ABSTRACT

Title of dissertation: Evolutionary Design of Artificial Neural Networks Using a Descriptive Encoding Language

Jae-Yoon Jung, Doctor of Philosophy, 2007

Dissertation directed by: Dr. James A. Reggia
Department of Computer Science

Automated design of artificial neural networks by evolutionary algorithms (neuroevolution) has generated much recent research both because successful approaches will facilitate wide-spread use of intelligent systems based on neural networks, and because it will shed light on our understanding of how "real" neural networks may have evolved. The main challenge in neuroevolution is that the search space of neural network architectures and their corresponding optimal weights can be high-dimensional and disparate, and therefore evolution may not discover an optimal network even if it exists.

In this dissertation, I present a high-level encoding language that can be used to restrict the general search space of neural networks, and implement a problem-independent design system based on this encoding language. I show that this encoding scheme works effectively in 1) describing the search space in which evolution occurs; 2) specifying the initial configuration and evolutionary parameters; and 3) generating the final neural networks resulting from the evolutionary process in a human-readable manner. Evolved networks for "n-partition problems" demonstrate that this approach can evolve high-performance network architectures, and show by example that a small parsimony factor in the fitness measure can lead to the

emergence of modular networks. Further, this approach is shown to work for encoding recurrent neural networks for a temporal sequence generation problem, and the tradeoffs between various recurrent network architectures are systematically compared via multi-objective optimization. Finally, it is shown that this system can be extended to address reinforcement learning problems by evolving architectures and connection weights in a hierarchical manner. Experimental results support the conclusion that hierarchical evolutionary approaches integrated in a system having a high-level descriptive encoding language can be useful in designing modular networks, including those that have recurrent connectivity.

Evolutionary Design of Artificial Neural Networks Using a Descriptive Encoding Language

by

Jae-Yoon Jung

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2007

Advisory Committee:

Professor James A. Reggia, Chair/Advisor
Professor Amy Weinberg
Professor Aravind Srinivasan
Professor Lise Getoor
Professor Kyu Yong Choi, Dean's Representative

© Copyright by

Jae-Yoon Jung

2007

Acknowledgments

Most of all, I deeply thank my advisor, James A. Reggia, for his constant support, encouragement and guidance. He has been a great source of inspiration and a role model to me, and none of my work would have been possible without his years of painstaking efforts to teach me how to set up, examine, and present research ideas.

I am also grateful to professor Amy Weinberg, Aravind Srinivasan, Lise Getoor, and Kyu Yong Choi for serving on my thesis committee and for suggesting thoughtful comments and future directions.

Working within my research group has been a fortunate experience. I would like to thank all my group members, especially Dr. Shaun Gittens, Dr. Reiner Schulz, Matt Radio, Dr. Scott Weems, Dr. Alejandro Rodriguez, Dr. Alexander Grushin, Ransom Winder, Grecia Lapizco-Encinas, Tim Chabuk, and Charles Martin for giving their valuable comments and interesting ideas on my work.

I am also thankful to my Korean colleagues in the department. In particular, I am grateful to Dr. Hyun-Mo Kang, Il-Chul Yoon, Min-Kyoung Cho, Suk-Hyun Song, Sung-Woo Park, and In-Seok Choi for their help and support.

Finally, I would like to thank my parents for their everlasting encouragement and support. My wife, Phil-Hyoun and my son, Hyun-Woo have always been my greatest supporters and a source of wonderful joy. This work is dedicated to them.

Table of Contents

List of Tables v

List of Figures vi

1 Introduction 1
- 1.1 Motivations 1
- 1.2 Research Goals 5
- 1.3 Contributions and Thesis Organization 7

2 Background and Related Work 10
- 2.1 Evolutionary Computation 10
  - 2.1.1 Genetic Algorithms 12
  - 2.1.2 Genetic Programming 12
  - 2.1.3 Evolution Strategies 13
  - 2.1.4 Evolutionary Programming 14
- 2.2 Neuroevolution 14
- 2.3 Developmental Encoding Methods 15
- 2.4 Modularity in Neural Networks 17

3 Descriptive Encoding 19
- 3.1 Modular Design Principle 19
- 3.2 Explicitly Incorporating Domain Knowledge 26
- 3.3 Description File Examples 26
  - 3.3.1 Simple Error-Backpropagation Model 29
  - 3.3.2 Self-Organizing Map (SOM) Model 31
  - 3.3.3 Asymmetric Multi-Modular Model 34
- 3.4 Language Specification 34
  - 3.4.1 Layer 35
  - 3.4.2 Network 39
- 3.5 Encoding Properties 40
- 3.6 Neuroevolutionary Process 42
  - 3.6.1 Development and Learning Stage 42
  - 3.6.2 Evolutionary Process 45
    - 3.6.2.1 Fitness Evaluation 45
    - 3.6.2.2 Selection Process 45
    - 3.6.2.3 Mutation 46
    - 3.6.2.4 Topology Preserving Crossover 48

4 Module Formation in a Feedforward Network 49
- 4.1 Introduction 49
- 4.2 Encoding Details 51
- 4.3 The Evolutionary Procedure 53
- 4.4 Results of the Evolutionary Process 56

4.5 Discussion 61

5 Learning Word Pronunciations Using Recurrent Networks 63
5.1 Introduction 63
5.2 Encoding Details 66
5.3 Multi-objective Optimization 68
5.4 Experimental Results 69
5.5 Discussion 74

6 Evolving an Autonomous Agent 76
6.1 Introduction 76
6.2 Simulation Environment 80
6.3 Encoding Details 87
6.4 Experimental Results 91
6.4.1 Nested Evolution Strategy vs. Gaussian Mutation 95
6.4.2 Island Model vs. Single Population 96
6.4.3 Crossover vs. Mutation 97
6.5 Discussion 98

7 Disscussion 99
7.1 Contributions Revisited 99
7.2 Future Directions 101
References 104

List of Tables

2.1 Comparison of Typical Properties of Some Well-Known Evolutionary Algorithms.* 12

3.1 Example Module/Layer Properties 21

3.2 Default Values of Properties 36

4.1 Training Data for a 2-Partition Problem 50

4.2 Parallel n-Partition Problem Results 60

5.1 Representative Results for the Phoneme Sequence Generation Problem 73

6.1 Predetermined Properties for Predators and Agent. 82

6.2 Summarized Results with Compared Systems 95

List of Figures

3.1 (a) The first part of a description file specifies the network structure in a top-down hierarchical fashion (other information in the description file, such as details of the evolutionary process, is not shown in this example). (b) A schematic illustration of the corresponding class of recurrent networks that are described in (a). Question marks indicate architecture aspects that are considered evolvable. (c) Part of the tree-like structure corresponding to the genotype depicted in (a). Each rectangle, oval, and hexagon designates a network, layer, and property, respectively. This latter data structure provides the genetic material that is operated on by genetic programming algorithms. . . 23

3.2 A simple asymmetric neural network. (a) Initial description of the network, illustrating the language being developed in this research. (b) Each circle represents a network node, and each directed line shows a connection and its direction. Lateral connections are indicated by dotted lines. (c) Conceptual, hierarchical view of this network as a tree structure. Sequence and parallel structure of the different layers is indicated by the shaded blocks. 27

3.3 Error back propagation network with a hidden layer whose size is evolvable. (a) Initial network description. Some properties are not specified to keep the description small for illustrative purposes. The SIZE property of the hidden layer is evolvable. Indispensable parts are in bold type. (b) One possible realization or instantiation of this description in (a). The number of nodes in the hidden layer (here it is three) is randomly selected during the interpretation step. 30

3.4 A self-organizing map (SOM) example, adapted from [Levitan and Reggia, 2000], illustrating how even asymmetric unsupervised learning models can readily be represented by descriptive encoding language. (a) Two-dimensional layers with the unsupervised activation rule (SOM) is specified. (b) One possible realization or instantiation of the description. Each vertex in the grids denotes a network node. Example connections between input layers are illustrated by arrows. Note that the two map (upper) layers are asymmetric. 32

3.5 A complex multi-modular model, adapted from [Reggia et al., 2001a]. (a) Two blocks (left_brain, right_brain) in the middle structure may have different sizes. The INIT statements declare that these bilateral connections are fixed. (b) Conceptual organization of the network. Each arrow illustrates (a group of) connections forming a pathway, and the bilateral connections are represented by a shaded arrow. . . . 33

3.6 The iterative three-step development, learning, and evolution procedure used in this research. The input description file (upper left) is a human-written specification of the class of neural networks to be evolved (the space to be searched) by the evolutionary process, as outlined in the preceding sections of this chapter. The output file (lower right) is a human-readable specification of the best specific networks obtained that is described using the same encoding language. 43

4.1 (a) The initial network description and (b) a sketch of the space of networks to be searched for the 2-partition problem of Table 4.1. . . . 52

4.2 (a),(b) Examples of neural network architectures randomly created from the description in Figure 4.1a during initialization. Input and output layers are the same, but the number of hidden layers and their connections are quite different and specific now. Arrows indicate pathways that are sets of connections between layers (i.e., not individual node-to-node connections). (c) The chromosome description of the network illustrated in (a), as it would be written in our descriptive language. This is *not* a description file written by the user, but is automatically generated from that description file. Note that no EVOLVE attributes are present, for example. (d) Top part of the tree-like structure of the genotype in (c), making it directly usable by GP operators. Each rectangle designates a layer. 54

4.3 Typical network architectures found during evolution for the 2-partition problem are depicted. Dotted lines show connectivity with near-zero weights. (a) Final output description file having two independent pathways. (b) Conceptual network architecture described by (a). (c) Dual pathway network without direct input-to-output connections. Implicit hidden sub-layers are indicated by dotted ovals. 57

4.4 A typical example of a final evolved network for the 5 XOR partition problem. (a) Initial network description. Properties to be EVOLVEd are in bold font. (b) Final description file produced as output by the system. All EVOLVE properties have been replaced by the specific choices in bold font. Only SIZE and CONNECT properties are shown. (c) Depicted network architecture. Connections that have near-zero weights are pruned. 59

5.1 The Elman (a) and Jordan (b) network architectures shown here are widely used in neural network applications because of their simplicity, effectiveness, and efficiency. Dotted lines show the backward/recurrent one-to-one connections that essentially represent a copying of the output at one time step to a delay layer that serves as input at the next time step. 64

5.2 (a) The network description file for the phoneme sequence generation task. FWD, delayH:1, and delayO:1 mean to make a connection to the next layer in the same network block, to the first layer in the delayH network block, and to the first layer in the delayO network block, respectively. If such a block does not exist, the corresponding connectivity properties are ignored. The evolvable properties are in bold font. (b) A schematic illustration of the space of neural network architectures corresponding to the description file in (a) that are to be searched for the phoneme sequence generation problem. Dotted lines designate non-trainable, one-to-one feedback connections; solid lines indicate weighted, fully connected pathways trained by error backpropagation. Note that the Elman and Jordan networks of Figure 5.1 are included within this space as special cases. 67

5.3 (a) The performance/weights result of networks from all final generations are depicted. Each point represents one network architecture's values averaged over all evolutionary runs (most points are not labeled). The points on the solid line represent the Pareto-optimal set, and the labels on some of these latter points designate the type of network that they represent. For example, label Hh_3Oh_1 means that the network represented by that node has both hidden (H) and output (O) delay layers, while there are three hidden and one output delay layers, in both cases connected to the hidden (h) layer (see text). (b)-(d) Example of evolved network architectures and their corresponding labels. Evolved layers are shown in bold ovals. 70

5.4 The final network description of (a) an Elman network with single delay (labeled "Hh_1" in Figure 5.3a) and (b) a Jordan-like network with double delays (labeled "Oh_2"). Only SIZE and CONNECT properties are shown. The evolved properties (including the number of layers) are in bold font. (c) An illustration of the Jordan network specified in (b). Dotted lines designate fixed, one-to-one connections. 72

6.1 The iterative procedures used in this problem. The simulation environment is not considered as a part of the evolutionary system because the environment is problem-dependent. 78

6.2 An initial configuration of the simulation environment with 10 predators and 50 food sites. Each circle around a predator approximately indicates the distance that the predator can "see" (radius 7.0). The agent is located in the center of the environment when the simulation begins, but is not depicted in this figure. In order to get a food resource, the agent should reduce its velocity (to < 0.5) and collide against the food, which is shown as dots each with a radius of 0.5. . . 81

6.3 An agent's energy consumption model for each gait type and velocity. A fixed energy unit per time step (basal metabolic rate) is added when calculating the exact amount of energy consumption. 84

6.4 (a) the network description file used for evolving the network controller; (b) a sketch of the space of neural network architectures corresponding to the description file in (a). See text for details 87

6.5 The evolutionary parameter portion of the description file for this problem. 89

6.6 Average fitness of the best evolved agents compared with a non-adaptive, rule-based "control" agent as described earlier in the text. For each simulation configuration, the fitness value of the best network was average over ten independent runs. Part (a) shows the results with varying numbers of food sites while the number of predators was hold fixed at ten. In (b) the number of predators varied from zero to 30 while the number of food site was hold fixed at 100. . . . 92

6.7 (a) An example of a resultant network description file when the simulation configuration consisted of 10 predators and 100 food sites. The evolved properties are in bold font. (b) Depicted network architecture. 93

6.8 Fitness of best network controllers over varying number of food sites. The number of predators is fixed at 10. 94

6.9 Fitness of best network controllers over varying number of predators. The number of food sites is fixed at 100. 94

6.10 Average network size in terms of the number of connection weights, plotted versus the generation of an evolutionary process 96

Chapter 1

Introduction

1.1 Motivations

Most development of neural networks today is based upon manual design. A person knowlegeable about the specific application area specifies a network architecture and activation dynamics, and then selects a learning method to train the network via connection weight changes. While there non-evolutionary methods exist for automatic incremental network construction [Mehrotra et al., 1997], these generally presume a specific architecture, do not automatically discover network modules appropriate for specific training data, and have not enjoyed widespread use. This state of affairs is perhaps not surprising, given that the general space of possible neural networks is so large and complex that automatically searching it for an optimal network architecture may in general be computationally intractable, or at least impractical for complex applications [Blum and Rivest, 1992; Miller et al., 1989].

Neuroevolution refers to the design of artificial neural networks using evolutionary algorithms, and it has attracted much recent research both because successful approaches will facilitate wide-spread use of intelligent systems based on artificial neural networks, and because it will shed light on our understanding of how "real" neural networks may have evolved [Grushin and Reggia, 2005; Shkuro and Reggia, 2003]. Recent successes using evolutionary computation methods as

design/creativity tools in electronics, architecture, music, robotics, and other fields [Banzhf et al., 1997; Bentely and Corne, 2001; Koza et al., 1999] suggest that creative evolutionary systems could have a major impact on the effectiveness and efficiency of designing neural networks. This hypothesis is supported by an increasing number of neuroevolutionary methods that search through a space of weights and/or architectures without substantial human intervention, trying to obtain an optimal network for a given task (reviewed in [Balakrishnan and Honavar, 2001; Ruppin, 2002; Saravanan and Fogel, 1995; Yao, 1999]).

The main challenge in neuroevolution is that each problem may require evolving a unique neural network architecture and corresponding weight values, and the search spaces of architectures and weights are disparate and may be high-dimensional. In the earliest stages of neuroevolution research, a fixed network architecture was pre-selected and the evolutionary process searches the space of connection weights for this fixed network architecture (e.g., [Dill and Deer, 1991; De Garis, 1991; Montana and Davis, 1990; Srinivas and Patnaik, 1991]). This approach is partially supported by [Cybenko, 1989], claiming that a fully connected neural network with enough hidden nodes can approximate any continuous function in theory. However, evolving neural network architectures (structures) is still required because a small modification in a network architecture, deleting a node for example, may cause drastic changes in the search space of weight values, and choosing an optimal architecture for a given problem a priori is not possible in general.

The most important issue in evolving architectures is how to encode network structures efficiently. In a direct encoding scheme, all connection information is ex-

plicitly specified in a matrix format [Miller et al., 1989; Mitchell, 1996]. It is simple and all possible network architectures within a fixed matrix size can be represented. However, for large scale neural networks, searching for the optimal architecture with a direct encoding scheme can be impractical since the size of the space increases exponentially with the network size. On the other hand, developmental approaches are based on a genotype that specifies how to "grow" the neural network (the phenotype) rather than a direct encoding of its structure. In effect, the genome represents a "grammar" or set of rules that is used in a generative fashion. Example developmental methods include graph generation grammars [Kitano, 1994], cellular encoding [Gruau, 1995; Gruau et al., 1996], edge encoding [Luke and Spector, 1996], and attribute grammars [Hussain and Browse, 1998]. Developmental approaches are especially desirable for large neural networks where directly encoding a network's architecture is difficult. A grammatical representation can address this issue by efficiently capturing regularities or patterns in a network's structure, thereby providing a compact genome that is more effectively manipulated by an evolutionary process.

Modular architectures have also been the focus of a few successful neuroevolutionary models. Work in this area has been inspired in part by recognition that biological nervous systems are modular. For example, the vertebrate cerebral cortex is composed of cortical columns (small modules) that are in turn components of functional regions/areas (large modules) that collectively form the cerebral cortex [Mountcastle, 1998]. While many aspects of the evolution of biological modularity are not well understood, it is believed that, at a minimum, modularity contributes to adaptability [Schlosser and Wagner, 2004; Wagner, 1995]. Designing artificial neural

networks using a modular approach means that the designer works primarily with high-level multi-neuron modules/layers and their interconnections rather than with individual neurons/nodes and their connections. Combining modular design with developmental encodings is potentially valuable because it provides for compact, efficient specification of large networks, breaks the design process into manageable parts, allows the emergence of functionally-specialized components that do not interfere with each other, and supports scalability [Caelli et al., 1999]. Some recent studies have also provided evidence that modularity can improve neural network performance for specific tasks [Calabretta et al., 2000; Franco and Cannas, 2001; Schlessinger et al., 2006]. These results are encouraging, but much past work evolving modular neural networks has significantly restricted the range of architectures involved, typically by evolving networks composed of preset modules. For example, a common approach has been to use fixed, pre-designed modules representing cortical columns whose inter-modular connections are trained using Hebbian learning (e.g., [Cho and Shimohara, 1998; Happel and Murre, 1994]).

To summarize, neuroevolution requires a compact encoding scheme that helps to reduce the search space, enables the design of large scale neural networks, and incorporates biologically plausible modular network specification. This dissertation introduces such an encoding scheme and an application-independent neuroevolution system based on this encoding.

1.2 Research Goals

The central goal of the research described here was to develop and study a new neuro-evolutionary encoding scheme based on a human-readable descriptive language. Its hierarchical network description helps choose the level of sophistication in design of the network architecture and parameters, and explicitly indicates constraints on the desired neural networks that restrict the search space and thus make the evolutionary process more efficient. The use of a grammar-based, top-down description in this scheme increases the readability of represented neural networks. Further, encoded features span almost all aspects of recurrent neural networks including learning rules and activation dynamics, so that this system can be used as a general framework for neuroevolution. Within this context, the specific research objectives were as follows:

- Create an application-independent high-level descriptive language that is both human-readable and can serve as a genome for evolving neural networks.

- Implement a software environment based on this language and demonstrate that it is effective in evolving a range of neural network architectures and their connection weights.

- Examine the combination of evolutionary methods as a global search operator, and local tuning algorithms based on weight changes, extending past work in this area to contexts where genetic operations act on activation/learning rules and parameters as well as network architecture.

- Identify conditions under which modular neural networks are favored during evolution. The initial hypothesis was that when a problem size is large, has localized correlations in input data (or between input and output data), and there is a cost associated with larger networks and/or more connections, then multi-modular networks induced by minimizing costs while maximizing performance will prove to be more effective and fit than monolithic networks.

- Examine which types of recurrent neural network architectures are beneficial under which circumstances.

- Incorporate evolution of weight values in order to use this software environment as a framework for addressing machine learning problems in general.

If these objectives are achieved, this should not only make the evolutionary process more efficient for existing researchers in evolutionary computation, but should also broaden the range of people who can conveniently evolve neural networks to neural modelers in general, and to even neuroscientists whose primary interest is to use evolutionary methods to solve their specific application problems. Further, the same language is used to describe the initial specification, the chromosomes, and the final resultant networks, making all network descriptions human-readable. The top-down, hierarchical representation makes it easier to understand the details of the networks, overcoming one of the drawbacks in the low-level approaches (e.g., [Gruau et al., 1996; Luke and Spector, 1996; Hornby, 2004]) that the representations are very hard to follow and analyze since they are organized in a bottom-up fashion. This approach is analogous to the abstraction process used in contemporary

software engineering, in the sense that users write a text file specifying the problem to solve using a high-level language. The system then parses this file and searches for a solution within the designated search space, and finally produces the results as another human readable text file. The claim is that this approach has the potential to facilitate automated design of large scale neural networks covering a wide range of problem domains, not only because of its encoding efficiency, but also because it increases human readability and understandability of the initial search space specification and the final evolved networks (i.e., just as a contemporary high-level programming language such as C or Java increases software development productivity relative to using assembly language, I believe that this high-level language could increase neural network design productivity relative to low-level approaches). Finally, this approach smoothly integrates weight learning / evolution with the evolutionary process of architectures, permitting adaptation to occur prior to network fitness assessment. Separating evolution of the architecture from subsequent adaptation of connection weights (either via traditional neural network learning algorithms or another evolutionary process) is consistent with both biological events and with evidence that such separation can produce better artificial neural networks than trying to evolve both architectures and weights together [Yao, 1999; Ferdinando et al., 2001].

