriting cycle. (f) Resulting connectivity matrix. (g) Resulting network after interpreting the connectivity matrix in (f), by reading the entries of each row above the diagonal as outgoing connections of each neuron.

Finally, as grammar-based methods do not explicitly encode all the information of the network in the genotype, they are usually (not always) combined with computationally expensive training procedures for adjusting the weights at each generation. If no training process takes place, only those values of weights included in the genotype are used to build the whole network. This leads to the creation of repeated patterns of weights all over the network as a consequence of the rewriting cycles. This is the case, for example, of a technique called cellular encoding, to be discussed in Section 4.5.

Lesson 4 *Grammar-based methods could benefit from an encoding procedure to represent the weights, allowing them to be evolved simultaneously with the architecture. The encoding scheme should be independent of the size of the network, as this is the basic appeal of grammar-based methods.*

4.4 Evolving the architecture and the weights

Although biologically plausible, the evolution of the architecture combined with a separate training procedure has severe drawbacks. Firstly, if training is carried out until complete stagnation of the learning process, it is computationally very expensive [194, 125, 76, 118, 196]. If partial training is used, evolution may be misled by poor performance evaluation. Secondly, the evaluation process depends on the initial conditions. As a result, a good architecture may be discarded due to an unlucky initial set of weights. Thirdly, all limitations of training methods discussed in Section 2.2 apply. For these reasons, methods to evolve the architecture and the weights concurrently have been proposed. Most of them are extensions of the methods to evolve the architecture discussed in Section 4.3, with the inclusion of the connection weights in the genotype. As a consequence, the search space for the evolutionary process is considerably increased, and it is an open question whether this approach is superior or not. However, the efficiency of the search is considerably affected by the evolutionary operators used. Consequently, it is the task of the designer to develop specialized evolutionary operators based on a suitable representation, to efficiently evolve the architecture and the weights simultaneously.

Similarly to the evolution of the architecture, bitstrings can also be used to evolve the topology and the weights simultaneously, by encoding the network as a bitstring with a predefined structure and length, i.e. by allocating a certain number of bits to each feature of the network, layers, neurons, connections and weights.

For example, Dasgupta and McGregor [41] represent the connectivity of *N* neurons using a $N \times N$ binary matrix, where each row and column represent the outgoing and incoming connections of each neuron, respectively. The matrix is linearized by concatenating the relevant part of the rows (for a feedforward network only elements above the diagonal are encoded), and attached to a list of weights. One bit is used to encode each connection, and a specified number of bits encodes each weight. This separation of the connection space from the weight space in the genotype makes it more difficult for the offspring to inherit useful combinations of connections and weights from their parents. Maniezzo [110] improves on this, by interpreting each neuron as a functional unit and placing all information relevant to evaluate its output together in the genotype. Maniezzo linearizes the connectivity matrix by column, and places each weight next to the corresponding connection bit. In both methods all connections allowed by the connectivity (feedforward or recurrent) are represented, which is inefficient. To avoid this drawback, Saha and Christensen [162] encode sparse networks as a list of *sections*, where each section includes a neuron identifier linked to a list of *connection fields* and the bias, and each connection field includes the identifier of the connected neuron and the weight of the connection. In the three aforementioned methods, standard one-point crossover is used to evolve the population.

Although simple to implement, bitstring representations suffer from the same size limitations as the binary encoding schemes for training discussed in Section 4.2. To minimize this problem Dasgupta and McGregor [42] replaced the previous binary encoding with a real-valued representation of the weights, resulting in a hybrid genotype. The first part is still the linearized binary connectivity matrix, whereas the second part encodes a list of floating point numbers. However, the number of neurons encoded in the genotype is still fixed (not necessarily active in the network, as disconnected neurons can be present).

In order to explicitly delete elements of the network by the evolutionary operators, Tang *et al.* [183] introduced a *hierarchical* representation to build multilayer feedforward artificial neural networks. In his method, all networks in the population can have a maximum number of hidden layers of equal size. The genotype is divided into three levels: the first level enables or disables layers, the second level enables or disables neurons, and the third level stores the weights. The first level is a bitstring of length equal to the maximum number of layers, where a layer is represented as active (1) or inactive (0). The second level is also a bitstring representing the active (1) and inactive (0) neurons present in each layer. In the third level all the weights corresponding to the maximum connectivity possible (all layers and neurons active and full adjacent layers fully connected) and biases are stored as floating point numbers (see Figure 4.5). One-point crossover is performed separately at each level. Obviously, the approach does not favor sparse connectivity, as connections can only be eliminated if neurons or entire layers become inactive. Moreover, flipping of a single bit by crossover or mutation can render neurons or whole layers inactive, leading to big jumps in the search space.

Figure 4.5: Hierarchical representation to build artificial neural networks (adapted from [183]).

To avoid these problems a scheme such as the *marker-based* representation proposed by Miikkulainen *et al*. [59, 123] can be used to evolve feedforward as well as recurrent networks. In this method, the genotype is a circular list of *segments* delimited by *start* and *end markers*, where each segment represents a neuron with its identifier, incoming connections (indicated by the identifier of the connected neuron), weights and initial output values. The number of segments is fixed. Each field within the genotype (including the markers) is interpreted as an integer, and depending on its value, the field can represent a start marker or end marker. The interpretation of the other fields depends on their position relative to the start marker. Start and end markers are necessary because the end of a segment does not automatically lead to the beginning of another, i.e. there may be unused chunks of the genotype between markers (this problem has already been discussed in connection with grammar-based methods in Section 4.3.4). Standard two-point crossover is carried out, and mutation is implemented as random deviation on the integer values.

The method proposed by Miikkulainen *et al*. is intended to allow a variable number of neurons. To accomplish this, the mutation operator may change the value of the start and end markers, by changing their semantics (e.g. start maker can become an end marker and vice-versa), resulting in deletion or addition of neurons or connections. However, this procedure may lead to infeasible connections, as the addressed neuron may not exist anymore, resulting in a flawed network. Besides, due to crossover and mutation, duplicated segments may be created in the genotype. Notwithstanding these limitations, the method has the interesting feature that networks of different sizes can be encoded within a genotype that only limits the maximum number of neurons (constrained by the maximum number of segments possible).

Lesson 5 *Most linear representations treat neurons as natural building blocks, by organizing the elements necessary to compute their output close together in the genotype.*

4.5 Genetic programming and artificial neural networks

Although genetic programming is a powerful tool, its application to the evolution of artificial neural network has been limited by the lack of a good encoding mechanism. It is true that GP can be used to evolve artificial neural networks by using functions to represent neurons and weights, and terminals to represent the input to the network [96, 95, 101, 188, 202, 203, 204, 34]. However this approach is not so straightforward as it seems. A simple example of architecture encoding can illustrate the point. The network in Figure 4.6a can be encoded as the parse tree in Figure 4.6b. However, the parse tree in Figure 4.6b does not necessarily induce the network in Figure 4.6a. From Figure 4.6b one can only infer the network in Figure 4.6c, where the number of hidden and output neurons equals the number of functions in the parse tree. As a consequence, genetic programming may generate topologies considerably larger than necessary.

This inconsistency in the genotype/phenotype conversion mechanism is inherent to parse tree representations, resulting from their limited ability to encode oriented graphs. Consequently, the possibility of using genetic programming to evolve artificial neural networks with a direct encoding procedure is quite limited. An alternative is to use an indirect representation. For example, Gruau [70] developed a *grammar-based* method, where transformation rules to write graphs are encoded in *grammar trees*. The technique, called *cellular encoding*, has been applied to the evolution of the architecture and the weights of artificial neural networks [70, 69, 68, 194, 71, 146, 57, 56].

In cellular encoding, the nodes of the grammar tree are symbols representing rules for performing transformations on *cells*. The cells can be linked to each other forming an oriented graph. Each cell contains registers to store information such as connection weights, bias, a *recurrent counter*, and a *reading head*. The transformations include: cell division and modification of the cell registers. In a *sequential* division, a parent cell is replaced with two child cells: the first child inherits the input links of the parent cell, while the second one inherits its output links, and the first child is connected to the second child. In a *parallel* division, both child cells inherit the input and the output links of the parent cell. The

Figure 4.6: An example of parse tree representation of the architecture of an artificial neural network. *X1* and *X2* are variables representing input to the network. (a) Network. (b) Parse tree representation of the network in (a). (c) Network decoded from parse tree in (b).

cell division transformations allow the building of the architecture. To assign values to the bias and connection weights, instructions with parameters are used to modify the bias and weight registers of cells [69, 68, 194, 71].

The building of the network is carried out by starting with a single cell connected to the input and output of the network to be developed. The cell reading head points to the root of the grammar tree. At each step of the developmental process, each cell executes the graph transformation pointed to by its reading head, and advances the reading head to the left or right subtree (if the transformation was cell division, the first child points to the right and the second one points to the left). The order in which cells execute graph transformation is determined by a First In First Out (FIFO) queue. Once a cell executes its transformation, it enters the FIFO queue, and the next cell to execute its transformation is the head of the queue. The registers of the parent cell are copied to the child cells. When the reading head of all cells executes the operations indicated by the terminals at the leaves of the grammar tree, the development is complete, the cells become neurons and the links between cells become connections between neurons. The connection weights and biases are read from their cell registers. The grammar tree is the genotype of the individual, and is evolved by genetic programming [69, 68, 194, 71].

The set of transformation rules can be enriched, for example, by other forms of cell division, instructions for manipulating connections, and especially by an instruction for resetting the reading head of the cell to the root of the grammar tree. The recurrent counter specifies how many times the cell reading head can be reset and, similarly to other grammar-based methods discussed in Section 4.3.4, this resetting procedure allows the construction of large regular structures, by the repeated application of the same transformation rules encoded in the genotype.

Cellular encoding is an interesting approach, with some advantages over the grammar methods discussed in Section 4.3.4. Firstly, all transformation rules encoded in the genotype are necessarily used in the developmental process to generate the network. Secondly, it is a variable size representation, allowing the encoding of any number of transformation

rules. Thirdly, instructions to evolve the weights and biases are included in the genotype. Fourthly, with the adequate set of transformation rules, any type of architecture can in principle be generated. However, cellular encoding has also drawbacks.

For example, originally, Gruau [70, 69] applied the method to the evolution of artificial neural networks with binary weights in boolean classification problems. In [194, 71], an instruction was introduced to represent real-valued weights within a certain range. When executed, the instruction assigns values (given by integer parameters) to the weight registers (each weight instruction assigns values to weight registers of one cell). Unless such an instruction is executed for each cell, some of the connection weights receive default values, and these values can be copied all over the network, by the repeated use of the grammar tree, provided by the recurrent register. As the appealing feature of grammar-based methods is the ability of creating large structures from a relatively small genotype, it is clear that this may lead to a poor diversity of connections weights in the resulting network. In [70, 69], Gruau suggested the application of a limited form of learning for adjusting the connection weights in the case of multiple readings of the grammar tree. The procedure, coined *developmental learning*, uses a standard learning algorithm in the first reading of the grammar tree. The new weights are incorporated into the individual for the subsequent readings of the grammar tree, but they are not included in the genotype to take part in the evolutionary process.

In addition, similarly to Kitano's method, the creation of large structures by the repeated reading of the grammar tree, implies that only some architectures can be achieved, and also that small changes in the genotype may lead to large modifications in the phenotype. Cellular encoding is in principle a variable size representation, but in practice the size of the grammar trees has to be constrained in order to prevent their excessive growth.

Another problem results from the in-depth reading of the grammar tree controlled by the FIFO queue, in that the same subtree may have a complete different effect on the resulting network, depending on its position in the grammar tree. When subtrees are swapped by crossover, the networks resulting from decoding the offspring, may be completely different

from the networks resulting from decoding the parents. In other words, the grammar tree representation is strongly epstatic.

Cellular encoding also tends to generate highly interconnected networks. This is a result of the cell division process, where the child cells inherit all the links of the parent cell. Instructions to eliminate connections can be implemented, but this may lead to the systematic elimination of connections in the network due to the recurrent reading of the grammar tree.

Lesson 6 *The direct use of parse trees to represent artificial neural networks presents serious drawbacks. Some of the limitations can be bypassed by using genetic programming to evolve rules for building the network. However, GP-based methods still lack a good mechanism for representing the weights.*

4.6 Summary

Whether a direct or an indirect representation should be preferred is still an open question. For problems where a high degree of regularity is expected from the solution, indirect encoding approaches have some advantages. But if unconstrained weight distribution and specific constraints on the architecture are desired, direct encoding methods are necessary. Most direct (and some indirect) approaches treat neurons as the basic functional units, by putting all necessary information to compute their output close together in the genotype. The ability to move neurons around in the genotype is an interesting feature to be explored in the design of a crossover operator. But it may require the redefinition of connections to comply with connectivity constraints, a procedure difficult to justify unless the network is visualized as a two-dimensional structure, and an appropriate interpretation is given to the operation. To address this issue, on the grounds of the lessons learned, a new method based on a two-dimensional representation is proposed in the next chapter.

Chapter 5

Two-dimensional representation

5.1 Introduction

The lessons learned from the literature review presented in Chapter 4, allow the following *criteria* for a good representation to evolve artificial neural networks to be outlined:

criterion 1 - It is important to constrain the maximum size of the structures being evolved, either operationally or by design.

criterion 2 - To improve scalability, the connection weights should be represented as real-valued parameters.

criterion 3 - The representation should allow specialized evolutionary operators to explore the two-dimensional structure of ANNs.

criterion 4 - Neurons are natural building blocks of artificial neural networks. The evolutionary operators should be able to evolve them by evolving their connectivity, and also to explore new arrangements of the evolved building blocks in the network.

criterion 5 - The evolutionary operators must be able to change the size of the network without exceeding the specified maximum limits.

The literature review also made it clear that current methods do not meet these criteria (at least not all of them). Some of them do not even fulfill the general requirements of closure, completeness and scalability. Also, the previous methods do not satisfactorily explore the two-dimensional nature of artificial neural networks. So, a new method must be designed to meet all these requirements.

In [140], Poli introduced *Parallel Distributed Genetic Programming* (PDGP). In PDGP, instead of the usual parse tree representation of genetic programming, a graph representation is used, where nodes are allocated in a two-dimensional grid of fixed size and shape which constitute the genotype. The grid is particularly useful to evolve structures, like artificial neural networks, which are oriented graphs with a natural layered structure. Preliminary results obtained by Poli [141] suggested that with specialized operators and meaningful building blocks, the grid representation of PDGP could be used to evolve ANNs.

Bearing in mind the aforementioned criteria, in the next sections a new method inspired by PDGP is proposed for the evolution of artificial neural networks. The new method uses the grid representation of PDGP to define a new specialized crossover operator to evolve artificial neural networks.

5.2 Representation

The genotype is an ordered list of nodes. The nodes may be of two kinds: *terminal* or *neuron*. In the first case, the output of the node is a variable containing an input to the network. In the second case, the node represents a processing element of the encoded network. When the node is a neuron, it is represented as a list containing all the information to compute its output, namely: bias, incoming connections and weights. Connections are represented by indexes, indicating the position of the connected nodes in the genotype (see Figure 5.1). The order of connections in the list describing the neuron is irrelevant. Complying with criterion 1, all individuals in the population have the same number of nodes to constrain the maximum size of the network.

Figure 5.1: Genotype and neuron description.

This linear genotype is not much different from the linear representations discussed in Section 4.4. However, for the application of the crossover operator (see Section 5.3), the linear representation just described is interpreted as a two-dimensional arrangement of columns and layers (*grid*). The nodes of the linear genotype are mapped onto the grid according to a *description table*, which defines the number of layers and the number of nodes per layer. The description table has the same number of nodes as the linear genotype, and all individuals in the population use the same table. It is a feature of the population, and is not included in the genotype. The layers and columns of the grid are treated as circular entities, leading to a toroidal grid (the reason for this design will become clear in the description of the crossover operator). For example, by using the description table in Figure 5.2b, the individual in Figure 5.2a is interpreted as the two-dimensional representation in Figure 5.2c, where connections are indicated by links between nodes.

Any number of layers of different sizes are allowed. Therefore, by using different description tables, different grid representations might be obtained. For example. according to the tables in Figures 5.3b and d, the same individual in Figure 5.3a can be converted into the two-dimensional representations in Figures 5.3c and e, respectively. Note that both grids have the same total number of nodes, distributed over a different number of layers, with a different number of nodes per layer. That is to say, each description table defines a different two-dimensional interpretation for the linear genotype. No matter what is the network to

Figure 5.2: (a) Example of genotype. Variables *X1* and *X2* are terminals representing input to the network. (b) The table describing the number of layers and the number of nodes per layer of the grid. (c) The two-dimensional representation resulting from the mapping of the nodes of the genotype in (a), according to the description table in (b).

be evolved, it must comply with the shape constraints specified by the description table.

The nodes in the first layer (input layer) are necessarily terminals representing input to the network, whereas the nodes in the last layer (output layer) are necessarily neurons, returning the output of the network. The number of input and output nodes depends on the problem to be tackled. The remaining nodes, called *internal* nodes, constitute the *internal layer(s)*, and they may be either neurons or terminals. As a consequence, although the size of the genotype is fixed for the entire population, the networks represented may have different sizes. This happens because terminals may be present as internal nodes from the

Figure 5.3: (a) Example of genotype. (b) Description table with three layers: 2 nodes in the input layer, 3 in the second, and 1 in the output layer. (c) Two-dimensional representation produced by interpreting the genotype in (a) according to the description table in (b). (d) Description table with four layers: 2 nodes in the input layer layer, 2 nodes in the second, 1 in the third, and 1 in the output layer layer. (e) Two-dimensional representation resulting of the interpretation of the genotype in (a) according to the description table in (d).

beginning, or may be introduced by crossover and mutation (this is discussed in Sections 5.3 and 5.4). To visualize the actual network encoded in the genotype, connections from terminals in the internal layer can be replaced (and the corresponding terminals removed) with connections from corresponding terminals in the input layer. Subsequently, multiple connections from the same node can be merged into a single one, by adding up their weights (see Figures 5.4a, b, c and d).

There is no restriction about the connectivity whatsoever. Connections between non-adjacent layers, and connections within the same layer as well as recurrent connections are allowed, and even multiple connections between the same nodes are permitted (see example in Figure 5.5).

As the number of nodes in the genotype is constant, the representation meets, at least partially, criterion 1 outlined for a good representation. However, since multiple connections are allowed, the genotype may grow excessively, reducing the efficiency of the search. The problem could be eliminated by disallowing multiple connections. But multiple connections are important, since they increase the range of the connection weights. So, a compromise solution was introduced by limiting the maximum number of multiple connections.

Meeting criterion 2 for a good representation, weights and biases are encoded as floating point numbers. The biases could have been implemented as weights of connections from an additional node of constant output, to be evolved as an ordinary connection weight. But, as it will become clear in the description of the crossover operator, it is irrelevant whether it is treated as a separate parameter or not.

The grid representation is very similar to the phenotype. To compute the network output no decoding procedure is necessary. The transformation from genotype to phenotype presented in Figures 5.4a, b, c and d is only for visualization purposes. The output of the network can be directly obtained by assigning input values to the terminals, and evaluating the neurons according to their order in the genotype.

Figure 5.4: (a) Linear genotype. Note that node 5 contains the same terminal as node 1. (b) Grid description table. (c) Resulting two-dimensional representation. (d) Corresponding network after merging the connections between node 5 and 6, with the connection from node 1 to 6, by adding up their weights and removing node 5. Note that, due to the presence of a terminal in the internal layer, the resulting network has only two hidden neurons, whereas the genotype has three internal nodes.

Figure 5.5: Example of feedback and multiple connections. (a) Genotype. (b) Description table. (c) Two-dimensional representation.

5.3 Crossover operator

The network representation described in the previous section is a two-dimensional graph representation. Consequently, it is natural to define a crossover operator based on this structure (see criterion 3 for a good representation). Moreover, neurons are functional units of artificial neural networks, and it is important that recombination not only reorganizes these natural building blocks, but also creates new ones (see criterion 4). Therefore, in this work, a two-dimensional crossover operator is proposed, which works on two levels: at the level of the grid and at the level of the neuron.

The crossover operator works on the two-dimensional interpretation of the parents, by randomly selecting a node *a* in the first parent and a node *b* in the second parent (see Figure 5.6a), and replacing node *a* in a copy of the first parent (the offspring). Depending on the types of node *a* and node *b*, the replacement is carried out as follows:

Both nodes are terminals: This is the simplest case, node *b* replaces node *a*, and there is no change either in the topology or in the weights of the network.

Node *b* is a terminal and node *a* is a neuron: In this case, node *b* also replaces node *a*, but the complexity of the network is reduced, because a neuron is removed from the network, meeting criterion 5.

Node *b* is a neuron and node *a* is a terminal: In this situation, the crossover operation increases the complexity of the network, by replacing a terminal with a neuron and increasing the number of hidden neurons in the network (meeting again criterion 5). But before node *b* replaces node *a* in the offspring, each of its connections is analyzed and possibly modified, depending on whether they are connections from terminals or neurons.

This can be visualized in terms of graphs. As artificial neural networks are oriented graphs, a neuron is a special case of subgraph (see Figure 5.6b), where the arcs of the subgraph represent incoming connections to node *b*. The idea is to pick up the subgraph represented by node *b* to replace node *a* in the offspring. But the subgraph is not transferred rigidly to the offspring, its arcs are modified according to whether they originally represented connections from terminals or neurons.

- If the connection is from a neuron, the arc corresponding to this connection is ”deformed”, so that the arc still represents a connection from the same node, when node *b* is transferred to the position of node *a*. For example, the arcs representing connections from node 9 and 13 are deformed to still represent connections from these same two nodes (see Figures 5.6b and c),
- If the connection is from a terminal, the corresponding arc is not modified when node *b* is transferred to the position of node *a* (e.g. the connection from node 10). If the arc leads to a connection from a non-existent node when node *b* is transferred to the position of node *a*, the arc is wrapped around the layer (remember that layers are circular). For example, the arc corresponding to the connection from node 8 is wrapped around to become a connection from node

1 (see Figures 5.6b and c). The same wrapping-around procedure is applied to the columns (the grid is toroidal), if a node below the input layer or above the output layer is required.

After this transformation of arcs, the transformed subgraph is transferred into the position of node *a* in the offspring (see Figure 5.6d). Of course, all this manipulation of connections only makes sense, by considering the network as a two-dimensional structure.

This graph transformation is actually carried out in the list which represents node *b*, by redefining the indexes representing the connected nodes. This redefinition of indexes is described as follows:

- If the connection is from a neuron, the index of the connected node in the list describing node *b* is not modified. That means, the connection will still be from the same node after node *b* replaces node *a* in the offspring.
- If the connection is from a terminal, the index is modified to point to another node, as if the connection had been rigidly translated from node *b* to node *a*. That means, the same horizontal and vertical displacement that exists between nodes *a* and *b* is applied to translate horizontally and vertically each connection of node *b* coming from a terminal. For example, in Figure 5.7, node *a* is one layer below and one column to the right of node *b*. Consequently, the connection between node 10 and node *b* is transformed into a connection between node 7 and node *b* (see Figures 5.7b and c). Should the rigid translation of the connection point to a non-existent node outside the limits of the layer or column, the index of the connected node is modified as if the connection had been wrapped around the layer or column. For instance, the connection between node 8 and node *b* is transformed into a connection between node 1 and node *b* (Figures 5.7b and c).

Here a distinction between recurrent and feedforward connectivity must be made. After the transformation of connections in node *b*, loops might be created in the offspring. If a feedforward architecture is desired, connections from a higher order node to a lower one are deleted.

This procedure for connection inheritance aims at preserving as much as possible the information present in the connections and weights.

Both nodes are neurons: This is the most important case. By combining two neurons, the topology and the weights of the network can be changed. After implementing the modifications in the connections of node *b* as described in the previous case, nodes *a* and *b* are combined. This is carried out by selecting two random crossover points, one in each node, and replacing the connections to the right of the crossover point in node *a* with those to the right of the crossover point in node *b*. This creates a new node to replace node *a* in the offspring (see Figure 5.8). Note that the crossover point is only selected between connections, it does not separate the weight from the index of the connected node.

This process can easily create multiple connections between the same two nodes. They are very important because their net effect is a fine tuning of the connection strength between two nodes. If more than the allowed maximum number of multiple connections are created, some of them are deleted before the replacement of node *a* in the offspring.

The crossover operator just defined, allows nodes to be transferred from one position to another in the grid (meeting criterion 4). However, this procedure requires connections to be redefined, otherwise many of them would have to be deleted due to connectivity constraints, leading to loss of genetic material. The mechanism for modifying connections described, was designed to adress this issue. Although other forms of dealing with connections are possible, preliminary experiments with the odd-2 (XOR), odd-3, odd-4, and odd-5 parity problems (in the odd-*n* parity problems, the network is required to return 1 if there is an odd number of 1s in the input, and 0 otherwise), led to the conclusion that

Figure 5.6: (a) Two-dimensional representation of the parents. For clarity, only connections relevant to the operation are shown. (b) Subgraph representing node *b*. (c) Transformed subgraph. (d) Offspring generated by transferring the transformed subgraph to the position of node *a*.

Figure 5.7: (a) Two-dimensional representation of the parents. (b) Node *b*. (c) Copy of node *b* with modified connections. Connections of node *b* whose indexes indicated connections from terminals received new indexes. (d) Offspring generated by replacing node *a* with modified node *b*.

Figure 5.8: Combination of two neurons. (a) Node *a*. (b) Node *b* . (c) New node created to replace node *a* in the offspring. Note the multiple connections created.

it is more effective to deal with connections from neurons and connections from terminals differently. Four different variations of crossover were investigated:

Variation 1: This is the crossover operator just described. The results are summarized in Table 5.1. Column 2 represents the average number of generations (standard deviation in brackets). Columns 3, 4, 5 and 6 show the minimum, average, maximum and standard deviation for the number of neurons of the networks evolved, respectively. Columns 7, 8, 9 and 10 give the minimum, average, maximum and standard deviation for the number of connections of the networks evolved, respectively. Column 11 shows the computational effort, i.e. the minimal number of fitness evaluations necessary to obtain a solution with 99% probability in repeated runs [95].

Variation 2: In this form, all connections of node *b* are rigidly transferred. In the four tasks, the performance of this variation, as measured by the computational effort, was inferior to that of the previous form (see Table 5.2).

Variation 3: In this case, none of the connections of node *b* are modified. In three of the experiments carried out, the results were inferior to those obtained with the first form of the crossover operator (see Table 5.3).

Variation 4: In this variation, connections from neurons are transferred, whereas connections from terminals are not modified. Once more, the performance was inferior to that obtained with the crossover operator defined in this work (see Table 5.4).

These results suggest that artificial neural networks construct internal representations by building connections between neurons. Consequently, it is important to keep them as much as possible, by keeping them pointing to the same node. On the other hand, by moving connections from terminals, there is a chance that they are transferred to neurons by the crossover operator, helping to build new internal representations.

Nodes in the input layer are made unavailable for selection in the first parent. Firstly, because in the case of feedforward connectivity, the replacement of a terminal in the input layer would lead to an infeasible individual if a neuron were transferred to the input layer. Secondly, because it favors evolution toward less complex networks. This happens because terminals selected from the input layer of the second parent can replace neurons in the first parent, but not vice-versa. This feature is explored in the experiments reported in the next chapter.

The proposed crossover operator does not separate the index of the connected node from the corresponding weight. Consequently, it is irrelevant to define the bias as an ordinary weight or not.

Table 5.1: Summary of the results on binary classification problems, using the first variation of the crossover operator.

TASK	GEN (σ)	NEURONS				CONNECTIONS				EFFORT
		min	avg	max	σ	min	avg	max	σ	
XOR	2.7 (1.9)	6	9.6	10	0.8	15	24.3	31	2.4	1,800
3 parity	28.0 (32.6)	3	8.1	10	1.6	13	29.5	41	5.8	19,800
4 parity	205.9 (156.6)	4	7.8	10	1.4	23	40.4	59	7.9	163,200
5 parity	412.6 (129)	4	7.2	9	1.4	26	47.2	65	10.3	845,000

Table 5.2: Summary of the results on binary classification problems using the second variation of the crossover operator.

TASK	GEN (σ)	NEURONS				CONNECTIONS				EFFORT
		min	avg	max	σ	min	avg	max	σ	
XOR	2.2 (2.3)	4	9.4	10	1.1	12	23.7	28	2.8	2,400
3 parity	33.7 (69.2)	5	8.1	10	1.4	19	28.5	43	4.1	21,000
4 parity	232.4 (162.7)	3	8.1	10	1.6	18	40.0	53	7.9	219,600
5 parity	458.0 (97.6)	7	8.8	10	0.9	44	51.8	58	4.2	1,689,800

Table 5.3: Summary of the results on binary classification problems using the third variation of the crossover operator.

TASK	GEN (σ)	NEURONS				CONNECTIONS				EFFORT
		min	avg	max	σ	min	avg	max	σ	
XOR	2.4 (2.6)	8	9.5	10	0.7	19	24.1	28	2.0	2,400
3 parity	55.2 (107.8)	3	7.8	10	2.0	12	28.5	52	7.2	25,800
4 parity	244.5 (191.3)	4	7.2	10	1.7	22	37.6	52	7.9	168,000
5 parity	415.4 (150.5)	4	7.3	10	1.9	29	44.5	63	9.4	813,200

Table 5.4: Summary of the results on binary classification problems using the fourth variation of the crossover operator.

TASK	GEN (σ)	NEURONS				CONNECTIONS				EFFORT
		min	avg	max	σ	min	avg	max	σ	
XOR	2.4 (3.3)	1	9.1	10	2.0	5	23.2	29	4.7	2,000
3 parity	56.2 (121.8)	2	8.0	10	2.0	9	27.6	44	5.8	18,400
4 parity	309.5 (156.6)	4	7.7	10	1.7	21	38.3	55	8.0	336,000
5 parity	446.5 (110.8)	4	7.5	9	1.4	29	44.7	52	6.5	1,520,000

5.4 Mutation

The representation allows the implementation of the whole arsenal of mutation operators associated with evolutionary methods applied to artificial neural networks: addition and deletion of connections, crossover of an individual with a randomly generated one, crossover of a randomly selected node with a randomly generated one, addition of Gaussian noise to the weights and biases, etc.

The deletion or addition of nodes is not allowed, as the size of the genotype is constant. However, a neuron may be replaced with a terminal or vice-versa, and this may be used to reduce or increase the complexity of the network, within predefined limits.

Crossover and mutation alone can, in principle, generate solutions with varying degrees of complexity. However, unless a term is included in the fitness function to penalize the complexity of the network, there is no guarantee that parsimonious solutions will be achieved. Penalty terms require problem-dependent coefficients which are difficult to set. In this work, a different procedure was adopted, which does not involve any arbitrary parameter. After a 100% correct solution is found, one neuron is replaced with a terminal in each individual of the population, and the evolutionary process is resumed. This procedure is repeated every time a new solution is found, until the specified number of generations is achieved, or a sufficiently small architecture (according to a condition predefined by the user) is evolved. This strategy generates solutions of varying degrees of complexity, allowing the user to decide which solution is preferable: a complex one, possibly presenting fault tolerance, or a parsimonious one, with probably more generalization power.

Actually, this procedure may be viewed as a special form of mutation to be applied to the population as a whole at each generation with probability 1, if a new solution is found, or 0 otherwise.

Obviously, this procedure has a cost: the new population created by pruning should be reevaluated, otherwise the selection of parents for the next generation would be disturbed. However, as there is usually a gap of many generations between successive solutions, it

is also possible to carry out selection without reevaluation of the individuals. This may be interpreted as a perturbation in the evolutionary process. Both forms of pruning are explored in the experiments discussed in the next chapter.

5.5 Summary

On the grounds of the *lessons* learned in Chapter 4, several criteria for a good representation and good operators for the evolution of artificial neural networks were outlined. Based on these criteria, a new two-dimensional representation and a new crossover operator were proposed. The new approach is a direct encoding scheme, able to evolve the architecture and the weights simultaneously. The efficiency of the method proposed has been evaluated in previous work [151, 149, 150, 148], on well studied benchmark problems found in the literature. The results of these experiments are discussed in the next chapter.

Chapter 6

Experimental results with the two-dimensional representation

6.1 Introduction

Any method proposed, must work with typical parameters of genetic algorithms. Therefore, conventional procedures for selection, as well as crossover and mutation probabilities were used in the experiments. No optimization on the parameters of the genetic algorithm has been undertaken. In all experiments, the following procedure and parameter settings were used:

- Generational genetic algorithm;
- Tournament selection of parents (tournament size = 4);
- Mutation by crossover with a randomly created individual;
- Crossover and mutation probability of 70% and 5%, respectively;
- Maximum of 5 multiple connections per pair of nodes.

The maximum value for multiple connections was selected in order to operationally constrain the search, by keeping the value of the weights within a limited range, and preventing the bloat of the genotype by an excessive number of multiple connections.

All individuals in the initial population were initialized with random connectivity. However, it was assured that each node was directly or indirectly connected to the input and output layers, i.e. each node (including input nodes) was provided with a random connection to a higher order node (if available), and every neuron was provided with a random connection from a lower order node.

6.2 Binary classification problems

In these experiments a population of 200 individuals was evolved for a maximum of 500 generations. For each problem, 50 runs were performed with different random seeds. Unless otherwise stated, a single internal layer with 10 nodes was used. Initially, no terminals were present in the internal layer. The weights and biases were randomly initialized within the range [-1.0, +1.0]. The mean square error of the output of the network for all input patterns was used as fitness function. A threshold activation function was used, $f(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$. The exclusion of terminals of the internal layer in the initial population had the objective of demonstrating the ability of the method to reduce the complexity of the solutions.