1.3 Contributions and Thesis Organization

In this dissertation, I make a number of contributions in neuroevolution.

- I introduce a novel encoding scheme using a high-level description language. The tree structured encoding scheme is designed to be both amenable to genetic programming operators, and simultaneously to be human readable. I show that this approach is scalable and can be mapped into a valid set of recurrent phenotype networks (Chapter 3).

- I implement a problem-independent system that evolves neural network architectures as well as adapting their connection weights, based on the descriptive encoding scheme. The use of a description file provides a systematic, non-procedural methodology for specifying the search space and evolution parameters, and the same language that is used for the network description is used to produce a human readable final network description.

- With n-partition problems (Chapter 4), I demonstrate that the descriptive encoding can be effectively applied to problems with increasing complexity. The evolved networks support the hypothesis that modular design would be beneficial in terms of network performance, and shows how balancing high performance vs. low cost trade-offs encourages evolution of modular networks. The evolved results also justify why we need to search the space of architectures, by comparing with fixed structure networks.

- In Chapter 5, I apply the descriptive encoding to a problem of temporal sequence generation, which requires the evolution of recurrent neural network architectures. Various recurrent neural network architectures are systematically compared via a multi-objective optimization method [Coello Coello, 2002],

including two well-known recurrent architectures.

- I show that the descriptive encoding system can be effectively applied to addressing reinforcement learning problems where gradient-descent information is not available (Chapter 6). A separate evolutionary process based on an evolution strategy is adopted for evolution of connection weights. The results show that evolution can discover efficient solutions for real-valued reinforcement learning problems, which can be very difficult for typical reinforcement learning algorithms. The hierarchical evolution of architectures and weights in the descriptive encoding system is a first step towards a general framework for evolving neural networks, and for solving machine learning problems in general.

The contributions of this study and some future research directions are summarized in Chapter 7. I first explain the background of this research and related work in the next chapter.

Chapter 2

Background and Related Work

In this chapter, common features of several evolutionary computation algorithms as well as their differences are briefly discussed first. The evolution of neural networks using evolutionary computation algorithms are reviewed next, and then developmental encoding schemes that describe ways of growing target networks are explained, followed by a short review of modularity in neural networks.

2.1 Evolutionary Computation

Evolutionary computation refers to a set of general-purpose search algorithms inspired by natural selection and evolution in the real world [Ashlock, 2006; Bäck and Schwefel, 1993; Fogel, 1995; Jong, 2006]. These algorithms use a population of individuals that represent potential solutions for a given problem. For each generation, the environment (via a fitness function) indicates which individuals are more fit than others, and the next population is produced from these selected individuals via mutation, recombination, and/or other genetic operations. An outline of an example evolutionary algorithm is as follows (adapted from [Bäck, 1994]):

1. Set generation $t = 0$.

2. Create the initial population, $P(t)$.

3. Evaluate the fitness of each individual in $P(t)$.

4. While ending condition $end(P(t), t)$ is not satisfied, do

   a. Calculate $P'(t) = recombination(P(t))$, and $P''(t) = mutation(P'(t))$.

   b. Evaluate $P''(t)$.

   c. Q = set of individuals chosen from the original population, $P(t)$.

   d. Reproduce next population, $P(t+1) = selection(P''(t) \cup Q)$,

   e. Set $t = t + 1$.

Genetic algorithms [Goldberg, 1989b; Holland, 1975; Yuen and Chenung, 2006], genetic programming [Koza, 1992] evolution strategies [Schwefel, 1981], and evolutionary programming [Fogel et al., 1966; Fogel, 1991] are prominent instances of this approach. Each algorithm, in its canonical form, has its own representation schemes and genetic operators. For example, genetic algorithms use binary strings for a chromosome, while others use tree-structured programs (genetic programming), real vectors (evolution strategies), or finite state machines (evolutionary programming). However, these different approaches to evolutionary computation now share many features as the application of evolutionary computation has become more wide spread, and the research groups associated with each evolutionary approach have communicated with each other more effectively since the early 1990's. A comparison of these algorithms is summarized in Table 2.1.

Table 2.1: Comparison of Typical Properties of Some Well-Known Evolutionary Algorithms.*

	GA	GP	ES	EP
Chromosome	binary string	tree-structured program	real vector + strategy parameters	finite state machine
Mutation	reverse 1-bit	replace random subtree	perturb strategy param. then mutate target vector	node, link operators
Recombination	crossover (primary)	subtree crossover (primary)	separate recombination on target vector and parameters	not used
Selection	probabilistic	varies	deterministic	deterministic

* GA = genetic algorithm, GP = genetic programming, ES = evolution strategies, EP = evolutionary programming

2.1.1 Genetic Algorithms

In the canonical genetic algorithm, potential solutions are represented by fixed-length binary strings. Crossover is the primary operator for producing variations among chromosomes, while mutation is just inverting bits with a small probability ($\approx 10^{-3}$ per bit) and is often considered as a *background* operator. The selection method of the canonical genetic algorithm is fitness proportionate; the relative fitness value of each individual defines the probability of selecting that individual for the next generation. However, many implementations of current genetic algorithms have adopted various alternatives, including tournament selection.

2.1.2 Genetic Programming

Genetic programming is an important approach to evolutionary computation, and is most directly related to this work. The main objective of genetic programming is to evolve an optimal computer program that can solve a given problem, so the population consists of programs written in lisp or other languages, instead

of binary strings. Note that programs are often represented as tree structures, on which evolution operators are performed. A problem-specific set of functions and terminals are selected by the designer, and then the initial population is created by random composition of trees of these functions and terminals. The fitness measure is usually the correctness of the solution produced by each program individual, and tree crossover is favored as the major reproduction method over mutation (e.g., in early work [Koza, 1994], Koza didn't use mutation at all), although this has been a controversial subject. While genetic programming has been applied successfully to many application areas like pattern recognition, signal processing, and natural language processing (see [Banzhf et al., 1997, Chap. 12]), only a relatively limited number of results related to neuroevolution have been proposed.

2.1.3 Evolution Strategies

Real vector optimization is the general goal of evolution strategies. Each individual consists of a target vector and strategy parameters (i.e., variances and/or covariances) to perturb the target. Only the target vector is involved in the fitness calculation, but both of the target vector and strategy parameters may evolve throughout the evolutionary process, which enables *self-adaptation* to complex fitness surfaces. In this research, a variant of an evolution strategy algorithm has been adopted in order to evolve connection weights, which will be explained in detail in Chapter 6.

2.1.4 Evolutionary Programming

Although the original intention of evolutionary programming was to evolve finite state machines to predict future events, contemporary extended evolutionary programming has become similar to evolution strategies in that it adopts a real vector representation as well as mutation (strategy) parameters. Recombination operators are not used in typical evolutionary programming implementations since mutation operators are considered to be enough to create possible changes between generations [Porto, 1997], which would be an opposite perspective of the building block hypothesis in genetic algorithms. As with evolution strategies, the original selection method was deterministic; only the best fit individuals are carried forward into the next population (i.e., *elitism*).

2.2 Neuroevolution

There have been several different motivations for evolving neural networks. Often, designing a neural network requires knowledge of the problem domain. But in many real applications such knowledge is noisy or even unavailable, which usually leads to a repetitious trial-and-error approach. Back-propagation [Rumelhart et al., 1986] and other gradient descent algorithms (e.g., [Møller, 1993]) are used to find a global minimum in an error space, but they may get stuck in a local minimum, and require the error space to be differentiable [Sutton, 1986]. However, evolutionary algorithms do not require gradient information so that they can search virtually any kind of error space. Finally, neuroevolution can adapt to a dynamic environment

(i.e., changes in the environment) as well as the environment itself. This property is similar to evolution in the real world, so various issues in neural science can be computationally simulated and verified with neuroevolution.

2.3 Developmental Encoding Methods

Developmental encoding methods are a variant of indirect encoding schemes which use a predefined grammar. They are called developmental because the information (e.g., a set of ordered rules and parameters) stored in the genotype describes a way of "growing" the phenotype; the actual neural network is developed, starting from a basic unit, according to the given grammar. The virtue of this method is that a large network can be encoded in a compact and structured genotype.

In the seminal work in this field [Kitano, 1990, 1994], Kitano encoded a developmental rule as a *graph generation grammar* taking the form of a matrix, rather than specifying the whole network architecture. Constructing a network connectivity matrix starts from the initial start symbol in a chromosome. Rules are then applied to replace each non-terminal symbol in the chromosome with a 2x2 matrix of symbols, until there are all terminal symbols. Therefore the evolutionary process searches for the best set of rules that would generate an optimal network architecture for a given task. Kitano claimed that this approach is more compact and efficient than a direct encoding scheme, but this has been controversial as some subsequent results contradicted his claim [Siddiqi and Lucas, 1998].

Gruau also proposed a grammatical encoding approach, *cellular encoding*,

where each rule defines transformation or modification of a cell, and rules constitute a tree structure such that the order of execution for each rule is specified [Gruau, 1994, 1995]. It is a compact encoding scheme in the sense that all rules in the chromosome participate in the development of a network, and any kind of network architecture can be represented with this approach. While a variant of this approach has been successful with some real weight applications [Gruau et al., 1996; Whitley et al., 1995], weight assignment seems to be inefficient since rules are applied to each cell in general, not to a group of cells or to the whole network.

Hussain and Browse suggested another grammar-based representation scheme, *network generating attribute grammar encoding*, or NGAGE [Hussain and Browse, 1998, 2000]. While Kitano's approach keeps the complete grammar itself in a chromosome and makes it evolve, this method uses a fixed set of grammar rules but maintains a possible subset of this grammar in each chromosome. Since all legal productions can be represented as a parse tree, genetic operations in genetic programming (e.g., tree-crossover, and subtree mutation) can be directly applied to this model. Although this is somewhat similar to my approach in that it has top-down specification of network architecture, this approach can only represent network topologies with a limited range (e.g., all layer-to-layer connectivity is assumed to be fully-connected), and there is no way to specify other various network aspects like learning parameters.

2.4 Modularity in Neural Networks

A neural network is said to be *modular* if the whole system can be divided into several modules that can be distinguished from each other, from either an architectural or functional perspective. Typical modular network implementations require problem domain knowledge to establish the architecture of the network. Neural network designers typically set and fix the number of modules, connectivity and/or activation functions according to the application domain before the training process begins. Some modules may have a specific, predefined function upon the whole network (e.g., in [Jacobs et al., 1991], the gate module controls the probability distribution to select which expert module is the winner for the current input pattern), while the role of other modules may be identified later during the training stage. Model-based neural networks [Caelli et al., 1993], adaptive mixture of experts [Jacobs et al., 1991], decision-based neural networks [Kung and Taur, 1995], and some models of large scale brain structure [Levitan and Reggia, 2000] are included in this group. Other researchers have adopted various search algorithms, including evolutionary programming (e.g., [Cho, 1997]), to generate the architecture and constraints of the network, avoiding the user-defined network topology. The input data set is analyzed and partitioned into an appropriate number of clusters, then modules and the topology are created and modified accordingly during training. The structure and function of modules are typically identical or self-similar, while each module operates on different patterns or clusters of the input data. Examples of this approach include networks of networks [Guan et al., 1997] and self-partitioning

neural networks [Ranganath et al., 1995].

Since the network structure and parameters of the above approaches are tightly coupled with domain knowledge and the input data (e.g., [Bonissone et al., 2006]), they may not generalize well even though they are fast and efficient in the given domain with the specific data set. On the other hand, blind search without any domain information could easily turn out to be intractable, especially in the design of large-scale networks. To balance between these two extreme cases, an application-independent, hierarchical network description language is introduced in Chapter 3. Users can describe restrictions on the modular structure of the networks based on the domain knowledge of their own problem, and also specify which parts of the network will evolve during the evolution process.

Chapter 3

Descriptive Encoding

The encoding scheme introduced here is an extension of developmental encoding and module-based methods proposed in the past, and now formalized in a high-level language. I refer to this approach as a *descriptive encoding* since it enables users to describe the target space of neural networks that are to be considered in a natural, non-procedural and human-readable format. A user writes a text file like the ones shown later in this chapter to specify sets of modules (layers) with appropriate properties, their range of legal evolvable property values, and allowable inter-module connectivity ("pathways"). This input description does *not* specify individual neurons, connections, nor their weights.[1] The specification of legal property values affects the range of valid genetic operations. In other words, a description file specifies the configuration of the initial population and environment variables, and restricts the space to be searched by genetic operators during evolution.

3.1 Modular Design Principle

The basic unit of descriptive encoding is a module that is called a *layer*, which is defined as an array (currently either one-dimensional or two-dimensional) of net-

[1] Individual neurons, connections and weights *can* be specified by creating layers/modules containing single neurons, but this does not take advantage of the language's ability to compactly describe large scale networks.

work nodes that share common properties. In other words, individual neurons are not the atomic unit of evolution, but sets of neurons are. Modular, hierarchical structure is essential when the size of the resultant neural network is expected to be large, since monolithic networks can behave irregularly as the network size becomes larger. Moreover, there is substantial evidence that a basic underlying neurobiological unit of cognitive function is a region (layer), e.g., in cerebral cortex [Mountcastle, 1998], which strengthens the argument that hierarchical structure of layers should be the base architecture of any functional unit. Parametric encoding can also reduce the complexity of a genotype when there is a regular pattern in the network features, and opens the possibility for users to specify a set of appropriate network features according to given problem requirements (see [Jung and Reggia, 2004b] for discussion).

The description of a layer/module starts with an identifier LAYER, which is followed by an optional layer name and a list of properties. Properties of a layer can be categorized into three groups: structure (e.g., BIAS, SIZE, NUM_LAYER), dynamics (e.g., ACT_RULE, ACT_INIT, ACT_MIN, ACT_MAX), and connectivity (e.g., CONNECT, CONNECT_RADIUS, CONNECT_INIT, LEARN_RULE). The order of declared properties in a layer description does not matter in general. Individual properties can be designated to be evolvable within some range, or to be fixed. Each property has its own default value for simplicity: if some properties are missing in the description file, they will be replaced with the default values during the initialization stage and considered as being constant throughout the evolutionary process (i.e., the chromosome is in fact more strict than the description;

Table 3.1: Example Module/Layer Properties

Property	What it Specifies About a Module/Layer
BIAS	whether to use bias units and their initial value ranges if so
SIZE	number of nodes in the current layer
NUM_LAYER	number of layers of this type
ACT_RULE	activation rule for nodes in the layer
ACT_INIT	initial activation value for nodes
ACT_MIN	minimum activation value
ACT_MAX	maximum activation value
CONNECT	direction (or target layer name) of connections starting from this layer
CONNECT_RADIUS	range of connectivity from 0.0 (one-to-one) to 1.0 (full connectivity)
CONNECT_INIT	initial (range of) weights in the current connections
LEARN_RULE	learning rule for the current connections

it requires all default, fixed, and evolvable properties to be present in some form). Layer properties used in this dissertation are illustrated in Table 3.1. The meaning of each property is fairly straightforward, but the [CONNECT_RADIUS r] property with $0.0 \leq r \leq 1.0$ needs more explanation. It defines the range of the connectivity from each node in a source layer to the nodes in a target layer. For example, if $r = 0.0$, each node in the source layer is connected to just a single node in the matching position of the target layer. If r is a positive fraction less than one, each source node connects to the matching destination layer node and its neighbor nodes out to a fraction r of the radius of the target layer; thus, if $r = 1.0$, the source and target layers are fully connected. While these connectivity properties are basically intended to specify connections between two layers (i.e., an inter-modular connection), intra-modular connections such as self-connectivity can also be designated using the same properties. For example, one can specify that each node in a layer named $layer1$ connects to itself by using a combination of [CONNECT $layer1$]

and [CONNECT_RADIUS 0.0]. The user can also change the default value of each property for a specific problem domain by declaring and modifying property values in the evolutionary part of a description file, which will be explained later.

Figure 3.1a illustrates part of an input description file written using descriptive encoding language for evolving a recurrent network (a full syntax for descriptive encoding language is given in the Appendix). Each semantic block, enclosed in brackets [$\cdots$], starts with a type identifier followed by an optional name and a list of properties about which the user is concerned in the given problem. A network may contain other (sub)networks and/or layers recursively, and a network type identifier (SEQUENCE, PARALLEL, or COLLECTION) indicates the conceptual arrangement of the subnetworks contained in this network. If a network module starts with the SEQUENCE identifier, the sub-networks contained in this module are considered to be arranged in a sequential manner (e.g., like a typical feed-forward neural network). Using the PARALLEL identifier declares that the sub-networks are to be arranged in parallel, and the COLLECTION identifier indicates that an arbitrary mixture of sequential and parallel layers may be used and evolved. The COLLECTION identifier is especially useful when there is little knowledge about the appropriate relationships between the layers being evolved. As described earlier, a layer is defined as a set (sometimes one or two dimensional, depending on the problem) of nodes that share similar properties, and it is the basic module of the network representation scheme. For example, the description in Figure 3.1a indicates that a sequence of four types of layers are to be used: input, hidden, output, and context layers, as pictured in Figure 3.1b. Properties fill in the details of the network

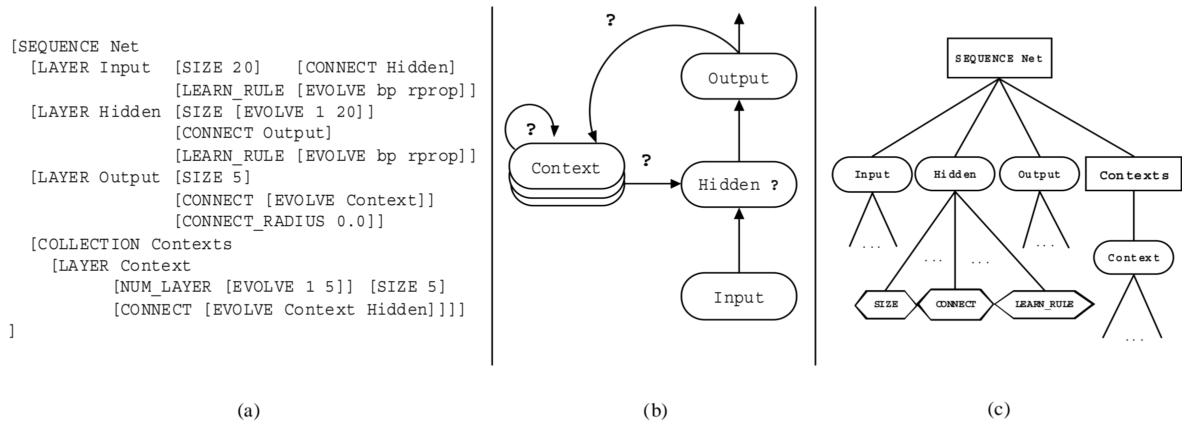


Figure 3.1: (a) The first part of a description file specifies the network structure in a top-down hierarchical fashion (other information in the description file, such as details of the evolutionary process, is not shown in this example). (b) A schematic illustration of the corresponding class of recurrent networks that are described in (a). Question marks indicate architecture aspects that are considered evolvable. (c) Part of the tree-like structure corresponding to the genotype depicted in (a). Each rectangle, oval, and hexagon designates a network, layer, and property, respectively. This latter data structure provides the genetic material that is operated on by genetic programming algorithms.

architecture (e.g., layer size and connectivity) and in general specify other network features including learning rules and activation dynamics.

Most previous neuroevolution research has focused a priori on some limited number of network features (e.g., network weights, number of nodes in the hidden layer) assuming that the other features are fixed. This situation plus the need to hand-code the evolutionary process of each specific application has impeded past neuroevolutionary models from being used more widely in different environments. In order to overcome this limitation, descriptive encoding allows users to decide which fixed properties are necessary to solve their problems, and which other factors should be evolved, from a set of supported properties that span many aspects of neural networks. Unspecified properties are replaced with default values and are treated as being fixed after initialization. So, for example, in the description of Figure 3.1a, the input layer has a fixed, user-assigned number of 20 nodes and is connected to the hidden layer, while the single hidden layer has an evolvable SIZE within the range 1 to 20 nodes. The EVOLVE attribute indicates that the hidden layer's size will be randomly selected initially and is to be modified within the specified range during the evolution process. Note that the learning rules to be used for connections originating from both input and hidden layers are also declared as an evolvable property (in this case, a choice between two variants of backpropagation). The description in Figure 3.1a also indicates that one to five context layers are to be included in the network; this is the main architectural aspect that is to be evolved in this example. These context layers are to be ordered arbitrarily, all contain five neurons, and they evolve to have zero or more inter-layer output connections to either other context layers

or to the hidden layer (Figure 3.1b). Finally, the output layer evolves to propagate its output to one or more of the context layers, where the CONNECT_RADIUS property defines one-to-one connectivity in this case. Since the number of layers in the context network may vary from one to five (i.e., LAYER context has an evolvable NUM_LAYER property), this output connectivity can be linked to any of these layers that were selected in a random manner during the evolution process. Figure 3.1b depicts the corresponding search space schematically for the description file of Figure 3.1a, and the details of each individual genotype (shown as question marks in the picture) will be assigned within this space at the initialization step and forced to remain within this space during the evolution process. Note that since the genotype structure is a tree, as shown in Figure 3.1c, fairly standard tree-manipulation genetic operators as used in genetic programming [Banzhf et al., 1997; Koza et al., 1999] can be easily applied to them with this approach.