To show the performance of the method proposed, it was applied to a test suite of standard benchmarks present in the literature: the odd-2 (XOR), odd-3, odd-4 and odd-5 parity problems (see Section 5.3), and the 4-symmetry and the *T-C* problems. In the 4-symmetry problem, the network is required to classify a 4 bits input bitstring as symmetric around its center or not [160, 110]. In the *T-C* task, the network is required to identify the characters *T* and *C* represented by a 3×3 bit template, placed in any position and orientation within a 4×4 matrix (see Figure 6.1). In this case, in order to compare results with other methods, a single internal layer with 16 nodes was used (all nodes initially occupied by neurons).

Figure 6.1: Templates for the *T*-*C* task. (a) Templates for the character *T*. (b) Templates for the character *C*.

The results are summarized in Tables 6.1 to 6.4. Column 2 represents the average number of generations (standard deviation in brackets). Columns 3, 4, 5 and 6 show the minimum, average, maximum and standard deviation for the number of neurons of the networks evolved, respectively. Columns 7, 8, 9 and 10 give the minimum, average, maximum and standard deviation for the number of connections of the networks evolved, respectively. Column 11 shows the computational effort (see Section 5.3).

Table 6.1 presents the results for the first solutions obtained in each run, i.e. the evolutionary process was interrupted as soon as a solution was found. Table 6.2 represents the results obtained when the evolution was carried out to the maximum number of generations specified, i.e. even if a solution was found, the search was not interrupted.

From the results in Table 6.1 one can realize that, with the exception of the XOR problem, the first networks evolved already present an average reduction in the number of hidden neurons, as compared to the number of hidden neurons at generation zero (remember that no terminals were presented as internal nodes initially). The XOR problem is an easy task, and the solutions were found so fast that there was not enough time for the evolutionary process to reduce the number of neurons. The ability of the method to evolve parsimonious solutions is more evident in Table 6.2, where a substantial reduction in the number of neurons was achieved in all tasks. This confirms that the proposed crossover operator favors solutions of lower complexity as argued in Section 5.3.

It must be emphasized that all networks in the initial population were much bigger than the smallest solutions obtained. In the case of the XOR, 3-parity, 4-parity and 5-parity, they were 10, 5, 5 and 3 times bigger than the smallest solutions achieved, respectively. Obviously, to find minimal solutions one should initialize the population with individuals of varying degrees of complexity, by using a random mix of terminals and neurons in the internal layers. This approach is explored for a different task in Section 6.3.

The results for the parity problems compare very favorably with those reported in the evolutionary computation literature in terms of generations to find a solution. For example, to solve the XOR problem the following numbers of generations to attain solutions have been

Table 6.1: Summary of the results on the binary classification problems using a single internal layer, obtained by stopping the evolutionary process when a first solution was found.

TASK	GEN (σ)	NEURONS				CONNECTIONS				EFFORT
		min	avg	max	σ	min	avg	max	σ	
XOR	2.7 (1.9)	6	9.6	10	0.8	15	24.3	31	2.4	1,800
3 parity	28.0 (32.6)	3	8.1	10	1.6	13	29.5	41	5.8	19,800
4 parity	205.9 (156.6)	4	7.8	10	1.4	23	40.4	59	7.9	163,200
5 parity	412.6 (129)	4	7.2	9	1.4	26	47.2	65	10.3	845,000
Symmetry	124.9 (152.4)	4	7.2	10	1.7	20	32.7	49	6.6	75,000
T-C	184.7 (170.2)	6	10.1	13	2.0	58	94.4	139	20.8	146,000

Table 6.2: Summary of the results on the binary classification problems using a single internal layer, obtained by carrying out the evolution to the maximum number of generations specified.

TASK	GEN (σ)	NEURONS				CONNECTIONS				EFFORT
		min	avg	max	σ	min	avg	max	σ	
XOR	280.2 (132.2)	1	2.6	5	1.0	5	9.9	17	3.0	98,800
3 parity	299.9 (121.7)	2	3.5	7	1.0	10	16.5	28	4.1	99,200
4 parity	373.5 (116.1)	2	5.8	9	1.6	14	31.0	54	8.3	292,200
5 parity	456.6 (78.6)	3	6.3	9	1.5	21	40.2	61	9.1	986,000
Symmetry	295.0 (152.1)	3	4.6	8	1.3	13	22.1	34	4.5	268,200
T-C	401.1 (120.6)	4	7.5	12	1.8	37	69.3	100	16.1	288,600

reported: 50 [127], 100 [162], 22 [166]. For the odd-4 parity, Zhang and Mühlenbein [202] report a solution achieved in 9 generations, but a population of 1,000 individuals was used, and the authors used training by a hillclimbing procedure at each generation. The resulting minimal network had 6 neurons in the hidden layer and 23 connections, to be contrasted to the solution with 4 hidden neurons and 23 connections obtained with the two-dimensional approach.

The scalability of the method can be assessed by comparing the computational effort for the odd-3, 4 and 5 parity problems to those reported using other methods. For example, Koza [95] using GP with a specialized function set including Boolean functions, reports an effort of 80,000 for even 3-parity, 912,000 for odd-4 parity, and 7,840,000 for even-5 parity (the odd and even parity are simply complements to each other).

To illustrate the conversion of the two-dimensional representation into a network, typical solutions obtained for the XOR and the odd-3 parity problem using a single internal layer are shown in Figures 6.2 and 6.3, respectively. Note how in both cases, most of the neurons in the internal layer (which was initialized with neurons exclusively) were replaced with terminals during the evolutionary process, to confirm the bias provided by the crossover operator.

For the *T-C* task, the two-dimensional method produced a solution with a 16-4-1 topology and 34 connections. In comparison, Braun and Zagorski [30] report a 11-1-1 architecture with 22 connections, and McDonnell and Waagen *et al.* [114] report a 13-6-1 topology with 34 connections. The fact that the two-dimensional method was unable to reduce the complexity of the networks any further might be a consequence of the high proportion of terminals in the internal layer of the individuals of the population after some point in the evolution. A new battery of 50 runs was then carried out, using the pruning procedure introduced in Section 5.4 to search for an even more parsimonious solution. The pruning procedure without reevaluation of the population yielded a solution with a 13-1-1 topology and 23 connections. The pruning procedure with reevaluation produced a 11-1-1 topology with 14 connections (see Figure 6.10). This means that, in the minimal solution, 96% of the

Figure 6.2: Typical solution for the XOR problem. For visualization purpose, the node in the output layer has been centered. Values in the circles are biases. (a) Two-dimensional representation with 10 internal nodes in a single layer. (b) Corresponding network, after grouping connections from the same terminals by adding up their weights. One neuron with no outgoing connections was also eliminated.

408 possible connections within a feedforward architecture of 16-16-1 and 5 unnecessary input neurons were not used. This testifies the efficiency of the pruning strategy with reevaluation. This is better visualized by comparing the evolution of the average number of hidden neurons in the population, with and without pruning, for the run which produced the minimal solution.

As can be realized from Figure 6.4, without pruning the average number of hidden neurons was reduced to about 10 neurons. However, there was not enough selective pressure to force a further reduction. The result for the same run using pruning with reevaluation presented in Figure 6.5 shows distinct instants in the evolutionary process, at generations 111 and 240, where solutions were found, leading to the pruning of a neuron in each member of the population, and forcing the reduction in the average size of the encoded networks.

Figure 6.3: Typical solution for the odd-3 problem. (a) Two-dimensional representation with 10 internal nodes in a single layer. (b) Corresponding network, after grouping connections from the same terminals by adding up their weights.

From generation 268 to 492, a cascade of additional solutions resulted in a substantial reduction on the average number of hidden neurons down to about 1 neuron. The evolution using pruning without reevaluation is shown in Figure 6.6. In this case, the procedure was not so effective. When a first solution was found at generation 111, a neuron was also deleted from every member of the population, but the individuals were not reevaluated. As a consequence, it took longer for the evolutionary process to find a new solution at generation 293. Afterwards, a sequence of solutions from generation 305 to 426 were able to reduce the average number of hidden neurons to about 2.5 neurons (although in the end this number increased again to about 6 neurons). The efficiency of the pruning strategies can also be visualized by inspecting Figures 6.7, 6.8 and 6.9, which show the evolution of the average number of hidden neurons for the best individual in the population over 50 runs.

The generalization power of the minimal solution evolved for the *T-C* problem was tested. This was performed by inverting one bit of the 16 bits of each character, and presenting them to the network. The network was still able to identify the noisy templates in 82% of the 512 cases. One might be led to the conclusion that the minimal solution shown in Figure 6.10 is telling one character from the other by simply counting the number of active bits, 5 for the *T* character, and 7 for the *C* character. This is not the case, since in some cases the number of active bits of both templates are the same, as the corresponding terminals in the input layer of the network are disconnected. In Figure 6.11a, the unused bits are indicated in the 4×4 matrix which contains the templates for both characters. In Figures 6.11b and c, two different pairs of templates are shown, where both characters have the same number of active bits. This suggests that the network makes the distinction not only by the number of active bits, but also by their position in the matrix.

Does the number of internal layers have any effect on the efficiency of the two-dimensional method? As was pointed out in Section 2.3, feedforward artificial neural networks with a single hidden layer are universal approximators. But this does not mean that more hidden layers will not do a better job. To answer this question, 50 additional runs were carried out in the same tasks, with the same number of internal nodes used in the previous experiments

Figure 6.4: Evolution of the average number of hidden neurons of the individuals of the population without pruning.

Figure 6.5: Evolution of the average number of hidden neurons of the individuals of the population using pruning with reevaluation.

Figure 6.6: Evolution of the average number of hidden neurons of the individuals of the population using pruning without reevaluation.

Figure 6.7: Evolution of the average number of hidden neurons of the best individual in the population without pruning. Average over 50 runs.

Figure 6.8: Evolution of the average number of hidden neurons of the best individual in the population using pruning with reevaluation. Average over 50 runs.

Figure 6.9: Evolution of the average number of hidden neurons of the best individual in the population using pruning without reevaluation. Average over 50 runs.

Figure 6.10: Minimal solution for the T-C task. (a) Two-dimensional representation. (b) Corresponding network, after grouping connections from the same terminals by adding up their weights.

Figure 6.11: (a) The unused bits in the input to the network in Figure 6.10b are indicated by numbers in the 4×4 matrix. (b) *T* and *C* templates with the same number of active bits. (c) Another example of templates with the same number of active bits.

now equally divided into two internal layers. The results shown in Tables 6.3 and 6.4 indicate a slightly better performance with a single layer in these problems. It seems that, for these tasks, the use of more than one internal layer to build more than one level of internal representation in the networks, is actually counter-productive for the search. In other tasks the situation might be different. In any event, the difference is not substantial, indicating that it is possible to use more than one internal layer if necessary or desired (this subject is further investigated with another set of experiments discussed in Section 6.3). A typical solution for the symmetry problem using two internal layers is shown in Figure 6.12.

Table 6.3: Summary of the results on the binary classification problems using two internal layers, obtained by stopping the evolutionary process when a first solution was found.

TASK	GEN (σ)	NEURONS				CONNECTIONS				EFFORT
		min	avg	max	σ	min	avg	max	σ	
XOR	2.6 (2.0)	7	9.6	10	0.7	23	30.5	37	3.6	1,800
3 parity	29.8 (35.7)	3	8.2	10	1.4	16	34.2	46	6.4	22,000
4 parity	233.9 (158.6)	4	7.8	10	1.6	22	44.5	62	9.6	196,800
5 parity	436.1 (123.3)	7	8.5	10	0.9	39	52.7	63	7.4	1,386,000
Symmetry	185.1 (170.3)	3	7.1	10	1.7	15	36.0	50	9.9	136,000
T-C	201.0 (174.9)	5	9.8	14	1.8	43	98.0	137	21.8	270,400

Table 6.4: Summary of the results on the binary classification problems using two internal layers, obtained by carrying out the evolution to the maximum number of generations specified.

TASK	GEN (σ)	NEURONS				CONNECTIONS				EFFORT
		min	avg	max	σ	min	avg	max	σ	
XOR	289.6 (125.1)	1	2.9	5	2.7	5	10.9	17	2.7	100,200
3 parity	309.5 (138.5)	2	4.0	6	1.2	10	18.5	32	4.7	99,600
4 parity	344.1 (131.4)	4	6.5	9	1.6	21	36.5	57	8.6	298,800
5 parity	458.2 (84.5)	4	7.2	10	1.7	31	45.0	63	9.7	1,500,000
Symmetry	347.4 (138.9)	2	4.7	8	1.4	12	23.5	37	6.0	298,200
T-C	377,2 (131.6)	4	7.9	11	1.8	34	76.5	118	20.3	293,400

Figure 6.12: Minimal solution for the symmetry problem. (a) Two-dimensional representation with 10 internal nodes distributed over two internal layers with 5 nodes each. (b) Corresponding network, after grouping connections from the same terminals by adding up their weights.

6.3 Pole-balancing problem

This problem is a well-studied benchmark for evolving neural controllers [193, 194, 71, 3, 198, 124, 21, 35, 163, 33]. The task consists of balancing a pole hinged in the center of a moving cart, by applying a force to the cart exclusively. The pole is only allowed to move in a vertical plane and the cart moves in a one-dimensional track (see Figure 6.13).

The only forces acting on the system are a control force applied to the cart and gravity. The mass of the pole is uniformly distributed along its length. The other dimensions of the pole are negligible compared to its length. Under these assumptions, the equations of motion and the usual parameter setting for the coupled cart-pole system are:

$$\ddot{\theta} = \frac{m_s g \sin\theta - \cos\theta[f + m_p L \dot{\theta}^2 \sin\theta]}{(4/3)m_s L - m_p L \cos^2\theta},$$

$$\ddot{x} = \frac{f + m_p L[\dot{\theta}^2 \sin\theta - \ddot{\theta}\cos\theta]}{m_s},$$

where

x is the cart position as measured from the center of the track (m),

$\dot{x}$ is the cart velocity (m/s),

$\ddot{x}$ is the cart acceleration (m/s^2),

θ is the inclination of the pole to the vertical (radians),

$\dot{\theta}$ is the angular velocity of the pole (radians/sec),

$\ddot{\theta}$ is the angular acceleration of the pole (radians/sec^2),

L is half the length of the pole = 0.5 m,

m_p is the mass of the pole = 0.1 kg,

m_s is the mass of the cart-pole system = 1.1 kg,

f is the control force applied to the cart (N),

g is the gravity acceleration = 9.8 m/sec^2

Figure 6.13: Cart-pole system.

The controller must provide a sequence of left and right pushes to the cart, in order to keep the system within specified limits. Different strategies must be compared exclusively on their net result: failure or success. There is no way of knowing in advance which individual action contributes to a successful or unsuccessful control strategy in the end.

The state of the cart-pole system is defined by four variables representing the position and velocity of the cart, and the angle of the pole to the vertical and its angular velocity. The equations of motion were numerically integrated to give the new state of the system at each time step (Euler's method with a time step of 0.02s was used). A failure signal was triggered when the angle of the pole or the position of the cart exceeded specified limits. The limit for the cart position was ± 2.4 m, and the limit for the angle of the pole varied according to the experiment (details are given further in the text).

At each generation, the fitness of the individuals in the population was evaluated as the number of time steps in which they were able to keep the pole balanced and the cart in the track. At each time step, the network received as input the four state variables of the cart-pole system, and returned as output the force to be applied to the cart. Before being introduced as inputs to the network, the state variables were normalized to be in the range ± 1.0 using the following normalization factors [194]: position of the cart (2.4m), speed of the cart (1.5m/s), angle of the pole (specified limit) and angular speed of the pole (115 degrees/s). The speed of the pole and of the cart were not limited during simulation, but they were still normalized by these factors.

In these experiments a population of 100 individuals was evolved for a maximum of 100 generations, with the evolution being interrupted as soon as a solution was found. All individuals were initialized with 10 internal nodes. Experiments with one and two internal layers were carried out. The weights and biases were randomly initialized within the range [-1.0, +1.0], and the hyperbolic tangent was used as activation function.

In order to compare the performance of the two-dimensional representation to indirect and direct encoding methods, two sets of experiments were carried out using different approaches.

6.3.1 First set of experiments

In [194, 71], Gruau *et al.* used cellular encoding to evolve a neural controller for the cart-pole system using two different control actions: a *bang-bang* and a *continuous* control force. In the first case, the force f can only take two discrete values: +10 or -10. In the second case, it can take any value in the range [-10.0,+10.0]. The fitness of an individual is given by the number of time steps in which the system is controlled starting from different initial states (see Table 6.5). A solution is a network which can control the system over all initial states for 1,000 time steps. The limit for the angle of the pole was set at 36 degrees.

Tables 6.6 and 6.7 show a summary of the results of 50 independent runs for both control actions. Column 1 indicates the shape of the grid used (control action in brackets). Column 2 represents the average number of generations (standard deviation in brackets). Columns 3, 4, 5 and 6 show the minimum, average, maximum and standard deviation for the number

Table 6.5: Combinations of initial states for the cart and pole adapted from [194].

Cart position	0	+2.14	-2.14	+2.4	-2.4	0	0	0	0	+2.4	-2.4
Cart velocity	0	0	0	0	0	0	0	0	0	0	0
Pole position	0	0	0	-27	+27	+13.5	-13.5	+27	-27	0	0
Pole velocity	0	0	0	0	0	+32.5	-32.5	0	0	-65	+65

of hidden neurons of the networks evolved, respectively. Columns 7, 8, 9 and 10 give the minimum, average, maximum and standard deviation for the number of connections of the networks evolved, respectively. Column 11 shows the computational effort.

In the experiments with the binary classification problems, the individuals of the initial population had no terminals in the internal layers. On the pole-balancing problem, the effect of the mix of neurons and terminals in the layers was investigated, by varying the proportion of terminals and neurons in the internal layers of the individuals of the initial population.

The results shown in Tables 6.6 were achieved using a random assignment of terminals and neurons in the internal layers of the individuals in the initial population. This implies a random distribution of networks of different sizes in the initial population. The grid with a single layer performed slightly better in the continuous case, while the grid with two internal layers fared better in the bang-bang case.

As can be realized from the average number of hidden neurons of the networks evolved, it seems to be wiser to initialize the population with individuals representing encoded networks with fewer hidden neurons, by using a higher proportion of terminals in the internal layers of the individuals in the initial population. To check this, experiments were carried out with an average of 75% of the internal nodes assigned to terminals. The results of these runs are shown in Table 6.7.

As expected, by using smaller networks, the computational effort to evolve a solution was reduced. The consequence of these results is that, if by any specific *a priori* information about the problem being solved, small solutions can be expected (small in comparison to the size of the grid used), it is possible to speed up the search by introducing more terminals in the internal layers of the grid. Consequently, a large grid can be used to constrain the maximum size of the architectures evolved, while still starting the search with small networks.

In order to compare the results obtained with the two-dimensional representation to those reported in [194, 71], a generalization test was performed for all solutions found, by counting the number of successful control runs of the cart-pole system for 1,000 time steps. Each solution was required to control the system starting from 625 different initial states, defined by each normalized state variable taking the values: ± 0.9, ± 0.5 and 0.

The results are summarized in Tables 6.8 and 6.9. The second column represents the number of fitness evaluations to achieve a solution during the learning process (standard deviation in brackets). Columns 3, 4 and 5 show the best, mean, and worst number of successful control runs, respectively. Column 6 represents the standard deviation of the generalization test.

In both cases the two-dimensional method considerably outperformed cellular encoding in terms of number of fitness evaluations to achieve a solution. The results are even more impressive if one considers the size of the populations used in the cellular encoding experiments: 2,048 in [194] and 4,096 in [71]. In [194], the number of individuals in the population is almost of the same size as the number of fitness evaluations to achieve a solution for the bang-bang case, indicating that a good approximation was already present in the initial population. In [71], the size of the population is considerably greater than the number of evaluations to achieve a solution in the continuous case, which means that the solution was already present in the initial population. It may be also pointed out that with both control actions, the number of fitness evaluations to achieve a solution with the two-dimensional representation were of the same order of magnitude. This suggests that the proposed method scales up better than cellular encoding with the difficulty of the problem.

The generalization power of the solutions obtained is superior to that reported in [71], but inferior to those reported in [194]. This may be a consequence of the different activation functions used (in [194] the authors used a clipped linear activation function between -1 and 1). However, a more extensive investigation should be carried out to clarify the situation.

Table 6.6: Summary of the results on the pole-balancing problem using an initial random proportion of terminals and neurons in the internal layers, obtained by interrupting the evolutionary process when a first solution was found.

TOPOLOGY	GEN (σ)	NEURONS				CONNECTIONS				EFFORT
		min	avg	max	σ	min	avg	max	σ	
4-10-1 (bang-bang)	7.2 (7.3)	1	4.3	8	1.7	7	17.6	30	5.4	2,600
4-5-5-1 (bang-bang)	5.5 (3.2)	1	3.8	8	1.8	6	17.3	33	6.9	1,600
4-10-1 (continuous)	9.4 (8.1)	0	3.8	8	1.8	3	16.8	36	7.4	3,600
4-5-5-1 (continuous)	14.3 (21.9)	0	4.4	7	1.7	4	19.5	38	6.8	3,800

Table 6.7: Summary of the results to obtain a first solution to the pole-balancing problem using an initial average fraction of 75% of terminals in the internal layers, obtained by interrupting the evolutionary process when a first solution was found.

TOPOLOGY	GEN (σ)	NEURONS				CONNECTIONS				EFFORT
		min	avg	max	σ	min	avg	max	σ	
4-10-1 (bang-bang)	3.7 (2.2)	0	1.0	5	1.1	4	8.1	26	4.4	1,100
4-5-5-1 (bang-bang)	4.0 (2.4)	0	1.3	4	1.1	4	9.0	22	4.9	1,300
4-10-1 (continuous)	7.0 (5.4)	0	1.0	3	0.9	3	7.7	16	3.9	2,600
4-5-5-1 (continuous)	8.2 (6.2)	0	1.4	6	1.3	4	9.8	34	6.0	2,700

Table 6.8: Comparison of results obtained with the two-dimensional representation and cellular encoding using a bang-bang control action. Individuals in the population were initialized with a random proportion of terminals and neurons in the internal layers.

Results of the learning process		Results of the generalization test			
Method	Number of fitness evaluations (σ)	Number of successful runs			
		Best	Mean	Worst	σ
Cellular encoding [194]	2,234 (N/A)	N/A	430	N/A	N/A
Two-dimensional (4-10-1)	612.4 (524.4)	372	329.5	195	34.4
Two-dimensional (4-5-5-1)	485.0 (226.3)	376	318.5	216	35.3

Table 6.9: Comparison of results obtained with the two-dimensional representation and cellular encoding using a continuous control action. Individuals in the population were initialized with a random proportion of terminals and neurons in the internal layers.

Results of the learning process		Results of the generalization test			
Method	Number of fitness evaluations (σ)	Number of successful runs			
		Best	Mean	Worst	σ
Cellular encoding [194]	19,011 (N/A)	N/A	386	N/A	N/A
Cellular encoding [71]	1,400 (N/A)	N/A	250	N/A	N/A
Two-dimensional (4-10-1)	775.9 (581.1)	361	301.5	217	32.8
Two-dimensional (4-5-5-1)	1099.6 (1530.5)	372	303.6	197	39.8

6.3.2 Second set of experiments

Whitley *et al*. [193] investigated the training of a neural controller for balancing the cart-pole system using Genitor, a GA-based software package, and also using a reinforcement learning technique called adaptive heuristic critic (AHC) [21].

In these experiments, a bang-bang control action based on a probabilistic interpretation of the output of the network was used, in which an output of 0.75 does not necessarily mean a push to the right, but rather that this action has a probability of occurrence of 75%.

The cart is initially in the center of the track with the pole vertically aligned, both with null velocity. An individual is considered to be a solution to the problem if it can keep the system within the predefined limits for 120,000 time steps. The experiments were carried out with a failure signal at two different limits for the angle of the pole: 12 and 35 degrees. The results are shown in Table 6.10 (the columns of the table have the same meaning as is in Table 6.6).

To verify the generalization power of the solutions found in the second set of experiments, each of them was required to balance the system starting from a set of 625 normalized random states for 1,000 time steps. The results of these experiments are summarized in Tables 6.11 and 6.12 (the columns of these tables have the same meaning as in Table 6.8)).

The results were substantially better than those reported in [193] in terms of number of fitness evaluations to achieve a solution, for both methods: Genitor and AHC. The results are remarkable if one considers that Whitley and collaborators did not evolve the architecture, only the weights. The generalization power of the solutions attained was also better.

Moriarty and Miikkulainen [124] also investigated the evolution of neural controllers for the pole-balancing problem, by using SANE (a GA-based method for building artificial neural networks) to evolve the connectivity and the weights of a neural controller based on a topology with a fixed hidden layer of 8 hidden neurons. The authors also trained a network by two different reinforcement learning methods: AHC and Q-learning [189].

Table 6.10: Summary of results obtained in the second set of experiments. In the first column, the specified maximum angle of the pole is indicated in brackets. Values represent average over 50 random runs.

TOPOLOGY	GEN (σ)	NEURONS				CONNECTIONS				EFFORT
		min	avg	max	σ	min	avg	max	σ	
4-10-1 (12 ^{o})	8.7 (6.0)	1	3.7	7	1.5	6	15.4	26	5.5	3,400
4-5-5-1 (12 ^{o})	9.6 (7.7)	1	4.3	8	1.8	6	19.2	37	6.9	3,000
4-10-1 (35 ^{o})	5.3 (3.5)	1	4.3	8	1.6	1	17.9	8	5.7	1,600
4-5-5-1 (35 ^{o})	6.1 (5.6)	0	3.9	8	1.9	4	17.3	35	7.1	2,400

Table 6.11: Results obtained with the two-dimensional representation and those reported by Whitley *et al.* [193] for the 12^o case.

Results of the learning process		Results of the generalization test			
Method	Number of fitness evaluations (σ)	Number of successful runs			
		Best	Mean	Worst	σ
Genitor	4,097 (2,205)	446	297	24	89
AHC	5,433 (2,390)	372	192	70	79
Two-dimensional (4-10-1)	720.5 (429.1)	498	380.6	101	87.3
Two-dimensional (4-5-5-1)	789.0 (554.1)	509	354.2	169	80.5

Table 6.12: Results obtained with the two-dimensional representation and those reported by Whitley *et al.* [193] for the 35^o case.

Results of the learning process		Results of the generalization test			
Method	Number of fitness evaluations (σ)	Number of successful runs			
		Best	Mean	Worst	σ
Genitor	4,206 (1,777)	430	304	92	92
AHC	3,922 (2,452)	406	271	19	108
Two-dimensional (4-10-1)	482.2 (246.8)	541	422.0	170	72.6
Two-dimensional (4-5-5-1)	538.5 (400.9)	535	426.3	219	87.3

Table 6.13: Results obtained with the two-dimensional representation and those reported by Moriarty and Miikkulainen [124] for the 12^o case.

Results of the learning process		Results of the generalization test			
Method	Number of fitness evaluations (σ)	Number of successful runs			
		Best	Mean	Worst	σ
SANE	535 (329)	81	48	1	25
AHC	8,976 (7,573)	76	44	5	20
Q-learning	1,975 (1,919)	61	41	13	11
Two-dimensional (4-10-1)	546.1 (308.0)	88	68.4	31	13.9
Two-dimensional (4-5-5-1)	488.5 (215.0)	90	65.9	36	14.9

As Moriarty and Miikkulainen used 60 degrees/s as normalization factor for the angular speed of the pole, new experiments were carried out with this parameter setting. In addition, the generalization test was performed with 100 random states. The results are summarized in Table 6.13. The columns of Table 6.13 have basically the same meaning as in Table 6.11, since for the AHC and the Q-learning algorithms, at each iteration of the learning process, a balance attempt, i.e. a fitness evaluation, is carried out.

In terms of fitness evaluations, as well as generalization power, the results obtained with the two-dimensional representation were considerably superior to those achieved by Moriarty and Miikkulainen with AHC and Q-learning. In terms of fitness evaluations only, the performance of SANE was slightly superior to the two-dimensional representation using the 4-10-1 grid, but inferior to that attained using the 4-5-5-1 grid. Also, the generalization power of the solutions evolved by the two-dimensional representation with both grids, was considerably superior to that obtained by Moriarity and Mikkulainen with SANE.

6.3.3 Tracker problem

The experiments in Sections 6.2 and 6.3 involved feedforward architectures. To assess the ability of the two-dimensional representation in evolving recurrent networks, it was applied to the moderately complex problem of evolving a control system for an agent whose objective is to track and clear a trail. The particular trail used, the John Muir trail [87, 6], consists of 89 tiles in a 32x32 grid (see Figure 6.14). The grid is considered to be toroidal.

The tracker starts in the upper left corner, and faces the first position of the trail. The only information available to the tracker is whether the position ahead belongs to the trail or not. Based on this information, at each time step, the tracker can take 4 possible actions: *wait* (doing nothing), *move forward* (one position), *turn right* 90^o (without moving) or *turn left* 90^o (without moving). When the tracker moves to a position of the trail, that position is immediately cleared. This is a variant of the well known ”ant” problem often studied in the GP literature [94, 95, 97].

Usually, the information to the tracker is given as a pair of input data [87, 6]: the pair is (1,0) if the position ahead of the current tracker position belongs to the trail, and (0,1) if it does not. The objective is to build a network that, at each time step, receives this information, returns the action to be carried out, and clears the maximum number of positions in a specified number of time steps (200 in the experiments carried out in this work). As information about where the tracker is on the trail is not available, it is clear that to solve the problem the network must have some sort of memory in order to remember its position. As a consequence, a recurrent network is necessary. Although it might seem to be unnecessary, the *wait* action allows the network to update its internal state while staying at the same position (this can be imagined as ”thinking” about what to do next).

This problem is very hard to solve in 200 time steps (in [95], John Koza allotted 400 time steps to follow slightly different trails using GP). However, it is relatively easy to find solutions able to clear up to 90% of the trail (in [87] Jefferson *et al*. present a statistical evaluation of the difficulty of the problem using finite automata and artificial neural networks). This suggests that the search space has many local minima to mislead evolution,

Figure 6.14: John Muir trail. Positions of the trail are numbered.

as confirmed by Langdon and Poli [100], who studied the fitness landscape of the ant problem for GP. This makes the problem a good benchmark to test the ability of the proposed method to evolve recurrent networks.

Asymmetric recurrent networks were used to solve the problem, where a neuron can receive connections from any other neuron (including output neurons). All neurons are evaluated synchronously as a function of the output of the neurons in the previous time step, and the current input to the network. Initially, all neurons have null output.

In all experiments, a population of 200 individuals was evolved. Terminals and neurons were randomly allocated in a 2-10-4 grid. The weights and biases were randomly initialized within the range [-1.0, +1.0]. The fitness of an individual was measured by the number of trail positions cleared in 200 time steps. The population was evolved for a maximum of 500 generations, and a threshold function was used as activation function, $f(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$. Table 6.14 shows the mapping of the network output into actions.

In 20 independent runs, the average number of positions cleared by the best individuals evolved was 84.3, with a standard deviation of 3.0. By assigning additional time steps to the best individuals evolved, in 50% of the runs, the best individuals cleared the 89 positions in less than 240 time steps. Two solutions were found which were able to clear the entire trail in 199 time steps: one with 6 hidden neurons and 70 connections, and another with 4 hidden neurons and 54 connections (see Figures 6.15 and Figures 6.16), leading to a computational effort of 4,373,600. An unexpected feature of the solutions found is that, although available, they do not make use of either the *turn left* or the *wait* options to move along the trail. Although both solutions clear the trail in the same number of steps and use the same options to traverse the trail, they implement different strategies to do that. This can be verified by comparing the sequence of moves of both trackers shown in Figures 6.17, 6.18, 6.19 and 6.20, as well as the evolution of the number of track positions cleared as a function of the number of time steps shown in Figures 6.21 and 6.22.

It is worth mentioning that additional solutions were obtained by allowing these successful runs to continue to the maximum number of generations specified. Actually, 7 different additional solutions were generated. However, these multiple solutions were not included in the statistics presented in the previous paragraph.

Table 6.14: Mapping of the network output into the 4 possible actions. For convenience, only two of the output neurons were used to define the actions. However, the other two neurons in the third layer are effectively used to compute the internal state of the network.

NETWORK OUTPUT	ACTION
OUT3 ¡ 0 and OUT4 ¡ 0	MOVE FORWARD
OUT3 ¿ 0 and OUT4 ¡ 0	TURN LEFT
OUT3 ¡ 0 and OUT4 ¿ 0	TURN RIGHT
OUT3 ¿ 0 and OUT4 ¿ 0	WAIT

Figure 6.15: Full solution to the tracker problem with 6 hidden neurons. Values in the circles are biases.

Figure 6.16: Full solution to the tracker problem with 4 hidden neurons. Values in the circles are biases.

Figure 6.17: Actions taken by the tracker as a function of the number of steps for the solution with 6 hidden neurons.

Figure 6.18: Actions taken by the tracker as a function of the number of steps for the solution with 4 hidden neurons.

Figure 6.19: Tracker with 6 hidden neurons. Position along the trail at each time step. Steps missing are spent executing the *turn right* action.

Figure 6.20: Tracker with 4 hidden neurons. Position along the trail at each time step. Steps missing are spent executing the *turn right* action.

Figure 6.21: Number of track positions cleared as a function of the number of steps for the solution with 6 hidden neurons.

Figure 6.22: Number of track positions cleared as a function of the number of steps for the solution with 4 hidden neurons.

These are very promising results. For comparison, Jefferson *et al.* [87] report a solution with 5 hidden neurons, which clears the trail in 200 time steps. The solution is a hand-crafted architecture trained by genetic algorithms using a huge population (65,536 individuals). Using an evolutionary programming approach, Angeline *et al.* [6] report a network with 9 hidden neurons, evolved in 2,090 generations (population of 100 individuals). The network clears 81 positions in 200 time steps and takes additional 119 time steps to clear the entire trail. The authors also report another network evolved in 1,595 generations, which scores 82 positions in 200 time steps.

6.4 Summary

The two-dimensional representation was applied to the evolution of feedforward and recurrent architectures in a test suite of benchmark problems. It was demonstrated that, in the tasks analyzed, it compares very favorably to the direct or indirect approaches used for comparison. In the next chapter, a variation of the method is discussed, which uses the new crossover operator to evolve the architecture, while genetic programming is used to evolve the weights.