The remainder of the description file consists of information about training and evolution processes (not shown in Figure 3.1a). A *training block* specifies the file name where training data is located and the maximum number of training epochs, when supervised learning methods are used for training connection weights. Default property values may also be designated here, like the learning rule (LEARN_RULE property) to be used. In other words, when a property value is specified in this block, the default value of that property is changed accordingly and affects all layers which have that property. An *evolution block* can also be present and specifies parameters affecting fitness criteria, selection method, type and rate of genetic operations, and other population information that will be illustrated later.

3.2 Explicitly Incorporating Domain Knowledge

When searching for an optimal neural network using evolutionary computation methods, a network designer usually wants to restrict the architecture, learning rules, etc. to some proper subset of all possible models. Thus, many problem specific constraints need to be applied in creating the initial population and maintaining it within a restricted subspace of the space of all neural networks. For example, the range of architectures and valid property values for each individual network in the initial population will depend upon the specific problem being addressed. While such constraints and initialization procedures have been treated implicitly in previous approaches, descriptive encoding scheme permits them to be described in a compact and explicit manner.

3.3 Description File Examples

Before considering the detailed evolution of neural networks, it is useful to examine an example of using the high-level descriptive encoding language to describe just a single, simple, fixed neural network architecture. This example has nothing to do with evolution; it is just intended to illustrate the language.

Figure 3.2b shows a simple example of a neural network in schematic form for which a description is sought. The network here is a typical feed-forward three layer, fully-connected neural network like those often used in error backpropagation, except two parallel subsets of hidden nodes occur (a left layer with one node, and a right layer with two nodes). There are asymmetric, lateral connections from

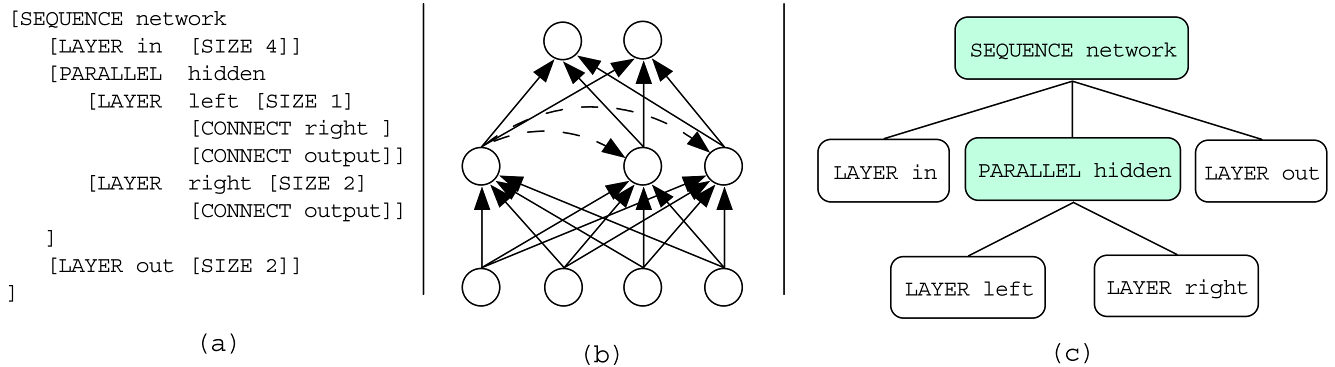


Figure 3.2: A simple asymmetric neural network. (a) Initial description of the network, illustrating the language being developed in this research. (b) Each circle represents a network node, and each directed line shows a connection and its direction. Lateral connections are indicated by dotted lines. (c) Conceptual, hierarchical view of this network as a tree structure. Sequence and parallel structure of the different layers is indicated by the shaded blocks.

the left layer to the right one (dotted arrows). This type of irregularly-structured, asymmetric networks is very hard to represent with other previous encoding schemes. However, as shown in Figure 3.2a, descriptive encoding scheme specifies this network architecture in a clear and succinct manner. The human-readable description here is written in a top-down fashion, and consists of nested statements of the form

[KEYWORD ... details ...].

Thus, at the top level, Figure 3.2a indicates that this network is a SEQUENCE of modules. Inside the SEQUENCE statement, it is indicated that the sequence of modules involved is a LAYER named *in* with four nodes, then two PARALLEL hidden layers, and finally a LAYER named *out* with two nodes. The two hidden LAYER's are named *left* (with one node) and *right* (two nodes). In the absence of other information, the SEQUENCE of modules is fully connected in only a forward direction by default (hence the forward connections in Figure 3.2b). The connections between the parallel hidden layers are indicated explicitly in Figure 3.2a by a CONNECT statement. Figure 3.2c depicts the conceptual, tree-structured hierarchy of this network. Although only the essential properties are shown for illustrative purposes in this example, the language can actually specify activation dynamics, learning rules, initialization methods, and temporal/spatial parameters as well as evolution parameters (e.g., which property of the network will be evolvable, population size, number of generation, and so on).

We now turn to considering a more detailed explanation of the encoding scheme and descriptive language in the following sections, first considering three examples.

In the first example of a typical error backpropagation model, the basic grammar and its properties will be explained. The second example is a simplified self-organizing map, which can illustrate another unique feature of the descriptive encoding scheme, its ability to incorporate different network models and learning paradigms (e.g., supervised and unsupervised) into one unified system. The third example is a more complicated bilateral connection model that is used to show evolvable layer properties and scalability.

3.3.1 Simple Error-Backpropagation Model

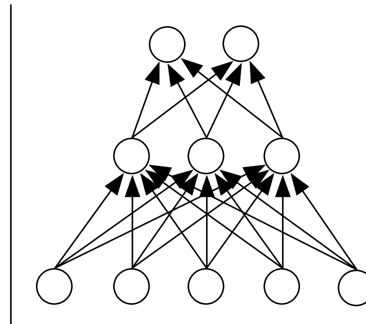
This example shows how to specify the evolvable properties and set the range of legal values. A typical error backpropagation model is a fully connected, feed-forward network that has a fixed input and output layer, and a hidden layer with variable size, as specified in Figure 3.3a. The number of nodes in the hidden layer is specified as

[SIZE [EVOLVE 2 10]]

in which the inner block means that this property is evolvable, and the two values inside of the brackets define legal boundary values for this property. Figure 3.3b depicts one possible realization (with three hidden nodes) of this description. When the initial population is generated based on the description file, all range-valued properties are filled with a random value within the legal range. Thus each individual in the population will generally have different property values although the population is created from a single description. Note that only the size property of

```
[SEQUENCE simple_ebp
   [LAYER input   [SIZE 5]
                  [ACT_RULE linear]
                  [CONNECT FWD]]
   [LAYER hidden  [SIZE [EVOLVE 2 10]]
                  [ACT_RULE logistic]
                  [CONNECT FWD]
                  [LEARN_RULE ebp]]
   [LAYER output  [SIZE 2]
                  [ACT_RULE logistic]
                  [MIN_ACT 0.0]
                  [MAX_ACT 1.0]]]
```

(a)



(b)

Figure 3.3: Error back propagation network with a hidden layer whose size is evolvable. (a) Initial network description. Some properties are not specified to keep the description small for illustrative purposes. The SIZE property of the hidden layer is evolvable. Indispensable parts are in bold type. (b) One possible realization or instantiation of this description in (a). The number of nodes in the hidden layer (here it is three) is randomly selected during the interpretation step.

hidden layer has a range which is evolvable in this example, and all other properties are fixed or missing (the latter will be replaced with default values). Therefore this initial description indicates a relatively small search space.

3.3.2 Self-Organizing Map (SOM) Model

The next example illustrates the description of a spatial structure and the corresponding activation/learning dynamics. A self-organizing map [Kohonen, 1982], depicted in Figure 3.4a, is a neural network method for unsupervised learning that creates a mapping from high dimensional input vectors to a lower dimension, typically a two-dimensional, lattice network.

The whole network is sequential, with one input layer of fixed size 5x5, and two map layers in parallel. The SIZE property in the left map layer is defined as [SIZE [EVOLVE [5 10][5 10]]], which means that the left map network is two-dimensional and evolvable, and all individuals in the initial population start with arbitrary size between 5x5 and 10x10, e.g., a 7x8 left map layer. The right map layer may have a larger size, between 10x10 and 20x20, e.g., 12x17. Note that there are bilateral (recurrent) connections between two map layers that are defined as inhibitory connections by the INIT property. Figure 3.4b illustrates a possible realization (with 5x6 *left_cortex* and 10x10 *right_cortex*) of this description. In this example, the only evolvable network aspects are the dimensions of the two cortical map layers, and the connections between left and right map layers make this a recurrent network.

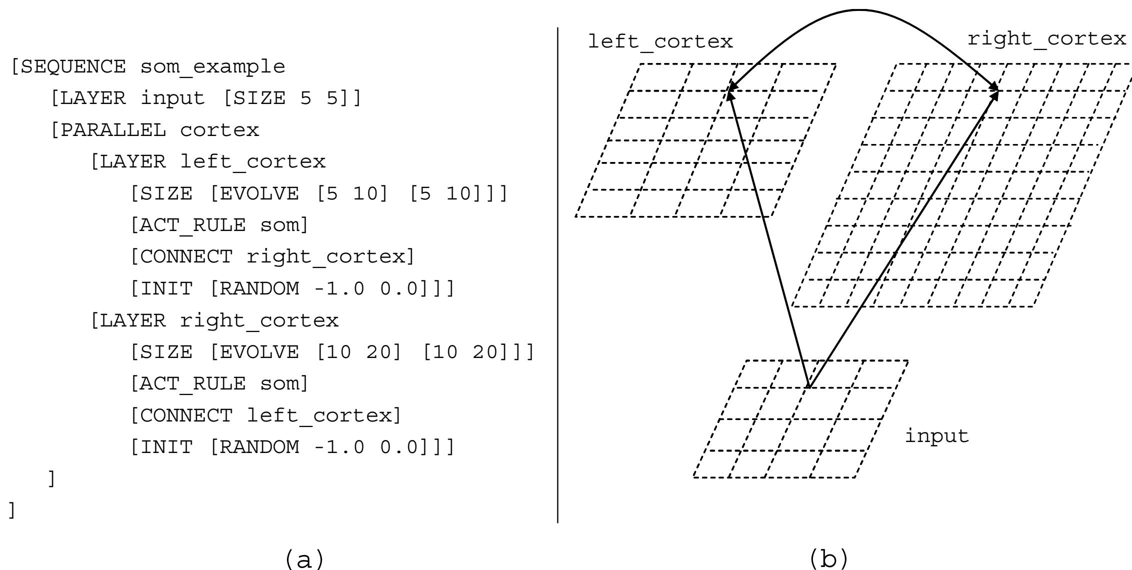
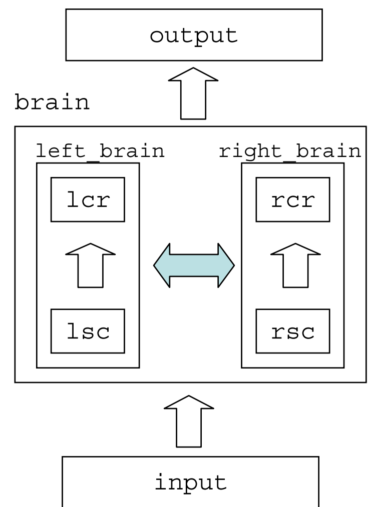


Figure 3.4: A self-organizing map (SOM) example, adapted from [Levitan and Reggia, 2000], illustrating how even asymmetric unsupervised learning models can readily be represented by descriptive encoding language. (a) Two-dimensional layers with the unsupervised activation rule (SOM) is specified. (b) One possible realization or instantiation of the description. Each vertex in the grids denotes a network node. Example connections between input layers are illustrated by arrows. Note that the two map (upper) layers are asymmetric.

```
[SEQUENCE asymmetric_network
  [LAYER input [SIZE 8]]
  [PARALLEL brain
     [SEQUENCE left_brain
        [LAYER lsc [SIZE [EVOLVE 5 20]]
                   [CONNECT rsc][INIT [-1.0 1.0]]]
        [LAYER lcr [SIZE [EVOLVE 5 20]]
                   [CONNECT rcr][INIT [-1.0 1.0]]]]
     [SEQUENCE right_brain
        [LAYER rsc [SIZE [EVOLVE 2 10]]
                   [CONNECT lsc][INIT [-1.0 1.0]]]
        [LAYER rcr [SIZE [EVOLVE 2 10]]
                   [CONNECT lcr][INIT [-1.0 1.0]]]]
  ]
  [LAYER output [SIZE 6]]
]
```

(a)



(b)

Figure 3.5: A complex multi-modular model, adapted from [Reggia et al., 2001a]. (a) Two blocks (left_brain, right_brain) in the middle structure may have different sizes. The INIT statements declare that these bilateral connections are fixed. (b) Conceptual organization of the network. Each arrow illustrates (a group of) connections forming a pathway, and the bilateral connections are represented by a shaded arrow.

3.3.3 Asymmetric Multi-Modular Model

Figure 3.5b illustrates an example of a parallel, asymmetric network architecture with bilateral connections, which is motivated by and modified from [Shkuro and Reggia, 2003]. Although the size of the input and output are fixed, the other four layers may have different sizes during either initial random creation or the evolution process. This description requires search of a larger and more rugged landscape (with multiple local minima) than previous examples. Bilateral connections between hidden layers can have different connection strengths among individuals, but actual weight values are fixed and are not involved in learning or evolution. The description for this network is depicted in Figure 3.5a.

The topmost network module has three sub-network modules; input, brain, and output. And the brain network module has two parallel sub-network modules with two bilateral connections, linking upper left/right layers and lower left/right layers, respectively. Note that the INIT statements explicitly specify that these lateral connections are fixed, and are initialized between -1.0 and 1.0. Modules that have the same properties are condensed here, but their actual property values will never be the same in general because of the random initialization process.

3.4 Language Specification

With the above examples in hand, we can now turn to examining in detail the language for describing layer/network modules, their properties, the default values of properties, and what aspects of a neural network are evolvable. The grammar

defining descriptive language is specified in Appendix.

3.4.1 Layer

A layer module is an array (either one-dimensional or two-dimensional) of network nodes that share common properties, and it is the basic unit of network composition. The description of a layer starts with an identifier LAYER, which is followed by an optional layer name and a list of properties. As explained earlier, each property has its own default value for simplicity. Therefore if some properties are missing in the description file, they will be replaced with the default values during the initialization stage and considered as being constant throughout the evolutionary process (i.e., the chromosome is in fact more strict than the description; it requires all default, fixed, and evolvable properties to be present). All default values are listed in Table 3.2. Detailed explanations for structure (BIAS, NEIGHBOR, NUMBER, SIZE, TOPOLOGY) and dynamics (ACT_RULE, INITACT, MINACT, MAXACT) properties are as follows:

- BIAS : There is one bias input per each node, except for the input layer(s). The input layer(s) and output layer(s) are identified during the interpretation step. If not set to a specific value, the bias is usually a random real number between -1.0 and 1.0.

- NEIGHBOR : In a competitive network, this property confines the range (radius) of lateral competition. For example, if this property is set at 1, each node will compete with only the adjacent nodes in the same layer. The application of

Table 3.2: Default Values of Properties

Category	Property	Default	Notes
Structure	BIAS	Random	[-1.0, 1.0]
	NEIGHBOR	0	Only effective with competitive networks.
	NUMBER	1	
	SIZE	1	
	TOPOLOGY	GRID	Only effective when SIZE is two-dimensional.
Dynamics	ACT_RULE	LOGISTIC	LINEAR for the input layers
	INIT_ACT	0.0	
	MIN_ACT	0.0	
	MAX_ACT	1.0	
Connection	CONNECT	FWD	
	INIT	Random	[-1.0, 1.0]
	LEARN_RATE	0.1	Dependent on LEARN_RULE
	LEARN_RULE	EBP	
	RADIUS	FULL	Fully connected.

this property is also affected by SIZE (one-dimensional or two-dimensional), TOPOLOGY (grid or hexagonal shape), and ACT_RULE (WTA or SOM) properties.

- NUMBER : Total number of duplicate layers to be created with current layer information in this network. A typical usage of this property is to make different number of layers in each individual. For example, [LAYER ... [NUMBER [1 10] ...] designates that this layer will be randomly copied from one to ten times into each chromosome.

- SIZE : This is the number of nodes in this layer. If it is two-dimensional, two numbers specify the number of rows (vertical length) and columns (horizontal length).

- TOPOLOGY : Significant only if the SIZE property specifies a two-dimensional structure. Currently square grid (GRID) and hexagonal (HEX) structure are

available. In the grid structure, each node is a vertex of the grids surrounded by four adjacent nodes, while each node (vertex) has three adjacent nodes in the hexagonal structure.

- ACT_RULE : An activation rule (transfer function) for each node in this layer is selected from a set of rules. This set currently includes linear (LINEAR), sigmoid (LOGISTIC), winner-take-all (WTA), and multi winner self-organizing map (SOM) rules.

- INIT_ACT : This is the initial activation level (output value) of network nodes before any input pattern is given into this layer. This property is useful in recurrent networks because when the first input pattern is given to the network, all the recurrent output values may be decided by this property.

- MIN_ACT, MAX_ACT : The minimum and maximal values permitted for the activation level.

Connection statements specify the connection topology between two layer (or network) modules, and all learning properties for this connection. This implies that each connection may have its own learning rules and parameters, while most other approaches use just a global set of learning parameters. Although the descriptive encoding does not maintain whole weight vectors in a chromosome, it does keep a random seed value and the range of initial weights per each connection. This approach can be considered as a variant of hybrid methods that use evolutionary algorithms for the global, initial weight searching, combined with other local

search algorithms (e.g., back-propagation) for fine tuning during training. Detailed explanations for connection properties (CONNECT, INIT, LEARN_RATE/RULE, RADIUS) are as follows:

- CONNECT : Specifies the destination layer(s) of this connection, starting from the current layer. This can be done by listing names of target layer/networks. If a name of a network (e.g., *brain, left_brain* or *right_brain* in Figure 3.5) is specified, all layers contained in that network will be connected. If the list of the destinations is declared as evolvable (i.e., in parentheses), the number of connections and the destination layers will be selected randomly among them. Predefined symbolic values currently include FWD and BOTH; 'FWD' denotes a feedforward linkage between two adjacent layers in a sequential network, and 'BOTH' means a mutual connection between two neighbor layers, in either a sequential or parallel network.

- INIT : This property describes how to initialize the connection weights. When a symbolic value RANDOM is declared, connection weights are randomly chosen between -1.0 and 1.0, unless the legal range is explicitly specified in the description. If a real value is specified, all weights are set to that value and considered fixed during training; all learning rule related properties for this connection are ignored.

- LEARN_RATE, LEARN_RULE : Define how to train this connection. Standard error-backpropagation (EBP), resilient backpropagation (RPROP), and Hebbian (HEBB) are currently the possible learning rules. Note that these

properties should match up with activation rules (ACT_RULE). Such restrictions on matching values among properties are checked in the interpretation step.

- RADIUS : A full connection (i.e, each node in the source layer is connected to all nodes in the target layer) may be specified by a symbolic value, 'FULL'. Otherwise, a positive real value/range specifies the range of connectivity neighborhood.

- SEED : Stores the random seed value and the range of initial weight values.

3.4.2 Network

A network module (i.e., a SEQUENCE, PARALLEL, or COLLECTION description) functions as a container to build a top-down, and hierarchical representation. There are three identifiers for discerning the type of the network module. If a network module starts with the SEQUENCE identifier, the sub-network chain contained in this module is considered to be arranged in a sequential manner, e.g., like a typical feed-forward neural network. Using the PARALLEL identifier declares that the sub-network chain to be arranged in parallel.

A COLLECTION module is also a network module in which the architecture of the network will be created arbitrarily. It may contain layer descriptions only. After identifying the total number of layers in this network, the type of the topmost network (either SEQUENTIAL or PARALLEL) and the number of sub-networks are randomly selected. Then layers are again randomly assigned to each sub-network,

and this hierarchical creation of the network structure will continue in a recursive manner, until no layer remains. This type of network description is useful when there is little knowledge about appropriate structure between input and output layer.

3.5 Encoding Properties

The descriptive encoding approach describe above has some important encoding properties. It can represent recurrent network architectures, and is scalable with respect to node/connectivity changes. More specifically, the encoding approach being used here has:

- *Closure* : A representation scheme can be said to be closed if all genotypes produced are mapped into a valid set of phenotype networks [Balakrishnan and Honavar, 1995]. First, every genotype at the initial step is decoded into a valid phenotype since the initial population of genotypes is based on the user-defined description. Next, descriptive encoding is closed with respect to mutation operators that change property values in a layer, since property values are only allowed to be mutated within the legal ranges defined by users or the system. This is checked at runtime and any illegal mutation result is discarded. Although descriptive encoding scheme is not closed with crossover operators on a grammar level, it can be constrained to be closed on a system level by adjusting invalid property values, according to the description file. For example, if the number of layers in a network becomes too large after a crossover operation, such a network may be deleted (or the whole network

structure could be adjusted to maintain legal genotypes).

- *Completeness* : The descriptive encoding scheme can be used to represent any recurrent neural network architecture. This can be easily seen from the fact that if one confines the size of each and every layer to be one node, the descriptive encoding scheme is equivalent to a direct encoding which specifies full connectivity on a node-to-node basis.