Chapter 7

Evolution of artificial neural networks using weight mapping

7.1 Introduction

The training of artificial neural networks for a particular task can be seen as a mapping of the initial set of random weights into a new set of adapted weights which solves the problem. That means, the process of training defines a function to map the initial set of random weights into the correct ones. For example, the backpropagation training algorithm is an attempt to construct such a *mapping function* iteratively. Unfortunately, learning procedures always use the same strategy to adapt the weights, although the error surfaces associated with different tasks present completely different features. As a consequence, all sorts of difficulties arise in this process (see discussion in Section 2.2). Genetic programming has been used to evolve new learning rules [24, 126, 154], which overcome some of these difficulties. These approaches seek to evolve new general purpose training procedures, which are more robust and more efficient than existing ones. However, it would be more interesting to address each task by using a different learning method to adapt the weights.

In this chapter, a new approach is discussed, which combines genetic programming with a

variation of the two-dimensional representation introduced in Chapter 5. The new method uses GP to automatically build a non-iterative mapping function to adapt the weights. The function is evolved concurrently to the architecture, and is tailored to the task at hand.

7.2 Representation

The general structure of the new approach can be described as follows: each individual has its own mapping function to be used to compute the adapted weights from *raw* weights (biases are treated as ordinary weights). The process is illustrated in Figure 7.1. The individual in Figure 7.1a has a set of raw weights which are not suitable to solve a particular problem. New values are then computed by applying the mapping function in Figure 7.1b to the raw weights, resulting in the individual in Figure 7.1c. The mapping function is actually implemented as the parse tree shown in Figure 7.1d. The structure of the network in Figure 7.1a is encoded and evolved similarly to the two-dimensional representation presented in Chapter 5, whereas the parse tree is evolved by genetic programming.

All individuals in the population are structures which have one part to describe the architecture of the encoded network, and a second part to represent the function to map the raw random weights into the values used to evaluate the network performance. The representation is illustrated in Figure 7.2. The first part is encoded similarly to the two-dimensional representation described in Chapter 5, as an ordered list of nodes. Nodes may be of two types: *terminal* or *neuron*. In the first case, the node is a variable containing an input to the network. In the second case, the node represents a processing element of the encoded network. When the node is a neuron, it is represented as a list containing information about its connections. Connections are represented by indexes, indicating the positions of the connected nodes. A grid is also used to interpret the architecture part of the genotype as a two-dimensional structure. The second part of the genotype encodes the mapping function as a parse tree.

Figure 7.1: Conversion of raw weights into adapted ones. (a) Two-dimensional representation with raw weights (values in the circles are also weights representing biases). (b) Mapping function. The variable *W* represents a weight (c) Two-dimensional representation with adapted weights. (d) Parse tree encoding the mapping function.

Figure 7.2: Genotype divided into two parts. (a) First part represents the architecture. (b) Second part is a parse tree to encode the mapping function. *R1* and *R2* are random constants.

Note that neither weights nor biases are included in the genotype anymore. To assign these values to the architecture, a single ordered list of raw weights is created at the beginning of the evolutionary process, as if all nodes (including those occasionally occupied by terminals) were interconnected with the maximum allowed connectivity (feedforward or recurrent). The list also includes raw weights representing biases for all nodes (even for those nodes occasionally occupied by terminals). The set of raw weights is fixed and unique, and this is a characteristic of the population. It is used to assign the same raw weights and biases to the connections and neurons of all individuals of the population during fitness evaluation. For each neuron of an individual, this is performed by reading the indexes of the connected nodes, and picking up the corresponding connection weights and biases from the list of raw weights. Afterwards, the mapping function is applied to these raw values, to compute adapted ones.

As in genetic programming, to build the parse tree in the second part of the genotype, a set of *terminals* (not to be confused with the set of terminals used to define the architecture in the first part of the genotype) and a set of *functions* is defined. The set of terminals includes a variable representing the weight which the function is applied to, and also a variable which is used to initialize random constants when the individuals of the initial population are created (when a parse tree is initially created, the terminals which contain this variable are randomly initialized with real-valued constants). The set of functions may include simple arithmetic operations or any other suitable mathematical function.

In the two-dimensional representation described in Chapter 5, multiple connections between the same two nodes were allowed in order to fine tune the weights. In the current representation, the full adaptation of the weights is carried out by the mapping function. Consequently, multiple connections are not allowed.

It is interesting to note that for a fixed architecture, if the values of the correct adapted weights were known in advance, finding the mapping function (as an academic exercise only, since the target weights would be already known) would be a problem of fitting a curve to a set of points (see Figure 7.3a). However, as these values are not known, the mapping function has to be evolved based on the fitness of the individual, and then used to compute the adapted weights (see Figure 7.3b). This approach is similar to training by genetic algorithms (see Section 4.2). Training by GAs collects information about fitness from different points in the weight space, through the different sets of weights assigned to the individuals of the population. With the present approach, the raw weights are the same for all individuals, but the mapping function is different for each individual.

It must be pointed out that the combination of the network topology with a function for adapting the weights has also been explored by Lucas [106, 105, 107], by merging the learning rule and the network into a single oriented graph with typed nodes. Some of the nodes correspond to the neurons of the network, whereas others are used to compute the connection weights. When the nodes of the graph are evaluated repeatedly, the net effect is equivalent to a learning rule applied to the embedded network (Lucas presents a graph

Figure 7.3: (a) Fitting a curve to a known set of target weights. (b) Trying to figure out the correct curve based on fitness, in order to compute the target weights.

which simulates the execution of the backpropagation algorithm). With this method, the network and the learning rule are also simultaneously evolved for each task. However, it is difficult, if not impossible, to control the complexity of the network and the learning algorithm, as they are interwined in a single structure. With the mapping function approach, the architecture and the learning rule are separated in the genotype, and a crossover operator can be defined to evolve both parts simultaneously, but independently. Moreover, the complexity of the mapping function may depend on the diversity of the adapted weights but, in principle, it is independent of the size of the network.

7.3 Crossover operator

To evolve both parts of the genotype simultaneously, a combined crossover operator is defined. Firstly, recombination of the architectures of both parents is carried out similarly to that performed by the two-dimensional crossover operator introduced in Section 5.3. Secondly, the parse trees defining the mapping functions of both parents are recombined.

The architectures are recombined by selecting a node *a* in the first parent and a node *b* in the second parent, and replacing node *a* with node *b* in a copy of the first parent (the offspring). Depending on the types of node *a* and node *b*, the replacement is carried out as follows:

Both nodes are terminals: Node *b* replaces node *a*.

Node *b* is a terminal and node *a* is a neuron: Node *b* replaces node *a*.

Node *b* is a neuron and node *a* is a terminal: Performed as in the crossover operator described in Section 5.3. A modified node *b* is created to replace node *a* in the offspring, by manipulating the indexes of the connections of node *b* (note that the description of the nodes do not include either weights or biases anymore).

Figure 7.4: Combination of two neurons. (a) Node *a*. (b) Modified node *b* . (c) New node created by crossover with multiple connections. (d) Multiple connections deleted from the new node, to replace node *a* in the offspring.

Both nodes are neurons: After modifying node *b* as in the previous case, node *b* and node *a* are combined by selecting two random crossover points, one in each node, and replacing the connections to the right of the crossover point in node *a* with those to the right of the crossover point in node *b*, thus creating a new node to replace node *a* in the offspring. This process can easily create multiple connections between the same two nodes. These connections are deleted before the replacement of node *a* in the offspring, since they are meaningless in the current representation, as the tuning of the weights is performed by the mapping function (see Figure 7.4).

In a second step, a standard genetic programming crossover of the parse trees representing the mapping functions is performed by replacing a random subtree in the offspring with a random subtree selected from the second parent (see Figure 7.5).

In the crossover operator introduced with the two-dimensional representation, the limitation of the number of multiple connections was introduced to increase the efficiency of the search. Similarly, to prevent the excessive growth of the parse trees, a maximum depth is specified, and the selection of the subtrees for swapping is carried out according to this constraint.

A bias is also introduced to favor functions in the selection of the roots of the subtrees for

Figure 7.5: Crossover of parse trees. (a) Subtree selected in the first parent. (b) Subtree selected in the second parent. (c) Offspring.

crossover [95]. This is introduced due to the increase in the proportion of terminals in the parse trees in the course of evolution. Otherwise most of the time would be spent replacing terminals without much practical effect. In addition, two terminals representing variables are never selected as subtrees for crossover, since they encode the same weight.

7.4 A bit of complexity

One might argue that, since the adapted values for the weights have to be computed on the fly each time an individual is evaluated, the mapping function is actually equivalent to a conventional training procedure applied at each generation. Although this is true, the two approaches have considerably different complexities. In the case of conventional learning algorithms, the complexity is not in the final function that transforms the raw weights into adapted ones, but in the many training steps required to generate this function. In the case of the mapping function approach, the complexity is built in the evolutionary process, not in the function itself.

Another interesting point is that, for the mapping of weights, the size of the search space is constrained. For example, for grids with *N* internal nodes and *I* input neurons, the number of different genotypes for architectures with a single output neuron is given by $FP(N,I) \times TREES(S)$. ***FP(N,I)*** is the number of different configurations of neurons and connections that can be encoded in the first part of the genotype, and ***TREES(S)*** is the number of different parse trees with a maximum of *S* nodes that can be encoded in the second part of the genotype.

It is straightforward to prove that ***FP(N,I)*** is given by:

$$FP(N,I) = \sum_{n=0}^{N} \left(\sum_{i=1}^{M(N,n)} \left(\prod_{j=1}^{S(Config(i))} \left(\sum_{k=0}^{Max(i,j)} Perm(Max(i,j),k) \right) \right) \right)$$

where

M(N,n) = number of combinations of *N* internal neurons taken *n* at each time
Config(i) = *i*th neuron configuration (including the single output neuron) yielded by the M(N,n) combination
S(Config(i)) = number of neurons in the *i*th configuration
Max(i,j) = maximum number of connections of the *j*th neuron in the *i*th configuration
Perm(Max(i,j),k) = number of permutations of *Max(i,j)* connections taken *k* at each time (the order of the connections of a neuron is irrelevant)

Of course, many of the *FP(N,I)* configurations of neurons and connections may lead to functionally equivalent networks (this is connected to the *permutation* or *competing convention* problem [152, 72]), or even to the same network architecture. But, as the evolutionary process takes place in the genotype space, the size of the search space is given by $FP(N, I) \times TREES(S)$.

The number of parse trees with a maximum of *S* nodes depends on the *degree* of the nodes (the number of subtrees of a node), and the degree depends on the arity of the functions occupying the nodes. For example, for parse trees where each node can be either of degree 0 or 2 only, the number of different parse tree configurations with $2s+1$ nodes (*s* = 0,1,2...) is given by (adapted from [92]):

$$B(2s+1) = \frac{1}{s+1}\binom{2s}{s}$$

Such parse trees have s internal nodes and $(s+1)$ leaves. As internal nodes are occupied by functions, for a function set with F functions of arity 2 (e.g. the fundamental arithmetic operations), a single parse tree configuration with s internal nodes corresponds to F^s different parse trees. Similarly, as leaves are occupied by terminals, *T* different terminals lead to $T^{(s+1)}$ different parse trees. Consequently, $B(2s+1)F^sT^{(s+1)}$ different parse trees result from 2*s*+1 nodes. This implies that

$$TREES(S) = \sum_{s=0}^{S} \left(B(2s+1) F^s T^{(s+1)} \right)$$

Similarly to the networks, not all parse trees actually encode different mathematical expressions, but the size of the search space for parse trees is still given by $TREES(S)$. For example, if only the four fundamental arithmetic operations and only two different terminals are used, a maximum of 511 nodes (corresponding to a maximum depth of 8) result in $O(10^{766})$ different parse trees [1]. By computing *FP(N,I)* for the number of internal nodes used in the binary classification problems discussed in Section 6.2, the size of the search space for these tasks can be estimated (see Table 7.1).

Although large, the size of the search space yielded by the weight mapping approach is still much smaller than that yielded by the two-dimensional representation proposed in Chapter 5. If multiple connections are not allowed, the number of neurons and connections configurations for the two-dimensional representation is also given by *FP(N,I)*. For multiple connections, this number is considerably greater, as multiple connections can receive different weights, making them distinguishable from each other. Moreover, each of the multiple connections of a neuron can be replaced with another connection with a different weight available in the population. This means that the number of genotypes that can be achieved with the two-dimensional representation is huge, and dependent on the size of the population. If mutation is used to keep introducing new values for the weights at each generation, the search space of genotypes for the two-dimensional representation is in principle infinite.

However, in practical applications with thousands of individuals evolved for thousands of generations, only a very tiny fraction of the search space yielded by any method can be actually searched. Firstly, because even in the initial population individuals have many common features. Secondly, because the evolutionary process leads the population to converge to a small region of the search space during evolution. Consequently, it is unlikely

[1] In the computation of the number of binary trees with a specified number of nodes, no constraint was applied to the maximum depth of the tree. If such a restriction is imposed, the number of possible parse trees is reduced, but the general idea still holds

Table 7.1: Size of the search space for the weight mapping approach.

Task	FP(N,I)	$FP(N,I) \times TREES(S)$
XOR	$O(10^{49})$	$O(10^{815})$
3 parity	$O(10^{59})$	$O(10^{825})$
4 parity	$O(10^{69})$	$O(10^{835})$
5 parity	$O(10^{80})$	$O(10^{846})$
T-C	$O(10^{416})$	$O(10^{1182})$

that the constraint imposed by the mapping of weights on the size of the search space brings any intrinsic advantage or disadvantage.

7.5 Experimental results

To evaluate the performance of the mapping function approach, some of the experiments carried out for the two-dimensional representation were repeated.

The same conditions of the experiments performed in Chapter 5 were applied to the experiments with the mapping function approach, and the following additional conditions were applied to the parse trees:

- Parse trees initialized by the ramped half-and-half procedure [95].
- Maximum depth: 8.
- Fraction of random constants included as terminals: 25%. This means that, on the average, when a parse tree was created, the variable representing a weight was assigned to 75% of its terminals, and the remaining 25% of the terminals were randomly initialized with real-valued constants.
- The roots of the subtrees for crossover were selected using a probability distribution

which allocated 80% of the crossover points to the internal nodes of the parse trees and 20% to the terminals.

- Function set to select the internal nodes of the parse trees: [* , - , + , *PDIV*], where *PDIV* stands for protected division (returns the numerator if the denominator is zero).

- The raw weights were randomly initialized without repetition. This is necessary because two connections may require different adapted weights, and to guarantee that the mapping function exists, their raw weights must also be different.

7.5.1 Binary classification problem

To show the performance of the method proposed, it was first applied to the test suite of binary classification problems: the odd-2 (XOR), 3, 4 and 5 parity, the symmetry and the *T-C* problems. A Summary of the results obtained in 50 independent runs is shown in Table 7.2.

By comparing these results to those in Table 6.1 it can be realized that, in terms of computational effort to find a solution, the new method fared better on 2 of the tasks and worse on the other 4.

A typical solution to the odd-3 parity problem illustrates the process of building the network. Firstly, the weights are read from the list of raw values and assigned to the architecture evolved. This results in the two-dimensional representation in Figure 7.6a. Subsequently, the raw weights are adapted by the mapping function evolved simultaneously to the architecture, leading to the two-dimensional structure in Figure 7.6b. The two-dimensional representation can then be decoded into the corresponding network shown in Figure 7.7a, which can still be simplified to the network Figure 7.7b if desired. The mapping of the raw weights to adapted ones is performed by the function encoded in the parse tree in Figure 7.8a, whose mathematical expression is given in Figure 7.8b. The process of building the mapping function can be visualized by considering the evolution of this function for the best individual in the population as shown in Figure 7.9.

Figure 7.6: Two-dimensional representation of a typical solution to the odd-3 parity problem. (a) With raw weights. (b) With values adapted by the function in Figure 7.8b.

Figure 7.7: (a) Decoded network of the solution to the odd-3 parity problem, obtained by replacing connections from terminals in the internal layer (and removing the terminals) with connections from corresponding terminals in the input layer, and merging the resulting multiple connections by adding up their weights. (b) Further simplification of the network by removing neurons which do not contribute to the output of the network.

Figure 7.8: (a) Parse tree representation of the evolved mapping function for the solution to the odd-3 parity problem. (b) Corresponding mathematical expression.

Figure 7.9: Evolution of the mapping function of the best individual in the population for the odd-3 parity problem. For plotting purposes only, the values of the adapted weights were squashed into the interval [-1.0, +1.0] by the hyperbolic tangent.

Table 7.2: Summary of the results to obtain a first solution to the binary classification problems using a single internal layer containing no terminals initially.

TASK	GEN (σ)	NEURONS				CONNECTIONS				EFFORT
		min	avg	max	σ	min	avg	max	σ	
XOR	4.1 (8.5)	2	8.9	10	1.7	7	32.6	42	7.6	3,000
3 parity	56.0 (74.5)	1	7.8	10	1.8	7	34.6	57	8.6	33,600
4 parity	277.0 (191.7)	3	6.9	10	1.8	21	38.8	57	9.8	224,000
5 parity	406.9 (166.7)	2	4.8	9	2.1	16	31.2	57	11.6	481,600
Symmetry	348.1 (193.8)	5	7.5	10	1.4	24	37.5	48	7.3	248,000
T-C	48.0 (75.0)	2	11.3	15	2.9	28	100.5	159	30.3	30,600

7.5.2 Pole-balancing problem

The experiments with the pole-balancing problem were carried out using the same approach described in Section 6.3.1 for the bang-bang and the continuous control action. Table 7.3 shows the results of 50 independent runs, and also results transcribed from Table 6.6 for comparison.

For the pole-balancing problem, the weight mapping approach was superior, in the bang-bang as well as in the continuous control action, in terms of the computational effort to find a solution. With regard to the average number of generations to find a solution, the mapping of weights fared better in the bang-bang case, whereas the two-dimensional representation did better in the continuous case. This is confirmed by the average number of fitness evaluations to find a solution shown in Tables 7.4 and 7.5.

Tables 7.4 and 7.5 show the results of the generalization test carried out with the weight mapping approach, as well as corresponding results for the two-dimensional representation and cellular encoding taken from Tables 6.8 and 6.9. In comparison to cellular encoding, the method was substantially superior in terms of number of fitness evaluations to find a solution in the bang-bang case. For the continuous control action, the mapping of weights

approach yielded results superior to those reported in [194], and inferior to those reported in [71]. The generalization power of the solutions presented was better compared to the result in [71], and worse in comparison to the results in [194].

Table 7.3: Comparison of results to obtain a first solution to the pole-balancing problem using the weight mapping approach and the two-dimensional representation (values transcribed from Table 6.6). In the first column, *b* and *c* indicate bang-bang and continuous control action, respectively.

METHOD (control action)	GEN (σ)	NEURONS				CONNECTIONS				EFFORT
		min	avg	max	σ	min	avg	max	σ	
Weight mapping (b)	3.6 (5.7)	2	4.1	8	1.3	9	19.6	32	5.7	1,400
Weight mapping (c)	22.2 (34.5)	0	4.2	8	2.0	3	19.4	36	8.2	1,700
Two-dimensional (b)	7.2 (7.3)	1	4.3	8	1.7	7	17.6	30	5.4	2,600
Two-dimensional (c)	9.4 (8.1)	0	3.8	8	1.8	3	16.8	36	7.4	3,600

Table 7.4: Comparison of results obtained with the weight mapping approach, cellular encoding and the two-dimensional representation (results copied from Table 6.8), using a bang-bang control action.

Results of the learning process		Results of the generalization test			
Method	Number of fitness evaluations (σ)	Number of successful runs			
		Best	Mean	Worst	σ
Cellular encoding [194]	2,234 (N/A)	N/A	430	N/A	N/A
Weight mapping	349.2 (395.5)	372	319.8	185	41.2
Two-dimensional	612.4 (524.4)	372	329.5	195	34.4

Table 7.5: Comparison of results obtained with the weight mapping approach, cellular encoding and the two-dimensional representation (results copied from Table 6.9), using a continuous control action.

Results of the learning process		Results of the generalization test			
Method	Number of fitness evaluations (σ)	Number of successful runs			
		Best	Mean	Worst	σ
Cellular encoding [194]	19,011 (N/A)	N/A	386	N/A	N/A
Cellular encoding [71]	1,400 (N/A)	N/A	250	N/A	N/A
Weight mapping	1656.8 (2414.9)	376	313.4	209	41.4
Two-dimensional	775.9 (581.1)	361	301.5	217	32.8

7.5.3 Tracker problem

The recurrent version of the method was tested using the tracker problem discussed in Section 6.3.3. In 20 independent runs, the average number of positions cleared by the best individuals evolved was 80.4, with a standard deviation of 6.6. By assigning additional time steps to the best individuals evolved, in 50% of the runs, the best individuals cleared the 89 positions in less than 290 time steps. A solution with 6 hidden neurons and 48 connections was found (see Figure 7.10), which was able to clear the entire trail in 199 time steps, amounting to a computational effort of 8,766,000. Similarly to the experiments in Section 6.3.3, the solution found did not make use of either the *turn left* or the *wait* options to traverse the trail. Five additional solutions were also obtained in this successful run by allowing the evolutionary process continue to the maximum number of generations specified, but these multiple solutions were not included in the statistics just described.

It is remarkable, that the solution in Figure 7.10, using the same actions as the previous two solutions presented in Section 6.3.3, still yields a different strategy to clear the trail in 199 steps. This can be realized by comparing Figures 7.11, 7.12 and 7.13 to the corresponding ones in Section 6.3.3.

Figure 7.10: Full solution to the tracker problem. Values in the circles are biases. Interpretation of the output of the network is performed as in Section 6.3.3.

Figure 7.11: Actions taken by the tracker in Figure 7.10 as a function of the number of steps.

Figure 7.12: Number of track positions cleared as a function of the number of steps by the network in Figure 7.10.

Figure 7.13: Position along the trail at each time step. Steps missing are spent executing the *turn right* action.

7.6 Summary

The performance of the weight mapping approach was demonstrated to be superior to the two-dimensional representation in some of the experiments and worse in others. These results indicate that, for some applications, the mapping of weights is a viable alternative to the representation proposed in Chapter 5. Moreover, the mechanism proposed for encoding the weights is independent of the size of the network, a feature that can be explored in combination with other methods.

Chapter 8

Further research and summary

8.1 Introduction

The new method for the synthesis of artificial neural networks discussed in this work was implemented using standard genetic algorithm parameters. This was done on purpose to demonstrate that the method works without having to fine tune parameters. However, this does not imply that more elaborate approaches could not be beneficial. It was also mentioned that the extension to the two-dimensional representation presented in Chapter 7 could be combined with other methods. To address these issues, plans for future research are discussed in the next sections.

8.2 Variations on the basic crossover operator

Other variants of the proposed crossover operator could be investigated. For example, connections not allowed due to connectivity constraints might be kept in the genotype instead of being deleted. They could be ignored in the evaluation of the fitness of the individual, but would still be present for crossover, and might be reused later. On the one hand, this has the advantage that the elimination of genetic material is reduced (elimination to comply

with the maximum number of multiple connections would still remain). On the other hand, to keep a great number of introns (parts of the genotype which are not expressed in the phenotype) might reduce the efficiency of crossover in promoting evolution.

Another possibility is to allow a connection to acquire a new weight when crossover replaces a neuron with another neuron in the offspring. This can be implemented by choosing the crossover points in the neurons between the index of the connected node and the weight. In this case, it would make sense to implement the biases as weights of connections from an additional node of constant output, to be evolved as an ordinary connection weight. By assigning new weights to the connections, the flexibility of the crossover operator would be increased.

A third alternative worth investigating, is to perform two-point crossover between two neurons, instead of the one-point crossover used in this work. To do that, the neurons should be defined as circular lists, containing the bias and the connections. In this case, it would also make sense to implement the biases as ordinary weights, allowing them to be eliminated and reintroduced by crossover. The bias deleted from the list of connections of a neuron would receive a null default value, whereas multiple biases would be added up.

In genetic programming, a bias is usually introduced in the selection of roots of subtrees for crossover, to counteract the increase in the proportion of terminals in the parse trees. With the grid representation this is not so serious a problem because the genotype has a fixed size. However, as mentioned in Section 5.3, to reduce the complexity of the networks, the crossover operator proposed, slightly favors the replacement of neurons with terminals. Consequently, it may happen that late in the evolution, terminals start to dominate the genotype as internal nodes. In this case, most of the time may be spent replacing terminals with terminals without much practical effect. To bypass this problem, crossover could be designed to favor the selection of neurons instead of terminals later in the evolution.

8.3 Splitting the crossover operator

The crossover operator defined for the weight mapping approach could be used to define two specialized operators: an *architecture-evolving* operator to evolve the topology, and a *weight-evolving* operator to evolve the weights and biases. The former would be implemented by the part of the crossover operator introduced in Section 6.3 to evolve the architecture, whereas the latter would be the standard genetic programming crossover operator which evolves the mapping function. Both operators could be independently applied with different probabilities.

For instance, learning could be stressed at the expense of evolving the architecture, by using a small architecture-evolving to weight-evolving probability ratio. The weight-evolving operator could also be used for training fixed architectures, by initializing all individuals in the population with the same topology (actually, the architecture part of the genotype would be eliminated since it would not be evolved anymore). In this case, the adapted weights and biases would be evolved, in a kind of training by genetic programming, whereby GP would directly adapt the weights for each task.

As discussed in Section 2.2, conventional learning procedures have many limitations. Genetic programming has been used to evolve new learning rules which eliminate some of them. However, the new evolved rules are general purpose algorithms, and as such, can not exploit any peculiarity of the error surface defined in the weight space of a particular problem. Moreover, they still work based on local information which may mislead the training process. In contrast, the weight mapping approach builds a function which is tailored to the error surface of the task being addressed. Moreover, it is not based on local information, since genetic programming directly evolves the weights.

Genetic algorithms have also been used for training. However, training by GAs becomes intractable, but for small architectures, as GA-based methods directly encode all the weights into the genotype. Of course, the difficulty of training by any method increases with the size of the network, as more free parameters have to be adjusted. However, the size of the

genotype has a direct effect on the efficiency of the search, and the size of the representation in the weight mapping approach is independent of the size of the network. Actually, this is a feature which the new approach shares with conventional learning algorithms, which are also independent of the size of the network. As a consequence, the weight mapping approach might be competitive, even in those applications where conventional learning algorithms are expected to be more efficient than GA-based methods. Moreover, other variations of the method can be investigated, which may further improve its performance, for example:

- More sophisticated genetic programming techniques, such as automatic defined functions can be explored [95]. Also, in addition to the fundamental arithmetic operations, other functions (exponential, square root, logarithm, trigonometric, etc.) can be included in the function set. This would allow mathematical expressions to adjust the weights to be evolved more efficiently. Operators for conditional branching, iterative loops, etc. can also be included in the function set, to evolve complex algorithms to compute the adapted weights.

- A wrapping function can be used to constrain the output of the parse tree to a specified range. This would bias the search to avoid large values of the weights, preventing the saturation of the activation function.

- The weights of other connections can be included as input to the parse tree. This can be carried out by including more variables in the terminal set. For example, the additional variables can represent the weights of the incoming connections of a neuron (if a particular neuron does not have enough connections to assign values to all terminals, null values can be assigned to those which do not have a corresponding connetion). That means, the value of the adapted weight computed by the parse tree would be sensitive to the context (conventional learning methods use a similar strategy, as the update of a particular weight depends on the value of other weights).

- The random constants initialized when a parse tree is created can be made neuron dependent. For example, each random constant can be replaced with a vector of random constants, whose length is the number of neurons in the network. When the mapping function of an individual is applied to the raw weight of a connection of a particular neuron, the value of each constant is read from the entry corresponding to the neuron. That means, although the form of the mapping function would be the same for all connection weights, there would be neuron dependent parameters. This would enrich the method.

It must be pointed out that the initial set of raw weights is not a starting point as in conventional training procedures. Genetic algorithms use bitstrings or real-valued vectors to encode the weights, i.e. each individual of the population represents a different point in the weight space, With training by weight mapping, each individual of the population is a function, which must be applied to the raw set of weights, to yield adapted ones. That means, the different points in the weight space are actually represented by the output of the different mapping functions. The set of raw weights is only an ingenious way of encoding the adapted weights as output of functions. Consequently, the effectiveness of the weight mapping approach should be fairly independent of the initialization of the set of raw weights.

These features make clear the potential of the weight mapping approach as a new form of training of artificial neural networks by genetic programming.

8.4 Weight mapping combined with indirect encoding methods

8.4.1 Weight mapping with fixed size representation

Any method to evolve artificial neural networks which combines fixed size representation with training at each generation, could benefit from the weight mapping approach, by replacing the training process, with adaptation of the weights through the mapping function. In Chapter 4 it was pointed out that grammar-based methods could benefit from a mechanism for encoding the weights which did not depend on the size of the network (**Lesson 4**). This is exactly the appeal of the weight mapping approach.

For example, Kitano's method, described in Section 4.3.4, is a perfect candidate for this symbiosis, as for a maximum number of rewriting cycles, the connectivity matrices generated are all of the same size. Consequently, it is straightforward to define a set of raw weights to be assigned to the networks. The genotype would consist of the grammar rules for generating the connectivity matrix, plus the mapping function encoded in the parse tree.

8.4.2 Weight mapping with variable size representation

The definition of a set of raw weights is not limited to fixed size representations. Methods which use a variable size representation can also benefit from the mapping function approach. For example, in cellular encoding (see Section 4.5), different genotypes may lead to decoded networks with a different number of neurons. However, it is possible to define a set of raw weights by considering the number of neurons of the biggest decoded network in the initial population, with the maximum number of connections allowed by connectivity constraints. If in later generations additional weights are necessary (because the networks get bigger), new values are included in the original set. The genotype would consist of the standard grammar-tree of cellular encoding, and the parse tree encoding the mapping

function. In cellular encoding, the procedure of resetting the reading head to the root of the grammar-tree imposes a regularity in the distribution of the weights in the network. With the weight mapping approach this would not happen. Regularity, if any, would be encoded in the mapping function by the evolutionary process.

8.5 Summary

In this work, a new method for the synthesis of artificial neural networks based on a two-dimensional representation has been proposed. The method is able to evolve the architecture and the weights concurrently, without further local optimization of the weights by a learning procedure at each generation.

In order to make the thesis self-contained, an introduction about artificial neural networks was given, to discuss the main aspects involved in the building of ANNs. For the same reason, an introduction about genetic algorithms was presented.

To address the issue of integrating artificial neural networks and genetic algorithms, a literature review was presented, where previous applications of GAs to the building of ANNs were discussed. Two objectives were achieved with this literature review. Firstly, the deficiencies of existing methods were identified. Secondly, based on these shortcomings, criteria for designing a good method for evolving ANNs were defined.

On the grounds of these criteria, a new general purpose two-dimensional representation for the evolution of artificial neural networks was then proposed. The new approach was designed to comply with all the conditions specified for a good method. In addition, a new pruning strategy was introduced, which makes it possible to achieve solutions of varying degrees of complexity, without the need for the specification of arbitrary problem- dependent parameters. The ability of the new approach to efficiently evolve feedforward as well as recurrent architectures was demonstrated in a test suite of standard benchmark problems. It was also demonstrated that the performance of the two-dimensional representation in the tasks investigated was superior to the methods used for comparison.

An extension of the two-dimensional representation based on the evolution of a mapping function to adapt the weights of the network was also proposed. The new method uses genetic programming to evolve a mapping function to adapt the weights, whereas the architecture is evolved by GAs. The approach opens new possibilities for genetic programming in the evolution of artificial neural networks.

Finally, suggestions for future research were outlined, including variations on the basic form of the new crossover operator proposed for the two-dimensional representation, and the combination of the weight mapping approach with other methods, so as to provide them with a better alternative for the evolution of the connection weights.

References

[1] L. Almeida. Multilayer perceptrons. In E. Fiesler and R. Beale, editors, *Handbook of Neural Computation*, pages C1.2:1–C1.2:30. Oxford University Press, 1997.

[2] H. Andersen and A. Tsoi. A constructive algorithm for the training of a multilayer perceptron based on the genetic algorithm. *Complex Systems*, 7:249–268, 1993.

[3] C. Anderson. Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*, 9:31–37, 1989.

[4] P. Angeline. Genetic programming: a current snapshot. In D. Fogel and W. Atmar, editors, *Proceedings of the Third Annual Conference on Evolutionary Programming*, pages 224–232, San Diego, USA, Feb. 1994.

[5] P. Angeline. Subtree crossover causes bloat. In J. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. Fogel, M. Garzon, D. Goldberg, H. Iba, and R. Riolo, editors, *Proceedings of the Third Annual Conference on Genetic Programming*, pages 745–752. Morgan Kauffmann, Jul. 1997.

[6] P. Angeline, G. Saunders, and J. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1):54–65, 1994.

[7] P. Arena, R. Caponetto, I. Fortuna, and M. Xibilia. (M.L.P.) optimal topology via genetic algorithms. In *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms (ANNGA)*, pages 670–674, 1993.

[8] T. Bäck. Evolutionary algorithms. *ACM SIGBIO Newsletter*, pages 26–31, 1992.

[9] T. Bäck. Evolutionary algorithms: Comparison of approaches. In R. Paton, editor, *Computing with Biological Metaphors*, pages 227–243. Chapman and Hall, 1994.

[10] T. Bäck. Binary strings. In T. Bäck, D. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, pages C1.2:1–C1.2:3. Oxford University Press, 1997.

[11] T. Bäck, D. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*. Oxford Press, 1997.

[12] T. Bäck, D. Fogel, D. Whitley, and P. Angeline. Mutation. In T. Bäck, D. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, pages C3.2:1–C3.2:14. Oxford University Press, 1997.

[13] T. Bäck, U. Hammel, and H. Schwefel. Evolutionary computation: Comments on the history and current state. *IEEE Transactions on Evolutionary Computation*, 1(1):3–17, Apr. 1997.