- *Scalability* : This property can be defined by how decoding time and genotype space complexity are affected by a single change in a phenotype [Balakrishnan and Honavar, 1995; Gordon and Bentley, 2005]. The descriptive encoding scheme described above takes $O(1)$ time and space in a node addition/deletion, since changing the number of nodes means just changing a parameter value in a property in the corresponding genotype, and node addition/deletion does not make substantial changes in time and space requirements during the genotype-phenotype mapping. In a similar way, a node-to-node connection addition/deletion in a phenotype will cost $O(1)$ space in genotype and $O(N+C)$ decoding time, as N denotes the total number of nodes in a network, and C denotes the total number of layer-to-layer connections. If a connection is deleted in a phenotype, it will split the corresponding source and target layers since nodes in these layers do not share connectivity anymore, but this split is equivalent to deleting a node in both layers plus creating two single-node layers, which will cost $O(1)$ space (assuming a constant layer size) and $O(N+C)$ additional decoding time. In general, the descriptive

encoding scheme is $O(1)$ scalable with respect to nodes and $O(N+C)$ scalable with respect to connectivity.

3.6 Neuroevolutionary Process

3.6.1 Development and Learning Stage

The evolutionary process that is built upon the descriptive encoding introduced above involves an initialization step plus a repeated cycle of three stages, as shown in Figure 3.6. First, the text description file prepared by the user is parsed and an initial random population of chromosomes (genotypes) is created within the search space represented by the description (leftmost part of Figure 3.6). During the development stage, a new population of realized networks (phenotypes) is created or "grown" from the genotype population. Each phenotype network has actual and specific individual nodes, connection weights, and biases (see [Jung and Reggia, 2004a, 2006] for details). The learning stage involves training each phenotype network, assuming that the user specifies one or more learning rules in the description file, making use of an input/output pattern file that contains training data. As evolutionary computation is often considered to be less effective for local, fine tuning tasks [Yao, 1999; Ferdinando et al., 2001], neural network learning methods are adopted to train connection weights. In Chapter 6, this approach is expanded to adjust weights by evolution strategy in order to address reinforcement learning problems where gradient information is not available. After the adaptation stage, each individual network is evaluated according to user-defined fitness criteria and

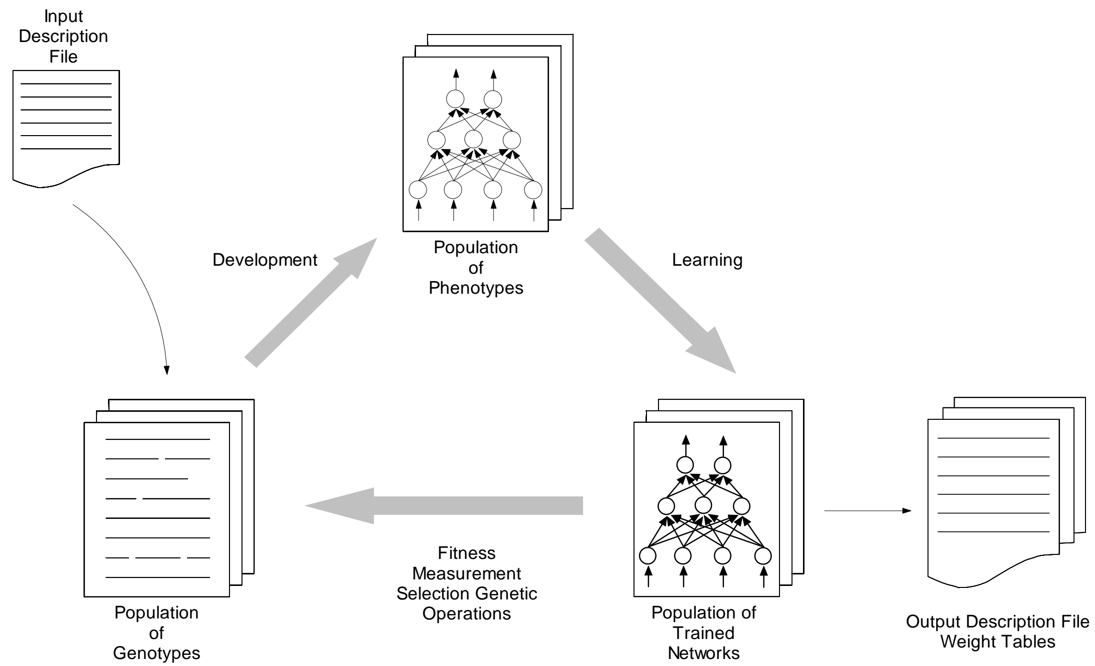


Figure 3.6: The iterative three-step development, learning, and evolution procedure used in this research. The input description file (upper left) is a human-written specification of the class of neural networks to be evolved (the space to be searched) by the evolutionary process, as outlined in the preceding sections of this chapter. The output file (lower right) is a human-readable specification of the best specific networks obtained that is described using the same encoding language.

genetic operators are applied to the genotypes. Fitness criteria may reflect both network performance (e.g., mean squared error) and a penalty for a large network (e.g., total number of nodes), or other measures. The end result of an evolutionary run consists of two things. First, an output description is produced (see Figure 3.6, bottom right). This file uses the same syntax as the input description file to specify the most fit specific network architectures discovered by the evolutionary process. Second, another file gives a table of all the weights found by the final learning process using these specific network architectures, so that a complete specification of the best neural networks is produced.

As explained above, a new aspect of the approach taken in this evolution system relative to past related work is the explicit use of an initial network description. This text file, which is written by the user, specifies the legal search space of neural networks that can be evolved. It permits a user to specify restrictions or constraints on what legal architectures are possible in a broad or narrow sense, including on network architecture, activation dynamics, range and initialization methods of network parameters, evolvability of each parameter, and learning rules. Further, the high-level modular structure of a network can be defined, indicating in a human readable form any constraints on evolution (e.g., whether feed-forward networks or recurrent networks are allowed, or limits on the number of parallel or serial hidden layers). To my knowledge no past developmental system has allowed this.

3.6.2 Evolutionary Process

3.6.2.1 Fitness Evaluation

Fitness measure can be defined in two ways: by explicitly specifying fitness using properties of neural networks, or by a user-defined fitness function. The default fitness measure is normalized, mean squared error (MSE). Other measures available for parsimony include the total number of network nodes (TOT_NODE), the total number of layer-to-layer connections (TOT_CONN), the total number of node-to-node connections (TOT_WEIGHT), and the total sum of absolute weight values (TOT_WEIGHT_SUM). These measures can be used together in various formats including weighted sum (WEIGHTED_SUM), product (MULTIPLY), and multi-objective optimization (SPEA), which will be explained in detail in Chapter 4 and 5. When the fitness property is declared as external, it means that a user-defined fitness function is provided outside of the descriptive encoding system. This user-defined fitness function is useful when the fitness of networks does not depend on networks' own properties. An example of this case will be explained in Chapter 6.

3.6.2.2 Selection Process

A tournament selection process is used in this research. A tournament pool size (TOURNAMENT_POOL) means the number of individuals compared in a single selection process. Fitness value of each network is compared with that of other randomly selected networks. Then one fittest chromosome among them is selected and copied to the next generation. No individuals other than the winners in the

tournament are inserted into the new population (no elitism).

3.6.2.3 Mutation

The seven existing operators are listed below, each of which is selected for use with equal probability. Note that all operations are conducted on evolvable properties in a chromosome only, within a range specified in the description file, if available. The initial description file is implicitly used in this step, since each chromosome has only specific property values and does not keep meta-information such as which property is declared as evolvable. The existing mutation operators are:

- Change Size: This operator starts with finding a list of layers in which the SIZE property is declared as evolvable. After randomly selecting a layer in the list, the new layer size will be chosen within the specified range. Note that a change of the layer size will implicitly affect the connectivity from/to this layer.

- Add Layer: Adding and deleting layer operators will only work under a random-structured network, specified as a COLLECTION network in the initial description, in order to avoid demolishing the specific architecture initially fixed by the designer. A set of connections starting from the newly added layer will be randomly created, as well as another set of connections targeting this layer.

- Delete Layer: This operator randomly deletes a layer and corresponding incoming/outgoing connections.

- Add Connection: This operator requires two random selections: a layer with evolvable outgoing connection (i.e., the CONNECT property is declared as evolvable), and a target layer to connect with. Note that adding/deleting connection operators act on a single layer-to-layer connection (i.e., they do not add a node-to-node connection, nor delete all connections from a layer).

- Delete Connection: This operator picks a layer then randomly deletes a connection from that layer. All the requirements for deletion are the same as those of the addition operator.

- Swap Connection: Like the add/delete connection operators, a list of layers with evolvable connections will be made. Then two randomly-picked layers will swap all their connections, and deleting illegal connections (i.e., the initial description does not allow such connections) will follow.

- Swap Layer: This operator will also work under a random-structured network. It selects two layers contained in the same network and swaps their topological order. All incoming and outgoing connections in both layers are not affected by this change, if they remain legal. However, this change will affect the order of (back)propagating output signals and errors, giving variations in the network performance.

3.6.2.4 Topology Preserving Crossover

The typical tree structured crossover operator in genetic programming is used in the evolutionary system, with two restriction rules. The first rule is that both of the roots of the subtrees that will be exchanged by a crossover operation should have a COLLECTION network as their common ancestor. This rule is required in order to build legal resultant networks in the search space that is specified by the description file. The second restriction rule is that all connections that are cut from a crossover operation should be re-connected to a topologically closest position of the original target layer. The topology of a layer is defined as the unique, directed path from the root network (the outmost network) to the layer, which can be specified as a string. In a similar manner, the topology of a connection can be defined as the shortest, directed path from a source layer to a target layer. When a layer is put in another network by a crossover operation, all outgoing connections originating from this layer are connected to the topologically closest layers of the new network, and incoming connections from the new network are linked to this layer only if the topology of a connection matches with the current position of the layer.

Chapter 4

Module Formation in a Feedforward Network

4.1 Introduction

Many animal nervous systems have parallel, almost structurally independent sensory, reflex, and even cognitive systems, a fact that is sometimes cited as contributing to their information processing abilities. For example, biological auditory and visual systems run in parallel and are largely independent during their early stages [Kandel et al., 1991]; the same is true for segmental monosynaptic reflexes in different regions of the spinal cord [Kandel et al., 1991], and the cerebral cortex is composed of regional modules with interconnecting pathways [Mountcastle, 1998]. Presumably evolution has discovered that such *partitioning* of neural networks into parallel multi-modular pathways is both an effective and an efficient way to support parallel processing when interactions between modules are not necessary. However, the factors driving the evolution of modular brain architectures having components interconnected by distinct pathways have long been uncertain and currently remain a very active area of discussion and investigation in the neurosciences [Dimond and Blizard, 1977; Brown et al., 2001; Killackey, 1996; Tooby and Cosmides, 2000].

Inspired by such modular neurobiological organization, and as a first test problem for the descriptive encoding system described in the preceding chapter, I examined whether an evolutionary process could discover the existence and details

Table 4.1: Training Data for a 2-Partition Problem

Input	Output	Input	Output	Input	Output	Input	Output
0 0 0 0	0 0	0 1 0 0	1 0	1 0 0 0	1 0	1 1 0 0	0 0
0 0 0 1	0 1	0 1 0 1	1 1	1 0 0 1	1 1	1 1 0 1	0 1
0 0 1 0	0 1	0 1 1 0	1 1	1 0 1 0	1 1	1 1 1 0	0 1
0 0 1 1	0 0	0 1 1 1	1 0	1 0 1 1	1 0	1 1 1 1	0 0

of n independent neural pathways between subsets of input and output units that are implied by training data. In the rest of the dissertation such problems will be called *n-partition problems*. Table 4.1 gives an example when $n = 2$ of a 2-partition problem. The goal is to evolve a minimal neural network architecture that can learn the given mapping from four input units to two output units, all of which are assumed to be standard logistic neurons in this case. The data in Table 4.1 implicitly represent a 2-partition problem in that the correct value of the first output depends only on the values of the first two input units (leftmost columns in the table), while the correct value of the second output depends only on the values of the remaining two input units. In other words, for example, the last two input nodes provide no information about the target value of the first output node. Thus, two parallel independent pathways from inputs to outputs are implied, and in this specific example each output is arranged to be the exclusive-or function of its corresponding inputs. A human designer would recognize from this information both that hidden units are necessary to solve the problem (since exclusive-or is not linearly separable), and that two separate parallel hidden layers are a natural minimal architecture. Of course, given just the training data in Table 4.1 and not knowing a priori the input/output relationships, such a design would not be evident in advance to an evolutionary algo-

rithm, nor most likely to a person, and the question being asked here is whether an evolutionary process would discover it when given a suitable descriptive encoding.

4.2 Encoding Details

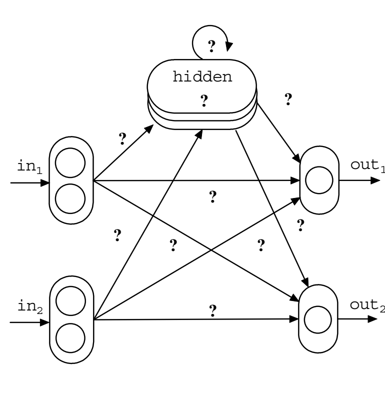
Figure 4.1a shows the description file used to evolve a neural network that solves the 2-partition problem of Table 4.1. First, it specifies the initial network architecture for the 2-partition problem, in which only the number of input and output nodes are fixed while other aspects, such as inter-layer connectivity and hidden layers' structure as shown in Figure 4.1b, are randomly created during initialization and evolve. In this example the input neurons are separated into groups that form the basis for the distinct pathways, but note that the learning algorithm makes no use of this fact. The NUMBER statements assign the range of how many layers of each type may be created with the same properties in the network. So the description for the input layer is equivalent (except for optional layer names) to specifying this:

```
[LAYER in1 [SIZE 2] [CONNECT [EVOLVE hidden output]] ]
[LAYER in2 [SIZE 2] [CONNECT [EVOLVE hidden output]] ]
```

The CONNECT property in the input layers descriptor indicates that nodes in each input layer may evolve to connect to hidden layers, output layers, neither or both. Second, this file also specifies the boundaries of the search space. The COLLECTION description indicates that networks can evolve to have zero to ten hidden layers, each with 1 to 5 nodes, and that they can be connected arbitrarily to them-

```
[SEQUENCE 2_partition
  [PARALLEL input
    [LAYER in  [NUM_LAYER 2][SIZE 2]
               [CONNECT      [EVOLVE hidden output]]]]
  [COLLECTION hidden
    [LAYER hid [NUM_LAYER    [EVOLVE 0 10]]
               [SIZE         [EVOLVE 1  5]]
               [CONNECT      [EVOLVE hidden output]]]]
  [PARALLEL output
    [LAYER out [NUM_LAYER 2][SIZE 1]]]]
[TRAINING
  [TRAIN_DATA "./inout_pattern.txt"] [MAX_TRAIN 100]
  [LEARN_RULE rprop]]
[EVOLUTION
  [FITNESS WEIGHTED_SUM
    [MSE:0.5 TOT_NODE:0.2 TOT_CONN:0.2]]
  [SELECTION TOURNAMENT] [TOURNAMENT_POOL 3] [ELITISM 0]
  [MUTATION_PROB 0.7]    [CROSSOVER_PROB 0.0]
  [MAX_GENERATION 50]    [MAX_POPULATION 50]]
```

(a)



(b)

Figure 4.1: (a) The initial network description and (b) a sketch of the space of networks to be searched for the 2-partition problem of Table 4.1.

selves and to the output layers. The EVOLVE attributes listed here indicate that the connections from input layers to hidden and/or output layers, the number and size of hidden layers, and the connections from hidden layers to other hidden layers and output layers, are all evolvable. These combinations are automatically, randomly and independently decided at the initialization step and enforced by genetic operators throughout the evolution process.

Each chromosome created from this description stores a representation of an architecture in the form of a tree, as well as other network features as embedded parameters (properties) in the tree. This hierarchical description of the network architecture has some benefits over a linear list of layers in previous layer-based encoding schemes, since it directly maps the topology of a network into the representation. These benefits are: 1) it enables crossover operations on a set of topologically neighboring layers, which was not possible with point crossover operators; 2)

functionally separated blocks can be easily specified and identified in a large scale, multi modular network; and 3) reusable subnetworks can be defined to address the scalability problem (e.g., like ADFs in GP [Koza, 1994]).

Figure 4.2a and b illustrate two example networks automatically generated from the description file of Figure 4.1a; they show different numbers of layers and topologies. Figure 4.2c shows the corresponding chromosome or genotype structure of one of these, the network in Figure 4.2a. Note that the overall structure is the same as the initial description in Figure 4.1a, but the COLLECTION hidden network has been replaced with a PARALLEL network with three layers and each property has a fixed value (i.e., the EVOLVE attributes are gone). Figure 4.2d shows the tree like structure of this genotype, making it amenable to standard genetic programming operators.

4.3 The Evolutionary Procedure

As explained earlier, a descriptive encoding generally provides additional information about the training and evolutionary processes to be used, which is illustrated here. This user-defined information follows the network part of the description (Figure 4.1a, top), setting various parameter values to control the training and evolutionary procedure. In this case, as specified in the training block in Figure 4.1a, each phenotype network is to be trained for 100 epochs with the designated input/output pattern file that encodes the information from Table 4.1. The default learning rule is defined as a variant of backpropagation (RPROP [Riedmiller and

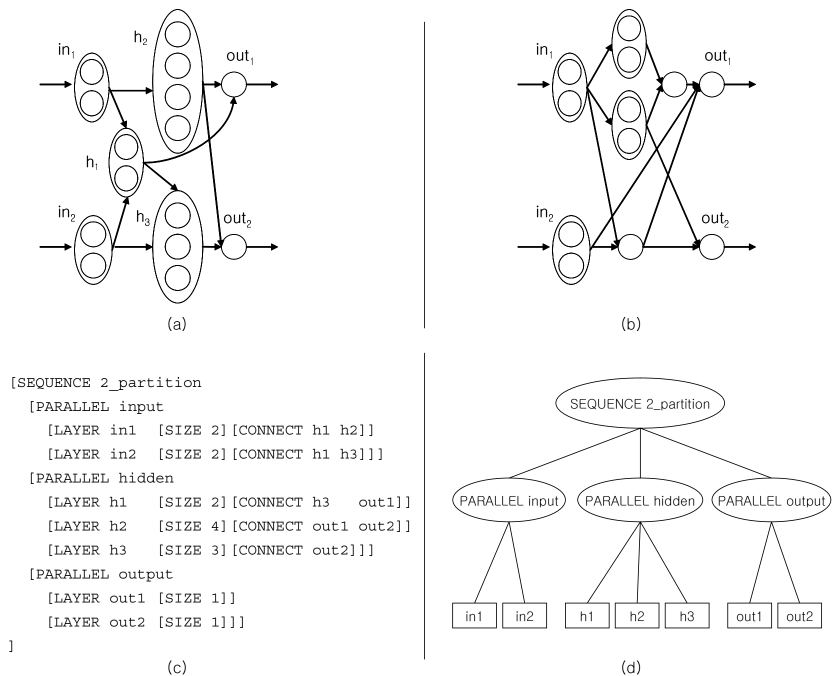


Figure 4.2: (a),(b) Examples of neural network architectures randomly created from the description in Figure 4.1a during initialization. Input and output layers are the same, but the number of hidden layers and their connections are quite different and specific now. Arrows indicate pathways that are sets of connections between layers (i.e., not individual node-to-node connections). (c) The chromosome description of the network illustrated in (a), as it would be written in our descriptive language. This is *not* a description file written by the user, but is automatically generated from that description file. Note that no EVOLVE attributes are present, for example. (d) Top part of the tree-like structure of the genotype in (c), making it directly usable by GP operators. Each rectangle designates a layer.

Braun, 1993]). Note that there is no issue of generalization in learning the boolean function here since all inputs and the correct outputs for them are given a priori. The EVOLUTION part of the description (Figure 4.1a, bottom) indicates that a weighted sum method of three criteria are to be used: mean squared error (MSE, e), total number of network nodes (n), and total number of layer-to-layer connections (c). MSE reflects the output performance of the network, and the other two criteria are adopted as penalties for larger networks. These three criteria are reciprocally normalized and then weighted with coefficients assigned in the description file. More specifically, the fitness value of the ith network, $Fitness_i$ that is described here is

$$Fitness_i = w_1 \cdot \left(\frac{e_{max} - e_i}{e_{max} - e_{min}}\right) + \ w_2 \cdot \left(\frac{n_{max} - n_i}{n_{max} - n_{min}}\right) + \ w_3 \cdot \left(\frac{c_{max} - c_i}{c_{max} - c_{min}}\right) \quad (4.1)$$

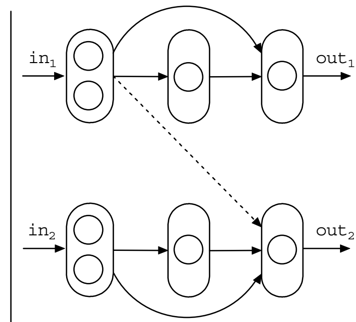
where $x_{min}(x_{max})$ denotes the minimum (maximum) value of criterion x among the population, and the coefficients w_1, w_2, and w_3 are empirically defined as 0.5, 0.2, and 0.2, respectively. In words, the fitness of an individual neural network is increased by lower error (a behavioral criterion), or by fewer nodes and/or connections (structural criteria). An implicit hypothesis represented in the fitness function is that minimizing the latter two structural costs may lead to fewer modules and independent pathways between them in evolved networks. Note that the EVOLUTION part of the description (Figure 4.1a) specifies the coefficients in the fitness function above, and it also specifies tournament selection with a pool size of 3 as the selection method, a mutation rate of 0.7, and that no crossover and no elitism are to be used. Operators in this case can mutate layer size and direction of an existing inter-layer connection, and can add or delete a new layer or connection.