[14] T. Bäck and H. Schwefel. Evolutionary computation: an overview. In *Proceedings of the third IEEE Conference on Evolutionary Computation*, pages 20–29. IEEE Press, 1996.

[15] K. Balakrishnan and V. Honavar. Properties of genetic representations of neural architectures. In *Proceedings of World Congress on Neural Networks*, Washington DC, USA, 1995.

[16] P. Baldi. Gradient descent learning algorithm overview: A general dynamical systems perpective. *IEEE Transactions on Neural Networks*, 6(1):182–195, Jan. 1995.

[17] A. Banerjee. Initializing neural networks using decision trees. In *Proceedings of the International Workshop on Computational Learning and Natural Learning Systems*, 1994.

[18] W. Banzhaf, P. Nordin, R. Keller, and F. Francone. *Genetic Programming, An Introduction. On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kauffmann Publishers, Inc., 1998.

[19] A. Barron. Approximation and estimation bounds for artificial neural networks. In *Proceedings of the Fourth Annual Workshop on Computational Learning Theory*, pages 243–249, 1991.

[20] A. Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information Theory*, 39(3):930–945, May 1993.

[21] A. Barto, R. Sutton, and C. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC–13(5):834–846, Sept/Oct. 1983.

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

[23] K. Belew, J. McInerney, and N. Schraudolph. Evolving networks: using the genetic algorithm with connectionist learning. In C. Langton, C. Taylor, J. Farmer, and S. Rasmussen, editors, *Artificial Life II, Santa Fe Institute Studies in the Sciences of Complexity*, volume X, pages 511–547. Addison-Wesley, 1991.

[24] S. Bengio, Y. Bengio, and J. Cloutier. Use of genetic programming for the search of a learning rule for neural networks. In *Proceedings of the First Conference on Evolutionary Computation, IEEE World Congress on the Computational Intelligence*, pages 324–327, 1994.

[25] A. Bergman. An evolutionary approach to designing neural networks. In R. Paton, editor, *Computing with Biological Metaphors*, pages 298–308. Chapman and Hall, 1994.

[26] E. Boers, H. Kuiper, B. Happel, and I. Sprinkhuizen-Kuiper. Designing modular artificial neural networks. Technical Report 93-24, Department of Computer Science, Leiden University, Netherlands, Sept. 1993.

[27] L. Booker, D. Fogel, D. Whitley, and P. Angeline. Recombination. In T. Bäck, D. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, pages C3.3:1–C.3.3:27. Oxford Press, 1997.

[28] S. Bornholdt and D. Graudenz. General asymmetric neural networks and structure design by genetic algorithms. *Neural Networks*, 5:327–334, 1992.

[29] N. Bose and P. Liang. *Neural Network Fundamentals with Graphs, Algorithms, and Applications*. McGraw-Hill, 1996.

[30] H. Braun and P. Zagorski. ENZO-M - a hybrid approach for optmizing neural networks by evolution and learning. In Y. Davidor, H. Schwefel, and H. Manner, editors, *Parallel Problem Solving from Nature (PPSN3)*. Springer-Verlag, 1994. Lecture Notes in Computer Science 866.

[31] T. Caudell and C. Dolan. Parametric connectivity: training of constrained networks using genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 370–374, 1989.

[32] D. Chalmers. The evolution of learning: an experiment in genetic connectionism. In D. Touretzky, J. Elman, T. Sejnowski, and G. Hinton, editors, *Proceedings of the 1990 Connectionist Models Summer School*, San Mateo, CA, 1990. Morgan Kaufmann.

[33] K. Chellapilla. Automatic generation of nonlinear optimal control laws for broom balancing using evolutionary programming. In *Proceedings of the IEEE International Conference on Evolutionary Computation (ICEC)*, pages 195–200, 1998.

[34] S. Cho and K. Shimohara. Modular neural networks evolved by genetic programming. In *Proceedings of the Third IEEE International Conference on Evolutionary Computation (ICEC), Special Session on Evolutionary Artificial Neural Networks*, 1996.

[35] M. Clergue and P. Collard. Genetic algorithm for artificial neurogenesis. In *Proceedings of the IEEE International Conference on Evolutionary Computation (ICEC)*, pages 410–415, 1998.

[36] T. Collins. Understanding evolutionary computing: a hands on approach. In *Proceedings of the IEEE International Conference on Evolutionary Computation (ICEC)*, pages 564–569, 1998.

[37] P. Courrieu. A convergent generator of neural networks. *Neural Networks*, 6:835–844, 1993.

[38] J. Crespo and E. Mora. Tests of different regularization terms in small networks. In *Proceedings of the International Workshop on Artificial Neural Network (IWANN)*, pages 179–184. Springer-Verlag, 1993.

[39] Y. Cun, J. Denker, and S. Solla. Optimal brain damage. In D. Touretzky, editor, *Advances in Neural Information Processing II*, pages 598–605, 1989.

[40] G. Cybenko. Approximation by superposition of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[41] D. Dasgupta and D. McGregor. Designing application-specific neural networks using the structured genetic algorithm. In L. Whitley and J. Schaffer, editors, *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN)*, pages 87–96. IEEE Computer Society Press, Jun. 1992.

[42] D. Dasgupta and D. McGregor. Genetically designing neuro-controllers for a dynamic system. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, volume 3, pages 2951–2954, Nagoya, Japan, Oct. 1993.

[43] L. Davis. *Handbook of Genetic Algorithms*. Van Nostrand Rheingold, New York, NY, 1991.

[44] R. Dawkins. *The Blind Watchmaker*. Penguin Books, London, UK, 1986.

[45] L. Eshelman. Genetic algorithms. In T. Bäck, D. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, pages B1.2:1–B1.2:11. Oxford Press, 1997.

[46] S. Fahlman and C. Lebiere. The cascade-correlation learning architecture. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 524–532. Morgan Kaufmann, 1990.

[47] S. Fahlman and C. Lebiere. A recurrent cascade-correlation learning architecture. In R. Lippmann, J. Moody, and D. Touretzky, editors, *Advances in Neural Information Processing Systems*, volume 3, pages 190–196. Morgan Kaufmann, 1991.

[48] A. Fekadu, E. Hines, and J. Gardner. Genetic algorithms design of neural net based electronic nose. In *First International Conference on Neural Networks and Genetic Algorithms (ICNNGA)*, pages 691–698, University of Innsbruck, Austria, Apr. 1993.

[49] E. Fiesler and R. Beale, editors. *Handbook of Neural Computation*. Oxford University Press, 1997.

[50] D. Fogel. *Evolutionary computation: toward a new philosophy of machine Intelligence*. IEEE Press, Piscataway, NJ, USA, 1995.

[51] D. Fogel. Real-valued vectors. In T. Bäck, D. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, pages C1.3:1–C1.3:2. Oxford University Press, 1997.

[52] D. Fogel and P. Angeline. A review of efforts combining neural networks and evolutionary computation. In *Proceedings of the International Society for Optical Engineering (SPIE)*, volume 2492, pages 586–589, 1995.

[53] D. B. Fogel. An introduction to simulated evolutionary optimization. *IEEE Transactions on Neural Networks*, 5(1):3–14, Jan. 1994.

[54] S. Forrest. Genetic algorithms: Principles of natural selection applied to computation. *Science*, 261:872–878, Aug. 1993.

[55] M. Frean. The upstart algorithm: a method for constructing and training feed-forward neural networks. *Neural Computation*, 2:198–209, 1990.

[56] K. Fredrikson. Genetic algorithms and generative encoding of neural networks for some benchmark classification problems. In *Proceedings of the Third Nordic Workshop on Genetic Algorithms*, pages 123–134, 1997.

[57] C. Friedrich and C. Moraga. Using genetic engineering to find modular structures and activation functions for architectures of artificial neural networks. In *Proceedings of the Fifth Fuzzy Days*, pages 150–161, 1997.

[58] S. Fujita and H. Nishimura. An evolutionary approach to associative memory in recurrent neural networks. *Neural Processing*, 1(2), 1994.

[59] B. Fullmer and R. Miikkulainen. Using marker-based genetic-encoding of neural networks to evolve finite-state behavior. In *Proceedings of the First European Conference on Artificial Life*, Paris, 1991.

[60] J. Ghosh and K. Tumer. Structural adaptation and generalization in supervised feed-forward networks. *Journal of Artificial Neural Networks*, 1(4):431–458, 1994.

[61] C. Giles, D. Chen, G. Sun, H. Chen, Y. Lee, and W. Goudreau. Constructive learning of recurrent neural networks: Limitations of recurrent casade correlation and a simple solution. *IEEE Transactions on Neural Networks*, 6(4):829–836, Jul. 1995.

[62] C. Giles and W. Omlin. Pruning recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(5):848–855, 1994.

[63] D. Goldberg. *Genetic algorithm in search, optimization and machine learning*. Addison-Wesley, Reading, Massachusets, 1989.

[64] D. Goldberg. Genetic algorithms and walsh functions: Part 1, a gentle introduction. *Complex Systems*, 3:129–152, 1989.

[65] D. Goldberg. Real-coded genetic algorithms, virtual alphabets and blocking. *Complex Systems*, 5:119–167, 1991.

[66] D. Goldberg. Genetic and evolutionary algorithms come of age. *Communications of the ACM*, 37(3):113–119, Mar. 1994.

[67] A. Gorban and D. Wunsch II. The general approximation theorem. In *Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN)*, pages 1271–1274, 1998.

[68] F. Gruau. Genetic micro programming of neural networks. In K. Kinnear Jr., editor, *Advances in Genetic Programmming*, chapter 24, pages 495–518. MIT Press, 1994.

[69] F. Gruau. *Neural network synthesis using cellular encoding and the genetic algorithm*. PhD thesis, Laboratoire de L'informatique du Parallélisme, Ecole Normale Supériere de Lyon, Lyon, France, 1994.

[70] F. Gruau and D. Whitley. The cellular developmental of neural networks: the interaction of learning and evolution. Research Report 93–04, Laboratoire de L'informatique du Parallélisme, Ecole Normale Supériere de Lyon, France, 1993.

[71] F. Gruau, D. Whitley, and L. Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In J. Koza, D. Goldberg, D. Fogel, and R. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 81–89, Stanford University, CA, USA, Jul. 1996. MIT Press.

[72] P. Hancock. Genetic algorithms and permutation problems: a comparison of recombination operators for neural structure specification. In D. Whitley, editor, *Proceedings of COGANN Workshop, IJCNN*. IEEE Computer Society Press, 1992.

[73] P. Hancock. Pruning neural nets by genetic algorithm. In I. Aleksander and J. Taylor, editors, *Proceedings of the International Conference on Artificial Neural Networks*. Elsevier, 1992.

[74] S. Harp and T. Samad. The genetic synthesis of neural network architecture. In L. Davis, editor, *Handbook of Genetic Algorithms*, pages 202–221. Van Nostrand Rheingold, New York, NY, 1991.

[75] S. Harp, T. Samad, and A. Guha. Toward the genetic synthesis of neural networks. In J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms (ICGA)*, pages 360–369, San Mateo, CA, USA, 1989.

[76] S. A. Harp, T. Samad, and A. Guha. Designing application-specific neural networks using genetic algorithms. In D. S. Touretsky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 447–454. Morgan Kaufmann, 1990.

[77] S. Haykin. *Neural networks, a comprehensive foundation*. Macmillan Publishing Company, 113 Sylvan Avenue, Englewood Cliffs, NJ 07632, 1994.

[78] J. Hertz, K. Anders, and R. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley Publishing Company, Redwood, California, 1991.

[79] G. Hinton. Connectionist learning procedures. *Artificial Intelligence*, 40(1):185–234, 1989.

[80] J. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor, Michigan, 1975.

[81] K. Hornik. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

[82] S. Huang and Y. Huang. Bounds on the number of hidden neurons in multilayer perceptrons. *IEEE Transactions on Neural Networks*, 2(1):47–55, Jan. 1991.

[83] D. Hush and B. Horne. Progress in supervised neural networks. *IEEE Signal Processing Magazine*, pages 8–39, Jan. 1993.

[84] A. Imada and K. Araki. Evolution of a hopfield associative memory by the breeder genetic algorithm. In *Proceedings of the International Conference on Genetic Algorithms (ICGA)*, pages 784–791, 1993.

[85] A. Jain, J. Mao, and K. Mohiuddin. Artificial neural networks: a tutorial. *IEEE Computer*, pages 31–44, Mar. 1996.

[86] D. Janson and J. Frenzel. Training product unit neural networks with genetic algorithms. *IEEE Expert*, pages 26–33, Oct. 1993.

[87] D. Jefferson, R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Korf, C. Taylor, and A. Wang. Evolution as a theme in artificial life: The genesys/tracker system. In C. Langton, C. Taylor, J. Farmer, and S. Rasmussen, editors, *Artificial Life II, Santa Fe Institute Studies in the Sciences of Complexity*, volume X, pages 549–578. Addison-Wesley, 1991.

[88] J. Kim and H. Myung. Evolutionary programming techniques for constrained optimization problems. *IEEE Transactions on Evolutionary Computation*, 1(2):129–140, Jul. 1997.

[89] H. Kitano. Designing neural networks using genetic algorithms with graph generation systems. *Complex Systems*, pages 461–476, 1990.

[90] H. Kitano. Neurogenetic learning: an integrated method of designing and training neural networks using genetic algorithms. *Physica D*, 75:225–238, 1994.

[91] J. Knowles, D. Corne, and M. Bishop. Evolutionary training of artificial neural networks for radiotherapy treatment of cancers. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, pages 398–403, 1998.

[92] D. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison-Wesley Publishing Company, 1968.

[93] P. Korning. Training of neural networks by means of genetic algorithm working on very large chromosomes. Technical Report DAIMI PB-486, Computer Science Department, Aarhus University, Denmark, 1994.

[94] J. Koza. Genetically breeding populations of computer programs to solve problems in artificial intelligence. In *Proceedings of the Second International Conference on Tools for AI*, pages 819–827, Los Alamitos, CA, Nov. 1990.

[95] J. Koza. *Genetic Programming, on the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, Massachusets, 1992.

[96] J. Koza and J. Rice. Genetic generation of both the weights and architecture for a neural network. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, volume 2, pages 397–404, 1991.

[97] I. Kuscu. Evolving a generalized behaviour: Artificial ant problem revisited. In V. Porto, N. Saravanan, D. Waagen, and A. Eiben, editors, *Proceedings of the Seventh Annual Conference on Evolutionary Programming (EP)*, volume 1447 of *Lecture Notes in Computer Science*, pages 799–808. Springer-Verlag, 1998.

[98] I. Kuscu and C. Thornton. Design of artificial neural networks using genetic algorithms: review and prospect. In C. et al. Bozsahin, editor, *Proceedings of the Third Turkish Symposium on Artificial Intelligence and Neural Networks*, pages 411–420, 1994.

[99] W. Langdon. The evolution of size in variable length representation. In *Proceedings of the IEEE International Conference on Evolutionary Computation (ICEC)*, pages 633–638, 1998.

[100] W. Langdon and R. Poli. Why ants are hard. Technical report CSRP-98-04, School of Computer Science, The University of Birmingham, 1998.

[101] M. Lewis, A. Fagg, and A. Solidum. Genetic programming approach to the construction of a neural network for control of a walking robot. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, volume 3, pages 2618–2623, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[102] A. Lindenmayer. Mathematical models for cellular interactions in development. *Journal of Theoretical Biology*, 18:280–299, 1968.

[103] R. Lippmann. A introduction to computing with neural nets. *IEEE Accoustics, Speech and Signal Processing Magazine*, 4:4–22, Apr. 1987.

[104] A. Logar, E. Corwin, and W. Oldham. A comparison of recurrent neural network learning algorithms. In *IEEE International Conference on Neural Networks*, volume 1–3, pages 1129–1134, Stanford University, 1993.

[105] S. Lucas. Growing adaptive neural networks with graph grammars. In M. Verleyssen, editor, *Proceedings of the European Symposium on Artificial Neural Networks*, pages 235–240, 1995.

[106] S. Lucas. The open ended evolution of neural networks. In *Proceedings of the First IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, Sheffield, UK, Sept. 1995.

[107] S. Lucas. Evolving neural network learning behaviours with set-based chromosomes. In *Proceedings of the European Symposium on Artificial Neural Networks*, pages 291–296, 1996.

[108] M. Mandischer. Representation and evolution of neural networks. In *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms (ANNGA)*, pages 643–649, 1993.

[109] M. Mandischer. Evolving recurrent neural networks with non-binary encoding. In *Proceedings of the Second IEEE Conference on Evolutionary Computation (ICEC)*, volume 2, pages 584–589, Perth, Australia, Nov. 1995.

[110] V. Maniezzo. Genetic evolution of the topology and weight distribution of neural networks. *IEEE Transactions on Neural Networks*, 5(1):39–53, 1994.

[111] A. Maren, C. Harston, and R. Pap. *Handbook of Neural Computing Applications*. Academic Press, 1990.

[112] N. Matsumoto and E. Tazaki. Emergence of learning rule in neural networks using genetic programming combined with decision trees. In *Proceedings of the IEEE Conference on Systems, Man, and Cybernetics*, pages 1801–1805, 1998.