4.4 Results of the Evolutionary Process

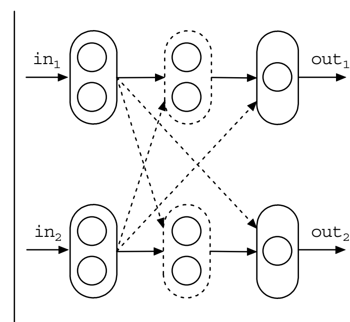
A total of 50 simulations were run with a fixed population size (50) and a fixed number of generations of 50. Between simulations, the only changes are the random initial architectures plus the initial value of connection weights that are assigned randomly in the range -1.0 to 1.0. For all runs, each final generation contained near-optimal networks that both solved the 2-partition problem (i.e., MSE $\sim$ 0.0) and had a small number of nodes and connections. Converged networks can be categorized into two groups identified by their connectivity pattern as depicted in Figure 4.3. The first group of networks, found during 44% of the runs, showed a dual independent pathway where each input pair has their own hidden layer and a direct connection to the corresponding output node (Figure 4.3a and 4.3b). Ignoring the dotted line connections which have near zero weight values shown in Figure 4.3a, this is an optimal network for the 2-partition problem in terms of the total number of network nodes and connections. In the second group of networks, found during 52% of the runs, input layers share a single hidden layer, without having direct connections to the corresponding output nodes. Such solutions require four hidden nodes, rather than two, to be an optimal network. Inspection of the connection weights shows that this model implicitly captures/discovers two distinct pathways embedded in the explicit hidden layer, if one ignores or prunes connections with near zero weights, as illustrated in Figure 4.3c. This type of network is also an acceptable near-optimal solution in terms of the number of connections needed for XOR problems and is sometimes used to illustrate layered solutions to single XOR

```
[SEQUENCE 2_partition
   [PARALLEL input
      [LAYER in1   [SIZE 2]
                   [CONNECT h1 out1 out2]]
      [LAYER in2   [SIZE 2]
                   [CONNECT h2 out2]]]
   [PARALLEL hidden
      [LAYER h1    [SIZE 1][CONNECT out1]]
      [LAYER h2    [SIZE 1][CONNECT out2]]
   ]
   [PARALLEL Output
      [LAYER out1 [SIZE 1]]
      [LAYER out2 [SIZE 2]]]]
```

(a)



(b)



(c)

Figure 4.3: Typical network architectures found during evolution for the 2-partition problem are depicted. Dotted lines show connectivity with near-zero weights. (a) Final output description file having two independent pathways. (b) Conceptual network architecture described by (a). (c) Dual pathway network without direct input-to-output connections. Implicit hidden sub-layers are indicated by dotted ovals.

problems in textbooks (e.g., [Haykin, 1999]). The remaining 4% of the runs did not converge on just one type of network as described above, but both types are found in the final population. Thus, the evolutionary process generally discovered that "minimal cost" solutions to this problem involve independent pathways. While the networks considered here are very simple relative to real neurobiological systems, the frequent emergence of distinct and largely independent pathways rather than more amorphous connectivity during simulated evolution raises the issue of whether parsimony pressures may be an underrecognized factor in evolutionary morphogenesis, as outlined at the beginning of this section (see [Shkuro and Reggia, 2003] for further discussion).

Without changing the evolutionary part of the description file, n-partition problems were tested for $n = 2, 3, 4,$ or 5 (the latter case requires $2^{2n} = 1024$ patterns for training). A typical input/output description file and network structure for $n = 5$ are illustrated in Figure 4.4. Table 4.2 summarizes the experimental results. The Minimum Hidden Nodes Found column shows the smallest number of hidden nodes found during the experiment, and the numbers in the parentheses are the theoretically possible minimum number of nodes. In an n-partition problem involving exclusive-OR relations, at least n hidden nodes are necessary even if direct connections from input to output are allowed. The Minimum Connections Found column shows the minimum number of layer-to-layer connections found in the best individual. Again assuming direct connectivity from input to output and without increasing the number of hidden nodes, the best possible number of connections in an n-partition problem is $3n$ (e.g., input to output, input to the corresponding

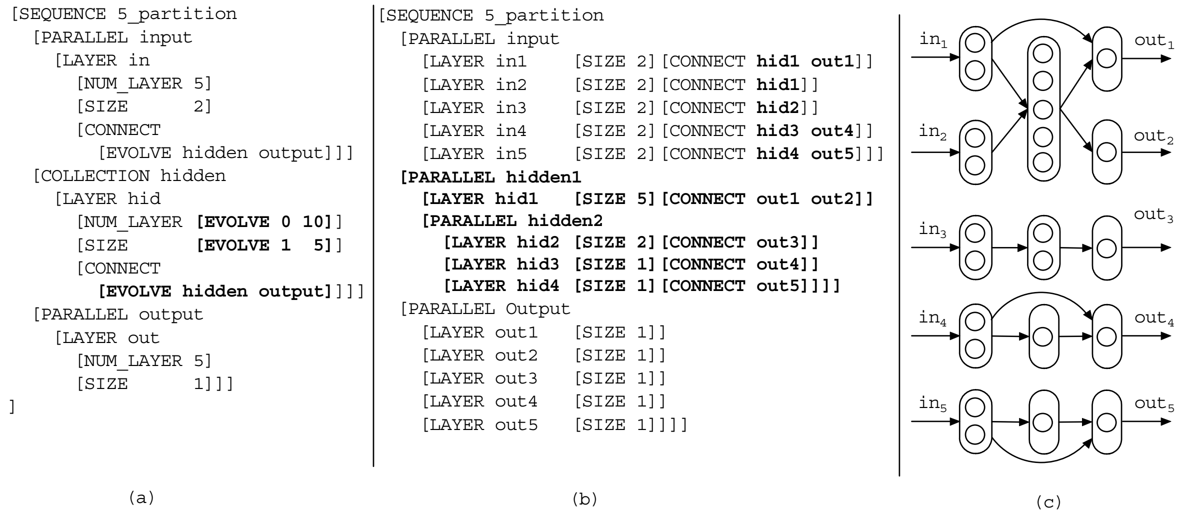


Figure 4.4: A typical example of a final evolved network for the 5 XOR partition problem. (a) Initial network description. Properties to be EVOLVEd are in bold font. (b) Final description file produced as output by the system. All EVOLVE properties have been replaced by the specific choices in bold font. Only SIZE and CONNECT properties are shown. (c) Depicted network architecture. Connections that have near-zero weights are pruned.

Table 4.2: Parallel n-Partition Problem Results

N	# of Patterns	Min. Hidden Nodes Found	Min. Connnections Found	Min. MSE	Fully Connected MSE	Average Time
2	16	2 (2)	7 (6)	0.00000	0.20823	2250
3	64	4 (3)	9 (9)	0.00021	0.20710	2848
4	256	5 (4)	14 (12)	0.00073	0.22224	3471
5	1024	9 (5)	19 (15)	0.00236	0.20821	6273

hidden layer, hidden to output), that are specified in the parentheses.

For each n partition problem, the best MSE results gathered from each evolutionary simulation were compared with that of standard fully connected, single hidden layer backpropagation networks. These latter networks have a single fixed hidden layer size of n, which is the theoretically minimal (optimal) number for each partition problem, initial weights randomly chosen from -1.0 to 1.0 (same as in the evolutionary simulations), and the MSE results averaged over 50 runs. With all other conditions set to be the same, post-training errors with the evolved networks are significantly less for each problem than with the standard fully connected backpropagation networks (p values on t-test were less than 10^{-5} for each of the four comparisons). More importantly, the fully connected networks sometimes produced totally wrong answers (i.e., absolute errors in output node values were more than 0.5), while this problem did not occur with the evolved networks. This shows the value of searching the architectural space even if it is believed that a fully connected network can theoretically approximate any function [Cybenko, 1989] ([Wolpert and Macready, 1997, 2005] for general discussion). The Average Time column in Table 4.2 shows the mean time (seconds) needed for a single evolutionary run. This result shows that the descriptive encoding system can identify the partial relationships be-

tween input and output patterns and represent them within an appropriate modular architecture.

4.5 Discussion

In this chapter it is shown that the descriptive encoding can efficiently represent the hierarchical structure of multi-layer neural networks, which is a desired property for designing large scale networks. The need for searching the space of network architectures is justified here by comparing the performance of evolutionary networks with networks that have predefined, fixed architectures. The evolutionary networks constantly outperform fixed architecture networks , and this comparison results also demonstrate the benefits of the hybrid approach combining evolutionary global search for the architectural space with gradient-descent based local tuning for the corresponding optimal weight values. Second, the n-partition problem introduced here shows an example of how the descriptive encoding can be used in problems with increasing complexity. Although the complexity of the problem in terms of the number of training patterns increases exponentially, the descriptive encoding requires no fundamental changes in order to encode and to address the problem, even for the evolutionary parameters in this case. Third, (near) optimal networks evolved for this problem typically have parallel, independent pathways, and the only factor that might be related to this result is a parsimony in the fitness measure that penalizes for larger networks. This emergence of modular architectures is interesting because it supports the hypothesis that modular design principle, on

which the descriptive encoding is based, will be of benefit in terms of performance, and because it can partly explain how natural evolution discovers modular structures.

Chapter 5

Learning Word Pronunciations Using Recurrent Networks

5.1 Introduction

Recurrent neural networks have long been of interest for many reasons. For example, they can learn temporal patterns and produce a sequence of outputs, and are widely found in biological nervous systems [Kandel et al., 1991]. They have been applied in many different areas (e.g., word pronunciation [Radio et al., 2001], and learning formal grammars [Giles et al., 1992]) and several models of recurrent neural networks have been proposed (see [Haykin, 1999; Kumar, 2004]). The previous chapter showed that the description-based encoding scheme was powerful enough to study research issues related to feedforward-only neural networks (i.e., inducing modular feedforward architectures by combining performance and parsimony in a single fitness function). Here I establish that the high-level descriptive language developed in this research is also sufficiently powerful to support the evolution of recurrent networks in situations involving multi-objective optimization. In other words, this current chapter shows that the neuroevolutionary approach introduced in this dissertation can be successfully applied to study research questions in situations involving two generalizations relative to the preceding chapter: recurrent networks, and multi-objective optimization.

To understand the results below, it is important to know some basic infor-

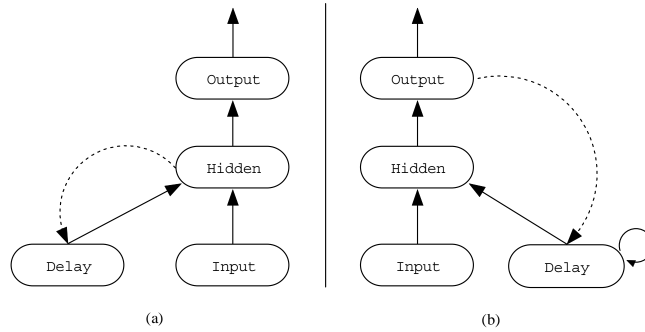


Figure 5.1: The Elman (a) and Jordan (b) network architectures shown here are widely used in neural network applications because of their simplicity, effectiveness, and efficiency. Dotted lines show the backward/recurrent one-to-one connections that essentially represent a copying of the output at one time step to a delay layer that serves as input at the next time step.

mation about recurrent networks, as follows. Two well-known, partially recurrent architectures that let one use basic error backpropagation from feed-forward nets essentially unchanged (because the feedback connection weights are fixed and unlearnable) are often referred to as Elman networks and Jordan networks. Elman [Elman, 1990] suggested a recurrent network architecture in which a copy of the contents of the hidden layer (saved in the delay layer) acts as a part of the input data in the next time step, as shown in Figure 5.1a. Jordan [Jordan, 1986] proposed a similar architecture except that the content of the output layer is fed back to the delay layer where nodes possibly also have a "decaying" self-connection, as shown in Figure 5.1b. These networks were originally proposed for different purposes: the Elman architecture for predicting the next element in a temporal sequence of inputs, and the Jordan architecture for generating a temporal sequence of outputs when

given a single fixed input pattern. However, little is known about how to select the best recurrent network architecture for a given sequence processing task and, to my knowledge, no systematic experimental comparison between these different recurrent neural network architectures has ever been undertaken, except for some specific application comparisons (e.g., [Pérez-Ortiz et al., 2001]). In other words, it is not currently established as to which of either of these two architectures is advantageous to use in applications, or what application features might guide such a choice.

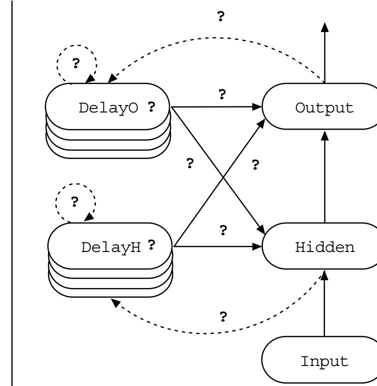
In this context, the essence of the problem considered in this chapter is to find appropriate recurrent networks to produce a sequence of phoneme outputs, given a *fixed* input representing the corresponding input word pattern. For example, for the word *apple*, a fixed pattern of the five letters A P P L E is the input, and the correct output temporal sequence of phonemes would be /ae/, /p/, and /l/, followed by an end of word signal. This challenging task was originally tackled in [Radio et al., 2001] using Jordan networks. Here, the same task is examined using an expanded set of input data (total of 230, two to six phoneme words selected randomly from the NetTalk corpus [Sejnowski and Rosenberg, 1987]). The focus is on finding the optimal architecture of delay layers and their connectivity. The question being asked is whether the high-level, modular developmental approach supported by the descriptive encoding language can identify the "best" recurrent architecture to use, or at least clarify the tradeoffs.

5.2 Encoding Details

Figure 5.2 gives the descriptive encoding and a corresponding schematic representation of the space of networks that for this problem. The fixed part of the structure is a feed-forward, three layer network consisting of input, hidden, and output layers (depicted on the right side of Figure 5.2b). The size of the input layer is decided by the maximum length of a word in the training data and the encoding representation, and the output layer size is 52 since a set of 52 output phonemes are used, including the end of a word signal. The number of hidden nodes is arbitrarily set to be the same as the output layer size. Note that hidden and delay layers may have connections to different destination layers with different configurations (e.g., CONNECT_RADIUS and LEARN_RULE), such that each set of properties for connections has been separated from the other by using double brackets in the description of Figure 5.2a. As shown here, the space of architectures to be searched by the evolutionary process consists of varying numbers of delay layers that receive recurrent one-to-one feedback connections (dotted arrows) from either the hidden layer (delayH) or the output layers (delayO). In either case, 0 to 4 delay layers may be evolved, but however many are evolved in each feedback pathway, they must be organized in a serial fashion, thus representing feedback delays of 0 to 4 time steps. Both feedback pathways from output and hidden layers may have zero layers, which means there are four possible architectures being considered during evolution: 1) feed-forward network only without delays; 2) hidden layer feedback only; 3) output layer feedback only; and 4) both types of feedback. In addition, for each class where

```
[SEQUENCE Psg_problem
   [LAYER Input   [SIZE 156] [CONNECT Hidden]]
   [LAYER Hidden  [SIZE  52]
      [[CONNECT DelayH:1] [CONNECT_RADIUS 0.0]
       [LEARN_RULE NONE]]
      [[CONNECT Output]]]
   [SEQUENCE DelayH
      [LAYER [NUM_LAYER [EVOLVE 0 4]] [SIZE 52]
         [[CONNECT FWD] [CONNECT_RADIUS 0.0]
          [LEARN_RULE NONE]]
         [[CONNECT [EVOLVE Hidden Output]]]]]
   [LAYER Output  [SIZE  52]
      [[CONNECT DelayO:1] [CONNECT_RADIUS 0.0]
       [LEARN_RULE NONE]]]
   [SEQUENCE DelayO
      [LAYER [NUM_LAYER [EVOLVE 0 4]] [SIZE 52]
         [[CONNECT FWD] [CONNECT_RADIUS 0.0]
          [LEARN_RULE NONE]]
         [[CONNECT [EVOLVE Hidden Output]]]]]
]
```

(a)



(b)

Figure 5.2: (a) The network description file for the phoneme sequence generation task. FWD, delayH:1, and delayO:1 mean to make a connection to the next layer in the same network block, to the first layer in the delayH network block, and to the first layer in the delayO network block, respectively. If such a block does not exist, the corresponding connectivity properties are ignored. The evolvable properties are in bold font. (b) A schematic illustration of the space of neural network architectures corresponding to the description file in (a) that are to be searched for the phoneme sequence generation problem. Dotted lines designate non-trainable, one-to-one feedback connections; solid lines indicate weighted, fully connected pathways trained by error backpropagation. Note that the Elman and Jordan networks of Figure 5.1 are included within this space as special cases.

a feedback pathway exists, it may have a varying amount of delay (1 to 4 time steps) and may provide feedback to the output layer, the hidden layer, or both. Each delay layer is sequentially connected to the adjacent delay layer by a one-to-one, fixed connection of weight 0.5, which acts as a decaying self-connection. Thus a total of 169 architectures are considered by the evolutionary process.[1]

5.3 Multi-objective Optimization

Fitness criteria based on two cost measures or objectives is used in this study: root mean squared error (RMSE) for performance, which was adjusted to be comparable with previous results [Radio et al., 2001], and the total sum of absolute weight values to penalize larger networks. The latter unbounded measure simply adds together the absolute values of all weights in the network after training. This is used rather than the number of nodes and connections as in n-partition problems, since the latter vary stepwise in this experiment while the summed weights are a continuous measure. This summed absolute weights measure is especially useful here as it can potentially discriminate between two different architectures having the same numbers of node and connections (e.g., Jordan vs. Elman networks). Similar weight minimization fitness criteria have been used previously when evolving neural networks and can be viewed as a "regularization term" that acts indirectly on an evolutionary time scale rather than directly during the learning process (see [Shkuro and Reggia, 2003] for discussion).

[1] No delay: 1, delayH only: 4 delays x 3 directions, delayO only: 4x3, both delays: (4x3)x(4x3) = 169 total.

A multi-objective evolutionary algorithm (SPEA [Zitzler and Thiele, 1999; Li et al., 2005; Knowles and Corne, 2003]) was adopted based on these two fitness criteria, which enables one to get a sense of the tradeoffs in performance and parsimony among the different good architectures found during evolution. This also illustrates that the evolutionary system studied here consists of components that can be expanded or plugged in depending on the specific problem. The population size was decreased to 25 compared to the configuration in n-partition problem because of the large computational expense of doing both learning and evolution in the same simulation, and the archive in which non-dominated individuals are stored externally in SPEA was set to be the same size as the population. The maximum number of generations was fixed at 50, and all networks trained for 200 epochs with RPROP, a variant of error backpropagation which has been shown to be very effective in previous research [Radio et al., 2001]. For genetic operations, only mutating the number of layers and their connectivity are allowed, specified by default and applied within the range of property values designated in the network description file.

5.4 Experimental Results

A total of 100 runs of the neuroevolutionary process were examined, randomly changing the initial network architectures and their weights in each run. The results are shown in Figure 5.3, averaged over the same architectures. In other words, each point in Figure 5.3 represents a network having a specific number of hidden

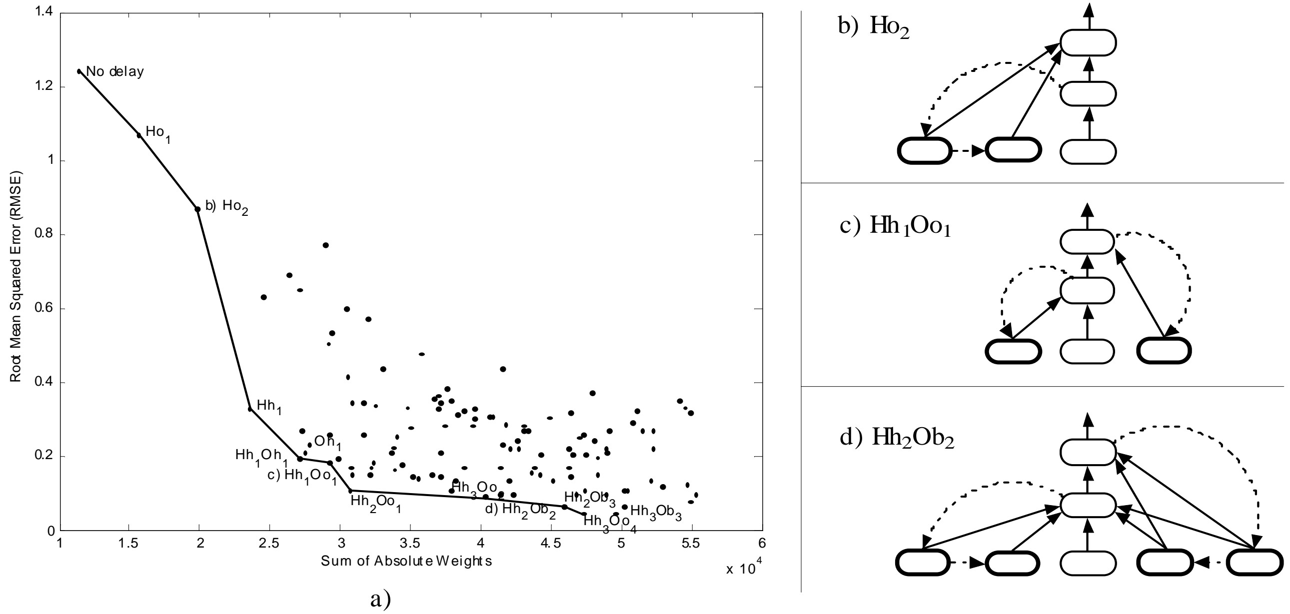


Figure 5.3: (a) The performance/weights result of networks from all final generations are depicted. Each point represents one network architecture's values averaged over all evolutionary runs (most points are not labeled). The points on the solid line represent the Pareto-optimal set, and the labels on some of these latter points designate the type of network that they represent. For example, label Hh_3Oh_1 means that the network represented by that node has both hidden (H) and output (O) delay layers, while there are three hidden and one output delay layers, in both cases connected to the hidden (h) layer (see text). (b)-(d) Example of evolved network architectures and their corresponding labels. Evolved layers are shown in bold ovals.

and output delays, and a specific layer-to-layer connectivity, with the RMSE and weight sum averaged over all runs. A total of 121 network architectures (over 169 theoretically possible architectures) were found in the final generations of all runs, and all of them are depicted in Figure 5.3. The labels "H$c_{\#}$O$c_{\#}$" in Figure 5.3 encode the network architecture evolved. More specifically, # indicates the number of hidden (H) or output (O) delays, and c designates the destination of delay outputs, either to the hidden (h), output (o), or both (b) layers. For example, "Hh_1Ob_2" means that there is one hidden delay layer (connected back to the hidden layer) and two output delay layers connected back to both hidden and output layer. Figure 5.3b-d shows some other examples of evolved network architectures. An example of the final network descriptions for Elman and Jordan networks is illustrated in Figure 5.4.