[113] E. Mayoraz and F. Aviolat. Constructive training methods for feedforward neural networks with binary weights. Research report RRR 34–95, RUTCOR - Rutgers University's Center for Operations Research, 1995.

[114] J. McDonnell and D. Waagen. Evolving neural network connectivity. In *Proceedings of IEEE International Conference on Neural Networks (ICNN)*, pages 863–868, San francisco, CA, USA, 1993.

[115] F. Menczer and D. Parisi. Evidence of hyperplanes in the genetic learning of neural networks. *Biological Cybernetics*, 66:283–289, 1992.

[116] M. Mézzard and J. P. Nadal. Learning in feedforward layered networks. the tiling algorithm. *Journal of Physics A: Mathematical and General*, 22:2191–2203, 1989.

[117] Z. Michalewicz. *Genetic Algorithms [plus] Data Structures = Evolution Programs*. Springer-Verlag, Berlin, 1994.

[118] G. Miller, P. Todd, and S. Hegde. Designing neural networks using genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms (ICGA)*, pages 379–384, 1989.

[119] M. Minsky and S. Papert. *Perceptron*. MIT Press, Cambridge, MA, 1988.

[120] M. Mitchell. *An introduction to genetic algorithms*. MIT Press, Cambridge, Massachusets, USA, 1996.

[121] D. Montana and L. Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of Eleventh International Joint Conference on Artificial Intelligence*, pages 762–767, San Mateo, CA, 1989. Morgan Kaufmann.

[122] D. Moriarty and Miikkulainen. Evolutionary neural networks for value ordering in constraint satisfaction problems. Technical Report AI94-218, Department of Computer Science, University of Texas at Austin, Austin, TX 78712, USA, May 1994.

[123] D. Moriarty and R. Miikkulainen. Evolving complex Othello strategies using marker-based genetic encoding of neural networks. Technical Report AI93–206, Department of Computer Science, The University of Texas, 1993.

[124] D. Moriarty and R. Miikkulainen. Efficient reinforcement learning through symbiotic evolution. In *Machine Learning*, volume 22, pages 11–33. 1996.

[125] P. Munro. Genetic search for optimal representations in neural networks. In *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms (ANNGA)*, pages 628–634, 1993.

[126] H. Murao and S. Kitamura. Evolution of locally defined learning rules and their coordination in feedforward neural networks. *Artificial Life and Robotics*, 1(2):89–94, 1997.

[127] T. Nagao, T. Agui, and H. Nagahashi. Structural evolution of neural networks having arbitrary connections by a genetic method. *IEICE Transactions on Information and Systems*, E7–6D(6):689–697, Jun. 1993.

[128] H. Nakayama, T. Iwata, and T. Yamauchi. Learning and structuring of neural networks using genetic algorithm and linear-programming. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, volume 3, Nagoya, Japan, 1993.

[129] K. Nakayama and M. Ohsugi. A simultaneous learning method for both activation functions and connection weights of multilayer neural networks. In *Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN)*, pages 2253–2257, 1998.

[130] S. Nolfi and Parisi D. Evolving artificial neural networks that develop in time. In F. Moran, A. Moreno, J. Merelo, and P. Chacon, editors, *Advances in Artificial Life: Proceedings of the Third European Conference on Artificial Life*, pages 353–367, Berlin Heidelberg, 1995. Springer Verlag.

[131] S. Nolfi and D. Parisi. Growing neural networks. Technical Report PCIA-95-15, Department of Cognitive Process and Artificial Intelligence, Institute of Psychology, National Research Counsil, Rome, Italy, Jun. 1991.

[132] S. Nolfi and D. Parisi. Genotypes for neural networks. In M. Arbib, editor, *The Handbook of Brain Theory and Neural Networks*, pages 431–434, Cambridge, Mass, 1995. MIT Press.

[133] S. Omatu. Stability of inverted pendulum by neuro-pid control with genetic algorithms. In *Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN)*, pages 2142–2145, 1998.

[134] L. Park and C. Park. Fast layer-by-layer training of the feedforward neural network classifier with genetic algorithm. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, volume 3, pages 2595–2598, Nagoya, Japan, Oct. 1993.

[135] D. Patterson. *Artificial Neural Networks, Theory and Applications*. Prentice Hall, 1996.

[136] C. Peck and Dhawan A. Genetic algorithms as global random search methods: An alternative perspective. *Evolutionary Computation*, 3(1):39–73, 1995.

[137] M. Pelillo and A. Fanelli. A method of pruning layered feed-forward neural networks. In *Proceedings of the International Workshop on Artificial Neural Network (IWANN)*, pages 179–184. Springer-Verlag, 1993.

[138] F. J. Pineda. Generalization of backpropagation to recurrent neural networks. *Physical Review Letters*, 59(19):2229–2232, Nov. 1987.

[139] Y. Pirez and D. Sarkar. Back-propagation algorithm with controlled oscillation of weights. In *Proceedings of the IEEE International Conference on Neural Networks(ICNN)*, pages 21–26, San Francisco, CA, 1993.

[140] R. Poli. Some steps towards a form of parallel distributed genetic programming. In *Proceedings of the First On-line Workshop on Soft Computing*, pages 290–295, Aug. 1996.

[141] R. Poli. Discovery of symbolic, neuron-symbolic and neural networks with parallel distributed genetic programming. In *Third International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA)*, 1997.

[142] R. Poli and W. Langdon. A new schema theory for genetic programming with one-point crossover and point mutation. In *Proceeding of the Second International Conference on Genetic Programming*, Jul.

[143] V. Porto. Neural-evolutionary systems. In E. Fiesler and R. Beale, editors, *Handbook of Neural Computation*, pages D2.1:1–D2.3:2. Oxford University Press, 1997.

[144] M. Potter. A genetic cascade-correlation learning algorithm. In *Proceedings of the International Workshop on Combination of Genetic Algorithms and Neural Networks (COGANN)*, pages 123–133. IEEE Computer Society Press, 1992.

[145] M. Potter and K. Jong. Evolving neural networks with collaborative species. In *Proceedings of Summer Computer Simulation Conference (SCGA95)*, Ottawa, Canada, Jun. 1995.

[146] P. Pratt. Evolving neural networks to control unstable dynamical systems. In A. Sebald and L. Fogel, editors, *Proceedings of the Third Annual Conference on Evolutionary Programming*, pages 191–204, San Diego, USA, Feb. 1994.

[147] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

[148] J. Pujol and R. Poli. Dual network representation applied to the evolution of neural controllers. In V. Porto, N. Saravanan, D. Waagen, and A. Eiben, editors, *Proceedings of the Seventh Annual Conference on Evolutionary Programming (EP)*, volume 1447 of *Lecture Notes in Computer Science*, pages 637–647. Springer-Verlag, 1998.

[149] J. Pujol and R. Poli. Efficient evolution of asymmetric recurrent neural networks using a pdgp-inspired two-dimensional representation. In W. Banzhaf, R. Poli, M. Schoenauer, and T. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming (EUROGP)*, volume 1391 of *Lecture Notes in Computer Science*, pages 130–141. Springer-Verlag, 1998.

[150] J. Pujol and R. Poli. Evolving neural networks using a dual representation with a combined crossover operator. In *Proceedings of the IEEE International Conference on Evolutionary Computation (ICEC)*, pages 416–421, 1998.

[151] J. Pujol and R. Poli. Evolving the topology and the weights of neural networks using a dual representation. *Special Issue on Evolutionary Learning of the Applied Intelligence Journal*, 8(1):73–84, Jan. 1998.

[152] N. Radcliffe. Genetic set recombination and its application to neural networks topology optimization. Technical Report EPCC-TR-91-21, University of Edinburgh, 1991.

[153] A. Radi and R. Poli. Discovery of optimal backpropagation learning rules using genetic programming. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, 1998.

[154] A. Radi and R. Poli. Genetic programming can discover fast and general learning rules for neural networks. In *Proceedings of the Third Annual on Genetic Programming Conference*, pages 314–322, Jul. 1998.

[155] R. Reed. Pruning algorithms: a survey. *IEEE Transactions on Neural Networks*, 4(5):740–747, 1993.

[156] C. Reeves and N. Steele. Neural networks and genetic algorithms. In D. James, editor, *Proceedings of the Eighth International Conference on Systems Engineering*, pages 166–173, Coventry, UK, Sept. 1991.

[157] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: the RPROP algorithm. In *Proceedings of the IEEE International Conference on Neural Networks (ICNN)*, San Francisco, CA, USA, 1993.

[158] S. Roberts and M. Turega. Evolving neural network structures: an evaluation of encoding techniques. In D. Pearson, N. Steele, and R. Albrecht, editors, *Artificial Neural Nets and Genetic Algorithms*. Springer-Verlag, 1995.

[159] A. Rooij, L. Jain, and R. Johnson. *Neural Network Training Using Genetic Algorithms*. World Scientific Publishing, P.O Box 128, Farrer Road, Singapore 912805, 1996.

[160] D. Rumelhart, G. Hinton, and R. Williams. Learning internal representations by error propagation. In J. McClelland, D. Rumelhart, and the PDP Research Group, editors, *Parallel Distributed Processsing: Explorations in the Microstructure of Cognition*, volume 1, pages 318–362. MIT Press, Cambridge, Massachusetts, 1986.

[161] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

[162] S. Saha and J. Christensen. Genetic design of sparse feedforward neural networks. *Information Sciences*, 79:191–200, 1994.

[163] N. Saravanan and D. Fogel. Evolving neural control systems. *IEEE Expert*, pages 23–27, Jun. 1995.

[164] J. Sarma and K. Jong. Generation gap methods. In T. Bäck, D. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, pages C2.7:1–C2.7:5. Oxford University Press, 1997.

[165] M. Sartori and P. Antsaklis. A simple method to derive bounds on the size and to train multilayer neural networks. *IEEE Transactions on Neural Networks*, 2(4):467–471, Jul. 1991.

[166] M. Sase, K. Matsui, and Y. Kosugi. Inter-generational architecture adaptation of neural networks. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, volume 3, pages 2941–2944, Nagoya, Japan, Oct. 1993.

[167] Y. Sato and T. Ochiai. 2-D genetic algorithms for determining neural network structure and weights. In *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming (EP95)*, San Diego, CA, USA, Mar. 1995.

[168] R. Schalkoff. *Artificial Neural Networks*. McGraw-Hill, 1997.

[169] W. Schiffmann, M. Joost, and R. Werner. Synthesis and performance analysis of multilayer neural networks. Technical Report 16/1992, Institute of Physics, University of Koblenz, Germany, 1992.

[170] W. Schiffmann, M. Joost, and R. Werner. Application of genetic algorithms to the construction of topologies for multilayer perceptrons. In *Proceedings of the International Conference on Neural Networks and Genetic Algorithms (ICNNGA)*, pages 675–682, 1993.

[171] A. Schneider. An adaptive system for generating neural networks using genetic algorithms. In *Proceedings of the International Society for Optical Engineering (SPIE)*, volume 2492, pages 284–292, 1995.

[172] M. Schoenauer and E. Ronald. Neuro-genetic truck backer-upper controller. In *First International Conference on Evolutionary Computation (ICEC)*, Orlando, USA, Jun. 1994.

[173] R. Schwaiger and H. Mayer. Genetic algorithms to create training data sets for artificial neural networks. In *Proceedings of the Third Nordic Workshop on Genetic Algorithms*, pages 153–162, 1997.

[174] R. Setiono. A penalty-function approach for pruning feedforward neural networks. *Neural Computation*, 9:185–204, 1997.

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

[176] J. Sietsma and R. Dow. Creating artificial neural networks that generalize. *Neural Networks*, 4:67–79, 1991.

[177] P. Simpson. Foundations of neural networks. In E. Sánchez-Sinencio and C. Lau, editors, *Artificial Neural Networks, Paradigms, Applications and Hardware Implementations*, pages 3–24. IEEE Press, 1992.

[178] F. Śmieja. Neural network constructive algorithms: Trading generalization for learning efficiency? *Circuits, System and Signal Processing*, 12(2):331–374, 1993.

[179] W. Spears, K. Jong, T. Bäck, D. Fogel, and H. Garis. An overview of evolutionary computation. In *Proceedings of the European Conference on Machine Learning*, 1993.

[180] S. Stepniewski and A. Keane. Topology design of feedforward meural networks. In *Parallel Problem Solving from Nature (PPSN4)*, pages 771–780, 1996.

[181] R. Sutton. Two problems with backpropagation and other steepest-descent learning procedures for networks. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, pages 823–831, 1989.

[182] G. Syswerda. Uniform crossover in genetic algorithms. *Proceedings of Third International Conference on Genetic Algorithms (ICGA)*, pages 2–9, 1989.

[183] K. Tang, C. Chan, K. Man, and S. Kwong. Genetic structure for NN topology and weights optimization. In *Proceedings of the International Conference on Genetic Algorithms in Engineering Systems: innovations and applications (GALESIA)*, pages 250–255. Sept. 1995.

[184] K. Tang, K. Man, S. Kwong, and Q. He. Genetic algorithms and their applications. *IEEE Signal Processing Magazine*, pages 22–37, Nov. 1996.

[185] J. Torreele. Temporal processing with recurrent networks: An evolutionary approach. In R. Belew and L. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms (ICGA)*, pages 555–561. Morgan Kaufmann, 1991.

[186] L. Tsinas and B. Dachwald. A combined neural and genetic learning algorithm. In *Proceedings of the first IEEE Conference on Evolutionary Computation (ICEC)*, volume 2, pages 750–752a, Orlando, FL, USA, Jun. 1994.

[187] E. Vonk, L. Jain, L. Veelenturf, and R. Hibbs. Integrating evolutionary computation with neural networks. In *Proceedings of the Elektronic Technology Directions to the Year 2000*, pages 137–143, Adelaide, Australia, May 1995. IEEE Computer Society Press.

[188] E. Vonk, L. Jain, L. Veelenturf, and R. Johnson. Automatic generation of a neural network architecture using evolutionary computation. In *Proceedings of the Elektronic Technology Directions to the Year 2000*, pages 144–149, Adelaide, Australia, May 1995. IEEE Computer Society Press.

[189] C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[190] A. Weigend, D. Rumelhart, and B. Huberman. Back-propagation, weight-elimination and time series prediction. In D. Touretzky, J. Elman, T. Sejnowski, and G. Hinton, editors, *Proceedings of the Connectionist Models Summer School*, pages 105–116, San Francisco, CA, 1990.

[191] D. Whitley. Genetic algorithms and neural networks. In J. Periaux and Winter G., editors, *Genetic Algorithms in Computer Science*. John Wiley and Sons Ltd, 1995.

[192] D. Whitley, S. Dominic, and R. Das. Genetic reinforcement learning with multilayered neural networks. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 562–569. Morgan Kaufmann, 1991.

[193] D. Whitley, S. Dominic, R. Das, and C. Anderson. Genetic reinforcement learning for neurocontrol problems. *Machine Learning*, 13:259–284, 1993.

[194] D. Whitley, F. Gruau, and L. Pyeatt. Cellular encoding applied to neurocontrol. In *Proceedings of Sixth International Conference on Genetic Algorithms*, pages 460–467. Morgan-kaufmann, 1995.

[195] D. Whitley and T. Hanson. Optimizing neural networks using faster more accurate genetic search. In J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms (ICGA)*, pages 391–396, San Mateo, 1989. Morgan Kaufmann.

[196] D. Whitley, T. Starkweather, and C. Bogart. Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel Computing*, 14-3:347–361, 1990.

[197] B. Widrow and M. Lehr. 30 years of adaptive neural networks: Perceptron, madaline, and backpropagation. In *Proceedings of IEEE*, volume 78, Sept. 1990.

[198] A. Wieland. Evolving controls for unstable systems. In D. Touretsky, J. Elman, T. Sejnowski, and G. Hinton, editors, *Proceedings of the Connectionist Models Summer School*, pages 91–102, San Mateo, CA, 1990. Morgan Kaufmann.

[199] R. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation 1*, 1:270–280, 1989.

[200] X. Yao. Evolutionary artificial neural networks. In A. Kent et al., editor, *Enclyclopedia of Computer Science and Technology*, volume 33, pages 137–170. Marcel Dekker Inc., New York, NY 10016, USA, 1995.

[201] X. Yao and Y. Shi. A preliminary study on designing artificial neural networks using co-evolution. In *Proceedings of the IEEE Singapore International Conference on Intelligent Control and Instrumentation*, pages 149–154, Jun. 1995.

[202] B. Zhang and H. Mühlenbein. Evolving optimal neural networks using genetic algorithms with Occam's razor. *Complex Systems*, 7(3):199–220, 1993.

[203] B. Zhang and H. Mühlenbein. Genetic programming of minimal neural nets using Occam's razor. In S. Forrest, editor, *Proceedings of the Fifth international conference on genetic algorithms (ICGA'93)*, pages 342–349. Morgan Kaufmann, 1993.

[204] B. Zhang, P. Ohm, and H. Mühlenbein. Evolutionary induction of sparse neural trees. *Evolutionary Computation*, 5(2):213–236, 1997.