Several observations can be made from Figure 5.3. First, feed-forward only networks without delays still remain in the final Pareto-optimal set (upper left). The Pareto-optimal set in this context consists of "non-dominated" neural networks for which no other neural network has been found during evolution that is better on all of the objective fitness criteria. Thus, feed-forward networks are included in the Pareto-optimal set because of their quite small weight values, even though their performance is poor relative to the other types. Following the Pareto-optimal front downward, we see that networks with one or two hidden delay layers connected to the output layer (labeled "Ho_1" and "Ho_2") are the next Pareto-front points (upper left of Figure 5.3). This type of network in which delays are connected to the output layer does not provide good performance in general. A big increase in performance occurs

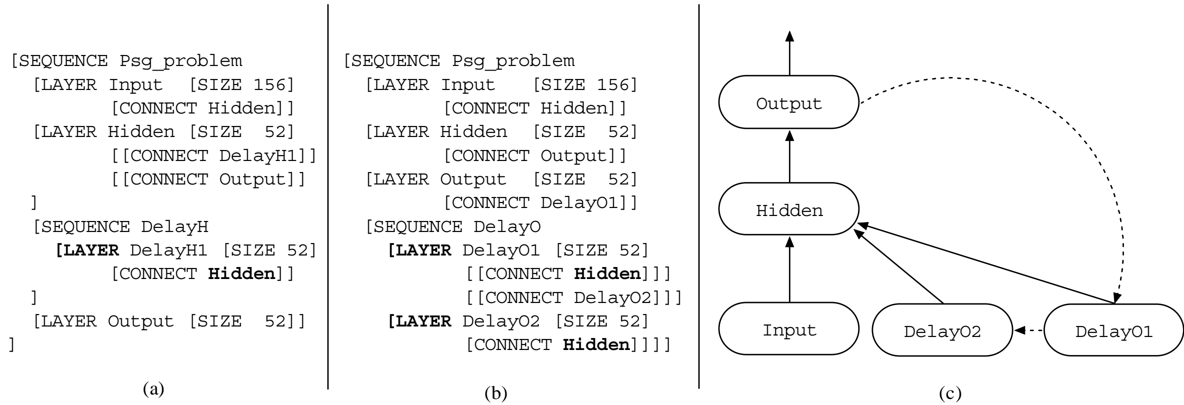


Figure 5.4: The final network description of (a) an Elman network with single delay (labeled "Hh_1" in Figure 5.3a) and (b) a Jordan-like network with double delays (labeled "Oh_2"). Only SIZE and CONNECT properties are shown. The evolved properties (including the number of layers) are in bold font. (c) An illustration of the Jordan network specified in (b). Dotted lines designate fixed, one-to-one connections.

however with networks having only hidden delay layers connected to the hidden layer (bottom left): an Elman network (labeled "Hh_1" in Figure 5.3 and depicted in Figure 5.4a) performs much better and is on the Pareto front. A Jordan network (labeled "Oh_1", lower right of "Hh_1" in Figure 5.3) performs even better at the cost of increased weights. Finally, networks with increasing numbers of delay layers that combine hidden and output delays generally performed progressively better, although at the cost of increasing numbers of weights and connections (bottom right in Figure 5.3). Surprisingly, "Hh_1Oh_1" on the Pareto-optimal front of Figure 5.3 performs better than the original Elman (Hh_1) and Jordan (Oh_1) networks with smaller total weight values than the latter, and would be a very good choice for an

Table 5.1: Representative Results for the Phoneme Sequence Generation Problem

Architecture	RMSE	Absolute Weight Sum	PCT*
No delay	1.239	11410.4	23.3
Ho_1	1.066	15711.7	43.2
Ho_2	0.869	19892.6	62.3
Oo_1	0.768	28909.3	70.5
Ob_1	0.599	30502.8	82.1
Hb_2	0.531	29421.4	85.9
Ho_1Ob_1	0.501	29243.5	87.5
Hh_1	0.328	23604.5	94.6
Hh_2	0.270	27232.6	96.4
Hh_3	0.255	29250.4	96.7
Oh_1	0.232	27893.6	97.3
Hb_3Oo_4	0.219	48989.6	97.6
Hh_2Oh_1	0.210	27540.5	97.8
Hh_1Oh_1	0.191	27090.4	98.2
Hh_1Oo_1	0.180	29290.6	98.4
Hh_3Oo_1	0.152	32109.6	98.9
Hh_2Oo_1	0.107	30718.3	99.4
Hh_3Oo_2	0.107	37946.3	99.4
Hh_2Ob_2	0.088	40383.5	99.6
Hh_2Ob_3	0.062	45920.2	99.8
Hh_3Ob_3	0.044	49589.8	99.9

*PCT = Percentage of phonemes generated completely correctly [Radio et al., 2001].

architecture for this problem (and one that was not evident prior to the evolutionary process). Summarizing, the Pareto-optimal front in Figure 5.3 and, more generally, the correlations between architectures and performance given in Table 5.1, explicitly lay out the tradeoffs for the human designer selecting an architecture. From a practical point of view, which Pareto-optimal architecture one would adopt depends on the relative importance one assigns to error minimization vs. network size in a specific application. A very reasonable choice would be networks such as Hh_2Oo_1 or Hh_2Ob_2 (Figure 5.3d) that produce low error by combining features from both Jordan and Elman networks while still being constrained in size. These results

show that the evolutionary approach using a high-level descriptive language can be applied effectively in generating and evaluating alternative neural networks even for complex temporal tasks requiring recurrent networks.

5.5 Discussion

In problem domains that involve memorizing temporal states or processing sequential patterns, recurrent neural network architectures that have feedback connections are typically required in order to solve the problem efficiently. However, for the two types of recurrent neural network architectures that have been widely used (Jordan and Elman Nets), no systematic comparison between them has been done yet. In this chapter, effective recurrent neural network architectures were evolved for a temporal sequence generation problem, demonstrating that the descriptive encoding system can be successfully applied to these problem domains of recurrent neural networks. Second, a multi-objective optimization method was incorporated in the descriptive encoding system, and the comparison of results of various recurrent architectures indicates the tradeoffs in the costs of architectural features versus network performance. A mixed network of two known recurrent architectures was discovered to outperform the two original networks in any fitness measure, which was not expected before the comparison. Third, this study shows an example of how descriptive encoding can be used to limit the search space effectively, and how user's domain knowledge can be utilized in describing the search space. The network description used here was general enough to include two known architectures as

possible phenotypes, as well as being specific enough to restrict the number of legal network architectures, making this systematic comparison practically available.

Chapter 6

Evolving an Autonomous Agent

6.1 Introduction

Reinforcement learning [Sutton and Barto, 1998] refers to a wide class of learning problems that deal with how an autonomous agent learns to choose the optimal behavior in its environment, without an explicit teacher. Unlike in supervised learning, there are no given input / output patterns for training in reinforcement learning problems, but only reward signals are provided to the agent after it processes one or more inputs. These signals indicate the desirability of the states of the environment where the agent reaches by (typically) choosing a sequence of actions, and the goal of the agent is to maximize the amount of cumulative rewards it receives from the environment in the long run. With this flexible problem definition, reinforcement learning has attracted much research and become an important sub-area of machine learning: many interesting real-world issues including game playing [Samuel, 1959; Tesauro, 1994], robotics [Asadi and Huber, 2007; Mataric, 1994; Schaal and Atkeson, 1994], and control problems [Crites and Barto, 1996] fall into this category. In this chapter I examine how the neuroevolutionary system developed in this research can be extended to address reinforcement learning problems by evolving recurrent neural network architectures and connection weights.

Historically there have been two main approaches to solving reinforcement

learning problems. One approach is to search in the space of behaviors in order to discover appropriate actions for the current state of the environment, which is largely done with evolutionary computation methods (e.g., [Paine and Tani, 2004]). The other approach focuses instead on estimating the usefulness of states of the environment, as the agent can take an action to reach the best state if it knows which possible next state would be the most beneficial. This has been largely done by conventional reinforcement learning algorithms utilizing dynamic programming and statistical inference [Sutton and Barto, 1998]. Although it is not yet clear which approach is better than the other under which circumstances [Kaelbling et al., 1996], there have been reports claiming that evolutionary algorithms are complementary to conventional reinforcement learning algorithms [Moriarty et al., 1999], or even found to be better in terms of performance [Stanley and Miikkulainen, 2002a] and robustness [Whitley et al., 1993].

In this context, the third problem examined in this research focused on building an agent that forages for food and avoids predators in a simulated artificial environment. This has been a frequent environment used in artificial life research [Reggia et al., 2001a; Ruppin, 2002]. For each simulation time step, the agent processes internal data and local sensory input coming from the environment, and then decides what its next movement should be. The movement of the agent is simulated in the environment and the updated sensory information is provided to the agent at the next time step. No performance information about the agent's behavior is fed back to the agent during the simulation, and the fitness of an agent is calculated after the simulation ends based on its food acquisition / consumption ratio and

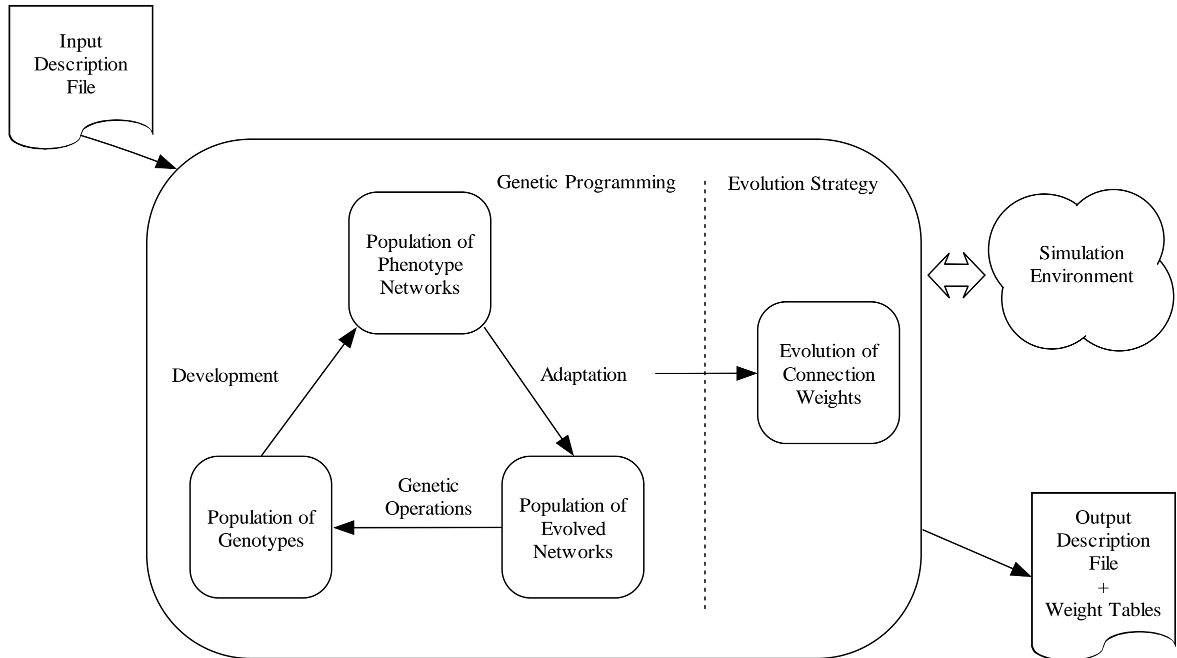


Figure 6.1: The iterative procedures used in this problem. The simulation environment is not considered as a part of the evolutionary system because the environment is problem-dependent.

its survival time. This problem shows typical conditions of reinforcement learning problems in the sense that the environment does not direct the agent as to which behavior would be optimal for its current situation, and the performance of the agent is determined by the overall result of its actions. Here, of course, consistent with the topic of this dissertation, an agent's behaviors were determined by an evolutionary neural network controller for the agent. This is because the simulated environment is a real-valued space and only partial sensory information is given to the agent. The continuous nature of the environment means that the number of possible states and actions is infinite in this case, and uncertainty in states makes it hard to apply conventional reinforcement learning algorithms that select among

discrete alternative actions at each state.

Figure 6.1 illustrates the overall evolutionary procedure used in this work. In the previous experiments of Chapters 4 and 5, the focus was on evolving architectures of neural networks, while finding appropriate weight values for the architecture has not been the main target of evolution. In previous chapters, initial weights were assigned and then they were changed during the evolutionary process by using supervised learning methods derived using gradient-descent in error space for actual weight training. Although such a hybrid method of evolutionary and local search / learning algorithms has been supported by many excellent experimental results ([Yao, 1999] for discussion), here the optimal connection weights are sought by evolution as the gradient descent information needed for supervised learning is not available (i.e., target answers are not supplied by a teacher here).

Another issue in evolving connection weights is that most previous research has employed a direct encoding scheme in which the architecture and connection weights are evolved concurrently as one genome [Gruau, 1994; Maniezzo, 1994; Pujol and Poli, 1998; Stanley and Miikkulainen, 2002b; Yao and Liu, 1996], except for some weight mutation operations. In contrast, here the evolution of connection weights is separated from the evolution of network architectures by building a population of possible weights for each architecture. For each phenotype network individual, a set of candidate weights are randomly generated (the right block in Figure 6.1). The performance of each network architecture combined with each weight vector in the corresponding weight population is evaluated through the simulation environment, and the best weight set is evolved using an evolution strategy [Beyer, 2001], some-

thing which will be explained in detail in Section 6.3. Then the evolutionary process for the network architecture follows, producing the next generation of genotypes.

To summarize, this chapter extends the results presented in previous chapters by showing that a descriptive encoding system can extend to 1) reinforcement learning; and 2) training the neural network controllers of simulated agents that operate in an external environment. Further, while network architectures are still evolved using genetic programming, 3) the learning step is replaced with a separate evolutionary process that acquires a good set of weights for each evolved architecture. This second evolutionary process occurs each generation of the genetic programming procedure (i.e., it is rested inside of each GP cycle), and is based on evolution strategies. Other changes from the experiments of previous chapters are: 4) the main population of architectures is divided into a group of subpopulations to maintain diversity of network architectures; and 5) crossover is used during the reproduction process. These changes are synergistic in terms of evolving highly fit networks, as will be seen from detailed comparison results in Section 6.4.

6.2 Simulation Environment

The simulation environment shown in Figure 6.2 consists of a two-dimensional, real-valued rectangular space (100.0 by 100.0) where each side is connected with the opposite side, implementing "periodic boundary conditions" (i.e., like a torus). At the beginning of each simulation, a predefined number of predators and food sites are randomly placed throughout the environment, and a single agent is located in the

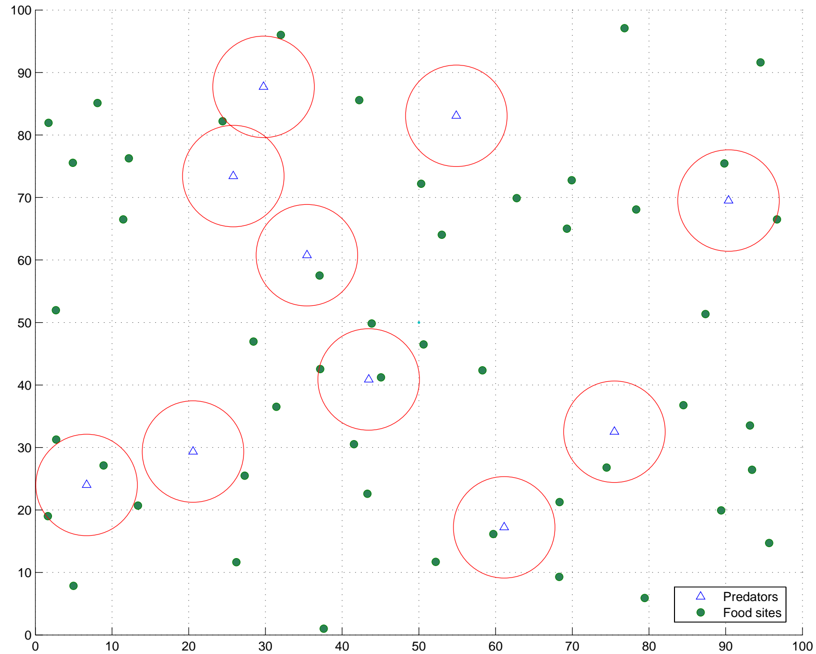


Figure 6.2: An initial configuration of the simulation environment with 10 predators and 50 food sites. Each circle around a predator approximately indicates the distance that the predator can “see” (radius 7.0). The agent is located in the center of the environment when the simulation begins, but is not depicted in this figure. In order to get a food resource, the agent should reduce its velocity (to < 0.5) and collide against the food, which is shown as dots each with a radius of 0.5.

Table 6.1: Predetermined Properties for Predators and Agent.

	Max. Visibility	Visible Angle	Max. Acceleration	Max. Velocity	Max. Rotation
Predators	7.0	2π	2.0	6.0	$\pi/2$
Agent	7.0	π	2.0	7.0	$\pi/2$

center of the environment. The agent is initially given a certain amount of "energy", and when this runs out, the agent will "die". The agent should evolve to run from predators if the predators find and chase the agent, and should forage for food sites as the initial energy level given to the agent is not sufficient for it to survive the simulation duration (500 time steps). The fitness of the agent is determined by its final energy level when the simulation ends and how long it survived. Figure 6.2 shows an initial simulation environment with a typical experimental configuration (10 predators and 50 food sites).

Predators are non-evolving, non-energy consuming entities whose behavior is decided by three internal states: resting, searching, and chasing (adapted from [Reggia et al., 2001b]). In the resting state, predators do not move nor do they change their orientation even if they find the agent within their visibility range (a distance of 7.0). All predators initially start with the resting state, in order to give the agent some chance to find predators and avoid them if they are initially located very close to the agent. Predators can "see" the agent in any direction if it is within their visibility range, while the agent can only perceive objects in the forward direction. After a predefined time step, resting predators enter the searching state in which they move around with a fixed velocity and random direction. If a searching

predator finds the agent, it enters the chasing state. Otherwise, it goes back to the resting state after a predefined time step. In the chasing state, predators increase their velocity and move directly towards the agent. The maximum acceleration in one time step for predators is the same as that of the agent, but the maximum velocity is set to be slightly lower than the agent's maximum velocity (predefined properties of the agent and predators are summarized in Table 6.1). If a chasing predator fails to catch the agent within five time steps, or if it loses the agent beyond its range of sight, it goes back to the resting state. The total number of predators is fixed before a simulation begins, and no predators die or are replaced with new ones during the simulation.

Food sites are fixed, randomly selected locations that have energy resources for the agent. Each site starts with a predefined amount of food (10), whose level is the maximum amount that a food site can store. When an agent moves close to a food site with a low velocity (< 0.5), it can consume one food unit per time step and the energy level of the agent will be increased by one. Sites may be temporarily depleted after all units in the site are consumed, but the food level will be restored after time (one food unit per ten time steps). No new food sites are generated.

The agent is the only entity that will adapt in this environment, through the evolution of the neural network controller which determines its behavior. This controller works as the "brain" of the agent producing motor signals as output, and the movement of the agent is simulated in the virtual environment according to these signals. The agent can perceive predators and food sites within its visibility range, but the range is limited to π degrees in its moving direction (i.e., the agent

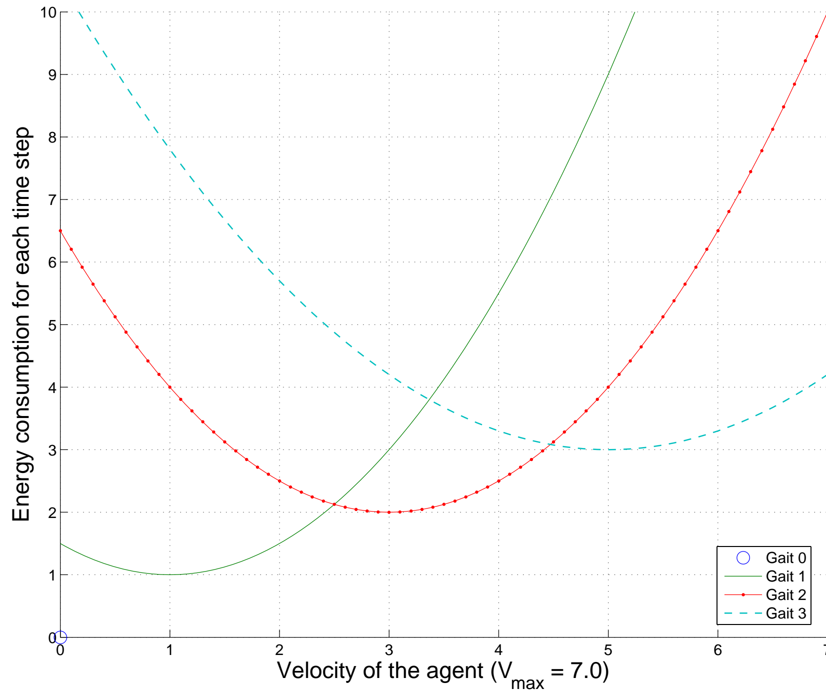


Figure 6.3: An agent's energy consumption model for each gait type and velocity. A fixed energy unit per time step (basal metabolic rate) is added when calculating the exact amount of energy consumption.

can't see predators chasing it from behind). If there are multiple predators within the agent's range of view, sensory information of the closest predator is provided to the agent, regardless of the internal state of the predator. The overall goal of an agent is to survive the full simulation time with as high an energy level as possible. An agent starts with a predefined energy level (100 units), and movement of the agent costs energy. In order to make this energy consumption model more realistic, a simplified notion of energy use was adopted, inspired by what occurs with animal gaits. While the factors that lead an animal to choose a specific gait are not fully

understood, the hypothesis that natural gaits are adjusted to require the minimal energy expenditure has been widely accepted [Hoyt and Taylor, 1981]. Following this hypothesis, four gait types are defined that have different energy consumption curves as depicted in Figure 6.3. For each time step t, the agent should choose a desired gait type (G_d^t), velocity (${V_d}^t$), and direction (θ_d^t). When the agent selects gait type 0, the motor signal for the desired velocity ${V_d}^t$ is ignored and the agent stays or rotates (i.e., changes its orientation by θ_d^t) in the current location with zero energy loss per time step due to movement per se. With the other three gait types, the agent's movement is simulated as follows:

- $G^{\,t+1} \quad = \quad G_d^t$
- $\Delta\, V^{\,t+1} \quad = \quad \mathrm{MIN}(\ |{V_d}^t - V^{\,t}|, \text{max_acceleration}) \cdot \mathrm{SIGN}\ ({V_d}^t - V^{\,t})$
- $\Delta\, \theta^{\,t+1} \quad = \quad \mathrm{MIN}(\ |\theta_d^t - \theta^{\,t}|, \text{max_rotation}) \cdot \mathrm{SIGN}\ (\theta_d^t - \theta^{\,t})$
- $\Delta\, location^{\,t+1} \quad = \quad (1 - |\Delta\theta^{\,t+1}|/\pi) \cdot (V^{\,t+1}) \cdot (cos\ \theta^{\,t+1},\ sin\ \theta^{\,t+1})$

where $location^{\,t+1}$ means two-dimensional position of the agent in the environment at time step $t+1$, and $(1 - |\Delta\theta^{\,t+1}|/\pi)$ indicates the cost of rotation (i.e., if the agent turns π, it cannot move at all at this time step). As shown in Figure 6.3, these quadratic energy consumption curves make only one gait type be reasonably acceptable in most velocity intervals. For example, when ${V_d}^t = 1.0$, only gait type 1 would be energy efficient. Therefore choosing the right gait type according to the desired velocity is essential to minimize the energy consumption level. Of course, the agent has no a priori knowledge of such information, which must be discovered by the evolutionary process. Another source adopted for consuming energy is based on the idea of basal metabolic rate. Specifically, each agent spends some fixed energy

units per time step regardless of its movement, and this baseline consumed energy is taken to be proportional to the size of the neural network controller. As the fitness of the agent depends on the final energy level, this latter restriction works as a parsimony factor that favors smaller networks. The simulation may end earlier than the maximum number of time steps if the agent is caught by a predator, or the energy level of the agent falls to zero.

In order to compare the performance of the evolved agent (i.e., as a control measure), a non-evolving agent is implemented whose behavior is predefined as follows: This rule-based agent starts in a searching state, where it moves randomly with a fixed, minimal velocity (1.0). When the agent sees a predator, it goes into an escaping state. In this state, the agent tries to avoid the observed predator with the maximum velocity and the corresponding gait type for three time steps. This means that it has a short-term memory of the direction of the predator that was seen previously, so the agent continues to move in the opposite direction as the predator even if it can't see the chasing predator anymore. When the agent observes a food site and there is no visible predator, it approaches the food site and stays there to gain energy units. This foraging state holds until the agent sees a predator (change to the escaping state), or the current food site is depleted (revert to the searching state).

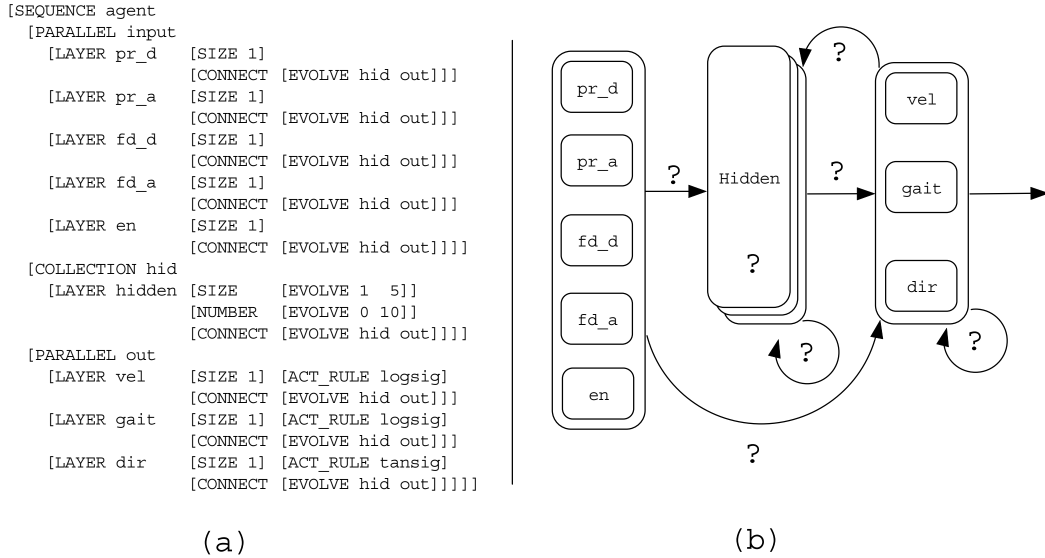


Figure 6.4: (a) the network description file used for evolving the network controller; (b) a sketch of the space of neural network architectures corresponding to the description file in (a). See text for details

6.3 Encoding Details

Figure 6.4a shows the network part of the description file used for this problem. The input to the network controller consists of local sensory input and the current energy level (labeled as *en*). For each time step, the simulation environment provides the information of the closest predator and food site visible to the agent, in a format of the normalized distance (labeled as *pr_d* and *fd_d*) and relative angle (labeled as *pr_a* and *fd_a*). Each of these five information sources separately takes a layer in the *input* network, and these layers can evolve to have connections to any of the hidden or output layers. As explained earlier, choosing the appropriate velocity based on the current sensory input and the energy level, and finding the relationship between

gait types and velocity, are essential to get higher fitness, but this description file does not specify such information to the system a priori.

The COLLECTION hidden network is an arbitrary structure of zero to ten layers that can be connected to themselves as well as to the output, so it might build a recurrent network like an Elman network, for example. The output subnetwork determines the desired velocity ${V_d}^t$, direction θ_d^t, and gait type G_d^t of the agent. These output values can also be fed back to the hidden layers or to each other, making a Jordan type recurrent network. From this network description, all possible architectures of up to ten hidden layers (and up to five nodes per each hidden layer) can be generated in theory, from a partial linear network where only some of the input layers are directly connected to output layers, to a fully recurrent network between hidden and output layers. Figure 6.4b illustrates the overall search space of the possible network architectures.

Figure 6.5 specifies various evolutionary options used in this problem. The number of neural network controllers tested in a generation is 100, and the maximum generation is empirically set as 500. As indicated, I divide this population into five subpopulations in which individuals compete with others only in the same group. This idea of separating individuals into groups, called an "island (migration) model" [Wright, 1932; Eldredge and Gould, 1972], has been applied in neuroevolution and other application in order to maintain the diversity of the whole population and to facilitate parallel implementation of the evolutionary process (e.g. [Martin et al., 1997]). The former is of interest in this experiment. A variation used here is that individuals are initially put into subgroups based on one of their network prop-

```
[EVOLUTION
   [MAX_POPLUATION 100]      [MAX_GENERATION 500]
   [SUB_POPULATION 5]        [CLUSTER num_weights]
   [MIGRATION_EPOCH 50]      [MIGRATION_MAX 1]
   [WEIGHT_EVOLUTION es]     [ES_MAX_GENERATION 1000]
   [ES_TYPE (100 , 200 (1 + 1)^num_weights)]
   [FITNESS external]
   [ELITISM 0] [SELECTION tournament] [TOURNAMENT_POOL 3]
   [CROSSOVER_PROB 0.7]      [MUTATION_PROB 0.1]
]
```

Figure 6.5: The evolutionary parameter portion of the description file for this problem.

erties, not randomly like in typical island model implementations. When the first genotypes are generated, individuals are sorted and subgrouped by the total number of node-to-node connections (shown in CLUSTER property). As I have adopted an energy consumption model proportional to the network size and the system initially generates all possible types of architectures allowable under the network description file with an equal probability, these subpopulations work as "islands" to make each individual compete against others of similar size, at least for the initial generations. Note that this size measure is not biased for modular networks, as it only counts node to node connections, not considering how the nodes are organized as layers. When a generation reaches the migration epoch (currently set to 50), the best individual in each subpopulation moves to another "island" that is randomly selected, and it participates in the reproduction process of the subpopulation to which it migrated.

For the evolution of connection weights, a nested evolution strategy algorithm [Beyer and Schwefel, 2002] is used. As shown in ES_TYPE option, this can be

described in the following format:

$$[\,\mu' \,,\ \lambda' \ \ (\mu\ +\ \lambda)^{\gamma}\,]$$

where μ' parent population of $(\mu\ +\ \lambda)$ individual strategies generate λ' offspring-populations, and γ means the number of isolating generations during which the inner $(\mu\ +\ \lambda)$ strategies run without any communication or interruption. For each phenotype network architecture, 100 candidate weight sets (initially random) and evolution strategy parameters generates 200 offspring weight sets by mutation, and these offspring weight sets evolve in parallel for γ (total number of weight values in this network) generations, following a $(1 + 1)$ evolution strategy. Then only the 100 best candidate weight sets are selected and this cycle repeats for 1000 generations. Finally, the best weight set is used in the fitness calculation for the current phenotype.

Fitness is declared as external in the description file (Figure 6.5) because it is calculated from outside of the system, not utilizing a network's own properties as in the previous chapters. For each phenotype network, the system calls a *get_fitness_external* () function which is assumed to be included in the simulation environment implementation, and this is the only function that the system interacts with the environment implementation. Although the main evolutionary system and the simulation environment should be compiled together, this separation of fitness function keeps the system problem-independent, as the fitness in this experiment is not directly related to the output of the networks. The fitness function of the network controllers in the specific application considered here is based on the energy

efficiency of the agent and how long it survived. The energy efficiency is defined as the ratio of the total number of obtained energy units to the total consumed energy during the simulation. Although this energy measure seems to clearly match with the aim of this experiment, its value may have a bias in favor of shorter simulation times (i.e., a relatively high fitness value will be given when simulations stop early before accumulating reliable statistics). For example, if a predator catches the agent in a very early stage, the agent may get a higher fitness value because the consumed energy level of the agent is lower than that of the other agents that survived longer time steps. In order to compensate for this bias, the ratio of time steps (the maximum is currently set as 500 steps) that the current agent survived is included as another fitness measure. The detailed definition of fitness for the ith neural network controller is as follows:

$$Fitness^i = \frac{Obtained \quad Energy^i}{Consumed \ Energy^i} \cdot \frac{Simulation \ Time^i}{Simulation \ Time^{Max}}$$

In the reproduction procedures, a tournament based selection method has been used with a pool size of three. Crossover probability is set as 0.7, and any type of mutation operations defined in the descriptive encoding system can occur randomly with a probability of 0.1. No best networks are automatically transferred into the next generation (i.e., there is no elitism).

6.4 Experimental Results

Ten runs each were done for a varying number of predators and food sites, as the performance of the agent depends on the configuration of the simulation

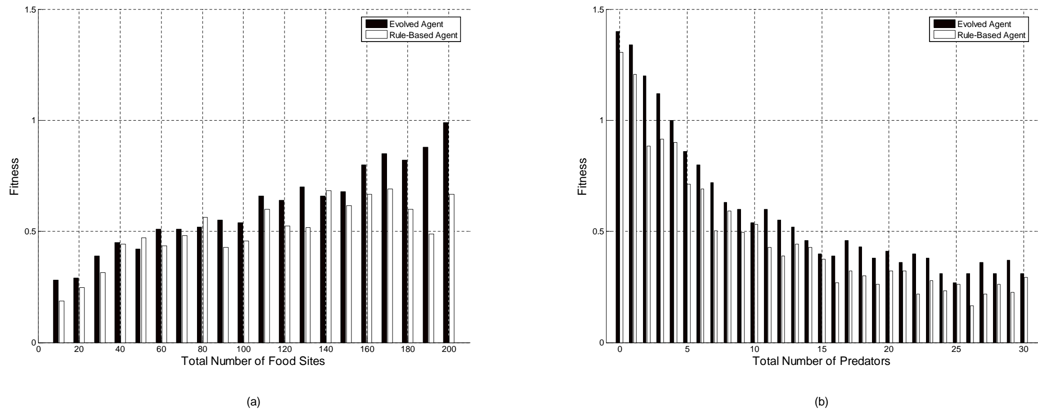


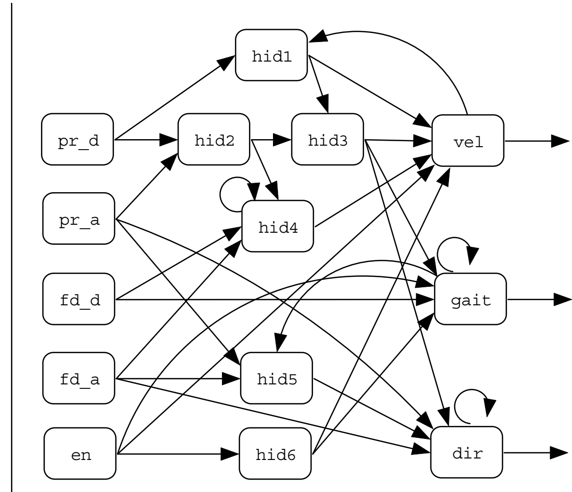
Figure 6.6: Average fitness of the best evolved agents compared with a non-adaptive, rule-based "control" agent as described earlier in the text. For each simulation configuration, the fitness value of the best network was average over ten independent runs. Part (a) shows the results with varying numbers of food sites while the number of predators was hold fixed at ten. In (b) the number of predators varied from zero to 30 while the number of food site was hold fixed at 100.

environment. The total number of predators was varied from zero to 30 in different simulations, and the number of food sites was varied from zero to 200 (in increments of ten). Figure 6.6 shows the averaged fitness of the best networks compared to the rule-based agent described in section 6.2. The evolved agents perform significantly better than the predefined agent (p values on t-test were less than 10^{-7} for all comparisons).

Figure 6.7 depicts a typical resultant network description which successfully incorporates required functionalities for the agent, and was the best agent controller when the number of food sites was 100, and the number of predators was 10. All

```
[SEQUENCE agent
  [PARALLEL input
    [LAYER pr_d   [SIZE 1]  [CONNECT hid1 hid2]]
    [LAYER pr_a   [SIZE 1]  [CONNECT hid2 dir ]]
    [LAYER fd_d   [SIZE 1]  [CONNECT hid4 gait]]
    [LAYER fd_a   [SIZE 1]  [CONNECT hid4 hid5 dir]]
    [LAYER en     [SIZE 1]  [CONNECT hid6 vel gait]]]
  [PARALLEL hid
    [LAYER hid1   [SIZE 1]  [CONNECT hid3 vel]]
    [SEQUENCE
      [LAYER hid2 [SIZE 2]  [CONNECT hid3 hid4]]
      [LAYER hid3 [SIZE 2]  [CONNECT vel gait dir]]]
    [PARALLEL
      [LAYER hid4 [SIZE 1]  [CONNECT hid4 vel]]
      [LAYER hid5 [SIZE 1]  [CONNECT dir]]
      [LAYER hid6 [SIZE 3]  [CONNECT vel gait]]]]
  [PARALLEL out
    [LAYER vel    [SIZE 1] [ACT_RULE logsig]
                  [CONNECT hid1]]
    [LAYER gait   [SIZE 1] [ACT_RULE logsig]
                  [CONNECT hid5 gait]]
    [LAYER dir    [SIZE 1] [ACT_RULE tansig]
                  [CONNECT dir]]]]
```

(a)



(b)

Figure 6.7: (a) An example of a resultant network description file when the simulation configuration consisted of 10 predators and 100 food sites. The evolved properties are in bold font. (b) Depicted network architecture.

output nodes have recurrent connections that work as memory for their output state. Specifically, in this case the velocity node has a recurrent connection to a hidden layer (labeled as *hid1*) that also processes predator input information, and another hidden layer *hid4* with a self-connection passes food site information to the velocity node. Each of the gait and direction node has a self-connection that stores the previous output value. Evolution also discovered a way of compensating for the agent's restricted visibility range, which was not expected before the experiment. A recurrent connection from the gait node to *hid5* layer triggers firing of this layer when gait type is either 0 or 1. This means that the agent can change its orientation constantly when it stays near a food site or moves with a relatively low velocity, looking in all directions as it rotates for approaching danger (predators).

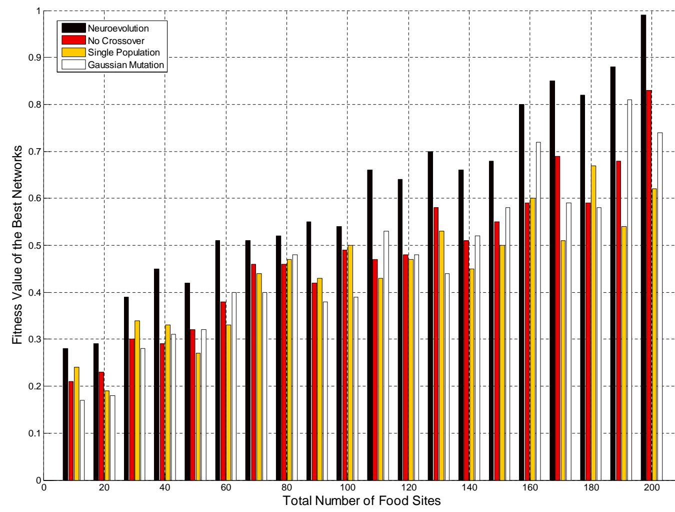


Figure 6.8: Fitness of best network controllers over varying number of food sites. The number of predators is fixed at 10.

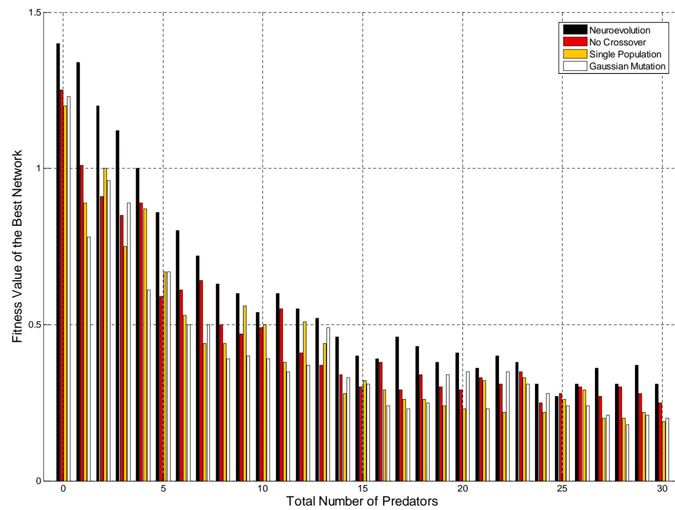


Figure 6.9: Fitness of best network controllers over varying number of predators. The number of food sites is fixed at 100.

Table 6.2: Summarized Results with Compared Systems

	Neuroevolution	No Crossover	Gaussian Mutation	Single Population
Avg. Fitness	0.578	0.473	0.427	0.429
p value	-	1.45 x 10^{-9}	1.05 x 10^{-7}	7.64 x 10^{-8}

The above results were based on the use of a nested evolution strategy for evolving connection weights, an island model for partitioning initial population, and crossover operations. To examine how each of these components of the system really contributed to finding the best networks, the comparative performance of the system was assessed in similar simulations that omitted each component. Table 6.2 summarizes the averaged fitness over varying number of food sites and the corresponding p values from a t-test. Figures 6.8 and 6.9 depict the average fitness value of best networks under varying number of predators or food sites. The label "neuroevolution" means the results from the system with all components present, while "No Crossover", "Single Population", and "Gaussian Mutation" means that the system lacks crossover operation, the island model, or the nested evolution strategy component, respectively. Each of these situations will be discussed below.

6.4.1 Nested Evolution Strategy vs. Gaussian Mutation

The nested evolution strategy adopted for weight evolution was compared with a variation of Gaussian mutation method introduced in [Saunders et al., 1993; Jansen and Wegener, 2006]. While the strategy parameters (i.e., mutation strength in the inner strategies) in the nested ES are controlled by the $1/5^{th}$ rule and outer strategy, Gaussian mutation selects mutation strength based on the fitness of the current

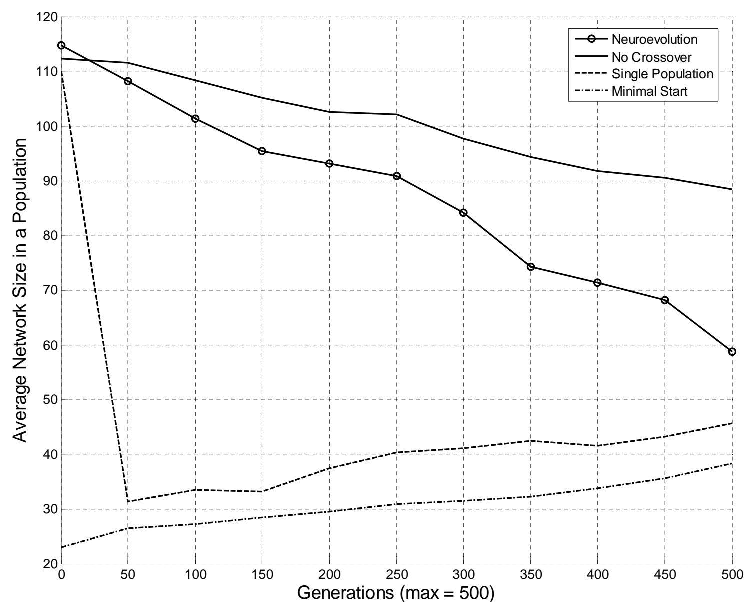


Figure 6.10: Average network size in terms of the number of connection weights, plotted versus the generation of an evolutionary process

network, and it does not adapt to the environmental changes. For all environmental configurations, the nested evolution strategy shows significantly better results than the Gaussian mutation version.

6.4.2 Island Model vs. Single Population

As shown in Figure 6.8 and 6.9, no result in the single population case showed better performance than the system with the Island model. Figure 6.10 illustrates typical changes in the network size over generations, when there are ten predators and 100 food sites initially placed in the environment. While the neuroevolution approach used in this chapter gradually decreases the size of networks while optimizing their architectures, the single population model without any speciation method

is reduced to some minimal networks in the very early stages and slowly adds up their network size in the later generations. Another possible way of generating the initial population is to start minimally (i.e., all input nodes are connected directly to the output nodes without any hidden layers) and sharing fitness among similar networks, as described in [Stanley and Miikkulainen, 2002b]. But the network size in this approach (labeled as "Minimal start" in Figure 6.10) also increases very slowly, and the performance of the most fit network (0.41) was significantly lower than that found with the Island model. Considering problem domains where only simple architectures are involved, I believe that this result does not conflict with their claims and that these two approaches complement each other. However, given such information is known a priori, one can reduce the initial search space by modifying the description file, making the model described in this chapter competitive with the minimal starting model.

6.4.3 Crossover vs. Mutation

The role of crossover has been controversial in neuroevolution as well as among the evolutionary computation community in general [Spears, 1993]. For example, Angeline et al. [1994] claimed "the prospect of evolving connectionist networks with crossover appears limited" due to the deceptive [Goldberg, 1989a] nature of network representation. However, there have been successful applications using crossover operations to evolve neural networks [Pujol and Poli, 1998; Stanley and Miikkulainen, 2002b], and here another experimental result relating to the value of crossover is

considered. Compared with the mutation only system (labeled as "No Crossover" in Figure 6.8-6.10), the performance of the system using crossover operations is significantly better and Figure 6.10 illustrates that it also helps to compress the overall size of search space faster.

6.5 Discussion

In this chapter, the descriptive encoding system is extended to address reinforcement learning problems. The main issue in reinforcement learning problems is that there is no teacher instructing details of beneficial actions, as many of real world learning problems would be. It becomes even more complicated if the problem involves a real-valued search space, such that conventional learning algorithms contrived for this problem class might not produce efficient solutions. In order to tackle this real-valued reinforcement learning problem, a separate evolutionary process for evolving connection weights was added to the main system of evolving architectures which is based on genetic programming. It was shown that the evolved recurrent networks outperform a rule-based, predesigned agent under various environmental configurations, and that evolution can discover behaviors compensating for the agent's given limitations and incorporate it into appropriate network architectures.

Chapter 7

Disscussion

7.1 Contributions Revisited

Recent advances in neuroevolutionary methods have repeatedly been successful in creating innovative neural network designs [Alonso et al., 2007; Balakrishnan and Honavar, 2001; Cho and Shimohara, 1998; Gruau, 1995; Gruau et al., 1996; Lehmann and Kaufmann, 2005; Ruppin, 2002; Saravanan and Fogel, 1995; Yao, 1999; Liao and Tsao, 2006; Chong et al., 2005]. However, these successes have had little practical influence on the field of neural computation. I believe this is partially because of a dilemma: The general space of neural network architectures and methods is so large that it is impractical to search efficiently, yet attempting to avoid this problem by hand-crafting the evolution of neural networks on a case-by-case basis is very labor intensive and thus also impractical.

In this context, I explored the hypothesis that a high-level descriptive language can be used effectively to support the evolutionary design of a broad range of task-specific neural networks. This approach addresses the impracticality of searching the enormous general space of neural networks by allowing a designer to easily restrict the search space to architectures and methods that appear a priori to be relevant to a specific application, greatly reducing the size of the space that an evolutionary process must search. It also greatly reduces the time needed to create a network's

design by allowing one to describe the class of neural networks of interest at a very high level in terms of possible modules and inter-module pathways, rather than in terms of individual neurons and their connections. Filling in the "low level" details of individual networks in an evolving population is left to an automated developmental process (the neural networks are "grown" from their genetic encoding) and to well-established neural learning methods that create connection weights prior to fitness assessment.

It remains to be established how effective the approach described here will ultimately be in practice. In this dissertation I have presented experimental evaluation results suggesting that it can be very effective. It was shown that human-readable description files could guide an evolutionary process to produce near-optimal solutions in n-partition problems. Resultant networks typically showed independent pathways and minimal number of hidden nodes, supporting the hypothesis that maximizing network performance while minimizing cost would lead to emergence of modular neural networks. These results were accomplished without fundamental changes in the description file while the problem size was increased exponentially. By comparing the performance of resultant networks with those of fully connected networks, I demonstrated the need for searching the space of network architectures. In a temporal sequence generation problem, it was shown that this approach could not only create effective recurrent architectures, but that it could simultaneously indicate the tradeoffs in the costs of architectural features versus network performance via multi-objective evolution. This experiment also showed that how users' domain knowledge can be incorporated in the description file in order to reduce the search

space effectively. Several mixed architectures of two basic recurrent networks were found to be very efficient in this problem, and some of these networks outperformed both of Jordan and Elman networks, which was not expectable before this experiment. Finally, It was demonstrated that this evolutionary system can be applied to address reinforcement learning problems. While conventional reinforcement learning approaches do not prove to be successful in general when a problem involves real variables, this problem was solved by searching both of architecture space and connection weight space.

7.2 Future Directions

Substantial room remains for further research developing evolutionary methods for neural networks based on high-level descriptive languages. For example, the approach taken in this dissertation has focused primarily on computational issues involving artificial neural networks, and has not addressed many of the complexities related to evolution of biologically-realistic neural networks that are often studied in neuroscience (spiking neurons, multi-compartment neuron models, realistic time delays in neural transmission, etc.), leaving this as a fertile area for future work. The scalability of the descriptive encoding approach to larger problems is also an important issue that remains to be established. While I have argued in favor of scalability in terms of decoding time and genotype space complexity in this work (see Chapter 3), the validity of such arguments remains to be confirmed in practice. Further, one can envision a number of extensions derived from contemporary evo-

lutionary computation methods, such as allowing the co-evolution of subnetworks that form components of larger networks [Palacios-Durazo and Valenzuela-Rendón, 2004; Potter and de Jong, 2000; Monroy et al., 2006; Popovici and Jong, 2006; Garcia-Pedrajas et al., 2005]. Also, while I believe that the high-level language presented here will be intuitively understandable to most neural modelers after a tutorial explanation, this remains to be established through further use in practice and experimental validation.

Appendix - A Simplified Grammar for the Description File

```
<description>  := <network> <training> <evolution>

<network>      := [<net_type> <name> <sub_network>] | <layer>
<net_type>     := SEQUENCE  |  PARALLEL   | COLLECTION
<sub_network>  := <network> <sub_network> | <network>
<layer>        := [LAYER <name> <ly_prop_list>]
<ly_prop_list> := <ly_property> <ly_prop_list> | <ly_property>    |
                  <ly_prop_list><co_prop_list> |
<co_prop_list> := [<conn_list>] <co_prop_list> | [<conn_list>]
<conn_list>    := <co_property> <conn_list>    | <co_property>
<ly_property>  := [NUM_LAYER       <value>]    | [SIZE       <value>] |
                  [BIAS            <value>]    | [ACT_RULE   <ename>] |
                  [ACT_INIT        <value>]    | [ACT_MIN    <value>] |
                  [ACT_MAX         <value>]    | <co_property>        | ...
<co_proerty>   := [CONNECT         <ename>]    |
                  [CONNECT_INIT    <value>]    | [LEARN_RULE <ename>] |
                  [CONNECT_RADIUS  <value>]    | ...

<training>     := [TRAINING  <tr_prop_list>]
<evolution>    := [EVOLUTION <ev_prop_list>]
<tr_prop_list> := <tr_property> <tr_prop_list> | <tr_property>
<ev_prop_list> := <ev_property> <ev_prop_list> | <ev_property>
<tr_property>  := [TRAIN_DATA   <path>]        | [MAX_TRAIN  <value>] |
                  [TRAIN_METHOD <name>]        | ...
<ev_property>  := [FITNESS     <name> <ratio>] |
                  [SELECTION   <name> ]        | [TOURNAMENT_POOL <value>] |
                  [ELITISM     <value>]        | [MUTATION_PROB   <value>] |
                  [MAX_GENERATION <value>]     | [CROSSOVER_PROB  <value>] |
                  [MAX_POPULATION <value>}     | [STOP_CRITERIA   <name> ] | ...

<value>        := [EVOLVE <range_value>]       | [<range_value>] |
                  [EVOLVE <fixed_value>]       | <fixed_value>
<range_value>  := <fixed_value> <range_value>  | <fixed_value>
<fixed_value>  := <integer>        | <float>   | <literals>
<ename>        := [EVOLVE <names>] | [<names>] | <name>
<names>        := <names> <name>
<name>         := <literals>
<path>         := ‘‘<name>’’
<ratio>        := [<ratio_list>]
<ratio_list>   := <ratio_list> <ratio_item>
<ratio_item>   := <name>: <value>
```

Bibliography

Alonso, J. M., Alvarruiz, F., Desantes, J. M., Hernndez, L., Hernndez, V., and Molt, G. (2007). Combining neural networks and genetic algorithms to predict and reduce diesel engine emissions. *IEEE Transactions on Evolutionary Computation*, 11(1):46–55.

Angeline, P. J., Saunders, G. M., and Pollack, J. B. (1994). An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, pages 54–65.

Asadi, M. and Huber, M. (2007). Effective control knowledge transfer through learning skill and representation hierarchies. In Veloso, M. M., editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2054–2059.

Ashlock, D. (2006). *Evolutionary Computation for Modeling and Optimization*. Springer.

Bäck, T. (1994). Evolutionary algorithms: Comparison of approaches. In *Computing with Biological Metaphors*, pages 228–243. Champman and Hall.

Bäck, T. and Schwefel, H.-P. (1993). An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1):1–23.

Balakrishnan, K. and Honavar, V. (1995). Properties of genetic representations of neural architectures. In *Proceedings of the World Congress on Neural Networks (WCNN '95)*, pages 807–813.

Balakrishnan, K. and Honavar, V. (2001). Evolving neuro-controllers and sensors for artificial agents. In *Advances in the Evolutionary Synthesis of Intelligent Agents*, pages 109–152. MIT Press.

Banzhf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1997). *Genetic Programming An Introduction*. Morgan Kaufmann.

Bentely, P. J. and Corne, D. W. (2001). *Creative Evolutionary Systems*. Morgan Kaufmann.

Beyer, H.-G. (2001). *The theory of Evolution Strategies*. Springer, Berlin.

Beyer, H.-G. and Schwefel, H.-P. (2002). Evolution strategies - a comprehensive introduction. *Natural Computing*, 1:3–52.

Blum, A. L. and Rivest, R. L. (1992). Training a 3-node neural network is NP-complete. *Neural Networks*, 5(1):117–127.

Bonissone, P. P., Subbu, R., Eklund, N., and Kiehhl, T. R. (2006). Evolutionary algorithms + domain knowledge = real-world evolutionary computation. *IEEE Transactions on Evolutionary Computation*, 10(3):256–280.

Brown, M., Keynes, R., and Lumsden, A. (2001). *The Developing Brain*. Oxford University Press.

Caelli, T. M., Guan, L., and Wen, W. (1999). Modularity in neural computing. *Proceedings of IEEE*, 87:1497–1518.

Caelli, T. M., Squire, D. M., and Wild, T. P. J. (1993). Model-based neural networks. *Neural Networks*, 6(5):613–625.

Calabretta, R., Nolfi, S., Parisi, D., and Wagner, G. (2000). Duplication of modules facilitates the evolution of functional specialization. *Artificial Life*, 6:69–84.

Cho, S. (1997). Combining modular neural networks developed by evolutionary algorithm. In *Proceedings of 1997 IEEE International Conference on Evolutionary Computation*, pages 647–650.

Cho, S. and Shimohara, K. (1998). Evolutionary learning of modular neural networks with genetic programming. *Applied Intelligence*, 9:191–200.

Chong, S. Y., Tan, M. K., and White, J. D. (2005). Observing the evolution of neural networks learning to play the game of othello. *IEEE Transactions on Evolutionary Computation*, 9(3):240– 251.

Coello Coello, C. A. (2002). Evolutionary multi-objective optimization: A critical review. In Sarker, R., Mohammadian, M., and Yao, X., editors, *Evolutionary Optimization*, pages 117–146. Kluwer Academic Publishers, New York.

Crites, R. H. and Barto, A. G. (1996). Improving elevator performance using reinforcement learning. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems 8*, Cambridge, MA. The MIT Press.

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

De Garis, H. (1991). GenNETS: Genetically programmed neural nets using the genetic algorithm to train neural nets whose inputs and/or output vary in time. In *Proceedings of the International Joint Conference on Neural Networks (5th IJCNN'91)*, volume 2, pages 1391–1396, Singapore. IEEE.

Dill, F. A. and Deer, B. C. (1991). An exploration of genetic algorithms for the selection of connection weights in dynamical neural networks. In *Proceedings of the IEEE 1991 National Aerospace and Electronics Conference NAECON 91*, volume 3, pages 1111–1115, Dayton, OH. IEEE, New York, NY.

Dimond, S. J. and Blizard, D. A., editors (1977). *Evolution and Lateralization of the Brain*. New York Academy of Sciences.

Eldredge, N. and Gould, S. J. (1972). Punctuated equilibria: An alternative to phyletic gradualism. In Schopf, T., editor, *Models in Paleobiology*, pages 82–115. Freeman Cooper.

Elman, J. E. (1990). Finding structure in time. *Cognitive Science*, 14(2):179–211.

Ferdinando, A. D., Calabretta, R., and Parisi, D. (2001). Evolving modular architectures for neural networks. In French, R. and Sougné, J., editors, *Proceedings Sixth Neural Computation and Psychology Workshop Evolution, Learning, and Development*.

Fogel, D. B. (1991). *System Identification Through Simulated Evolution: A Machine Learning Approach to Modeling*. Ginn Press.

Fogel, D. B. (1995). *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE.

Fogel, L. J., Owens, A. J., and Walsh, M. J. (1966). *Artificial Intelligence Through Simulated Evolution*. John Wiley & Sons.

Franco, L. and Cannas, S. A. (2001). Generalization properties of modular networks: implementing the parity function. *IEEE Transactions on Evolutionary Computation*, 5:1306–1313.

Garcia-Pedrajas, N., Hervas-Martinez, C., and Ortiz-Boyer, D. (2005). Cooperative coevolution of artificial neural network ensembles for pattern classification. *IEEE Transactions on Evolutionary Computation*, 9(3):271– 302.

Giles, C. L., Miller, C. B., Chen, D., Sun, G. Z., Chen, H. H., and Lee, Y. C. (1992). Extracting and learning an unknown grammar with recurrent neural networks. In Moody, J. E., Hanson, S. J., and Lippmann, R. P., editors, *Advances in Neural Information Processing Systems 4*, pages 317–324. Morgan Kaufmann, Denver, CO.

Goldberg, D. E. (1989a). Genetic algorithms and walsh functions: I. a gentle introduction. *Complex Systems*, 3(2):129–152.

Goldberg, D. E. (1989b). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley.

Gordon, T. G. W. and Bentley, P. J. (2005). Bias and scalability in evolutionary development. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 83–90, New York, NY, USA. ACM Press.

Gruau, F. (1994). *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm*. PhD thesis, Laboratoire de l'Informatique du Parallilisme, Ecole Normale Supirieure de Lyon, France.

Gruau, F. (1995). Automatic definition of modular neural networks. *Adaptive Behavior*, 3:151–183.

Gruau, F., Whitley, D., and Pyeatt, L. (1996). A comparison between cellular encoding and direct encoding for genetic neural networks. In *Proceedings of the Sixth International Conference on Genetic Programming*. Stanford University Press.

Grushin, A. and Reggia, J. A. (2005). Evolving processing speed asymmetries and hemispheric interactions in a neural network model. *Neurocomputing*, 65:47–53.

Guan, L., Anderson, J. A., and Sutton, J. P. (1997). A network of networks processing model for image regularization. *IEEE Transactions on Neural Networks*, 8(1):169–174.

Happel, B. L. M. and Murre, J. M. J. (1994). Design and evolution of modular neural network architectures. *Neural Networks*, 7(6-7):985–1004.

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation*. Prentice Hall, Upper Saddle River, NJ.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press.

Hornby, G. S. (2004). Shortcomings with tree-structured edge encodings for neural networks. In *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 495–506. Springer-Verlag.

Hoyt, D. F. and Taylor, C. R. (1981). Gait and the energetics of locomotion in horses. *Nature*, 292:239–240.

Hussain, T. S. and Browse, R. A. (1998). Network generating attribute grammar encoding. In *IEEE International Conference on Neural Networks (IJCNN'98)*, volume I, pages 431–436, Anchorage, AK. IEEE.

Hussain, T. S. and Browse, R. A. (2000). Evolving neural networks using attribute grammars. *IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks*, pages 37–42.

Jacobs, R., Jordan, M., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive mixtures of local experts. *Neural Computation*, 3:79–87.

Jansen, T. and Wegener, I. (2006). On the local performance of simulated annealing and the (1+1) evolutionary algorithm. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 469–476, New York, NY, USA. ACM Press.

Jong, K. D. (2006). *Evolutionary Computation A Unified Approach*. MIT Press.

Jordan, M. I. (1986). Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Eighth Conference of the Cognitive Science Society*, pages 531–546. Erlbaum.

Jung, J.-Y. and Reggia, J. A. (2004a). A descriptive encoding language for evolving modular neural networks. In *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 519–530. Springer-Verlag.

Jung, J.-Y. and Reggia, J. A. (2004b). Evolving large-scale modular neural networks. In Poli, R., editor, *GECCO 2004 Workshop Proceedings*, Seattle, Washington, USA.

Jung, J.-Y. and Reggia, J. A. (2006). Evolutionary design of neural network architectures using a descriptive encoding language. *IEEE Transactions on Evolutionary Computation*, pages 676–688.

Kaelbling, L. P., Littman, M. L., and Moore, A. P. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285.

Kandel, E., Schwartz, J., and Jessel, T. (1991). *Principles of Neural Science*. Appleton and Lange.

Killackey, H. (1996). Evolution of the human brain: A neuroanatomical perspective. In Gazzaniga, M., editor, *The Cognitive Neurosciences*, pages 1243–1253. MIT Press.

Kitano, H. (1990). Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4(4):461–476.

Kitano, H. (1994). Neurogenetic learning: An integrated method of designing and training neural networks using genetic algorithms. *Physica D*, 75:225–238.

Knowles, J. and Corne, D. (2003). Properties of an adaptive archiving algorithm for storing nondominated vectors. *IEEE Transactions on Evolutionary Computation*, 7(2):100– 116.

Kohonen, T. (1982). Self-organizing formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69.

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

Koza, J. R. (1994). *Genetic Programming II*. MIT Press, Cambridge, MA.

Koza, J. R., Bennett, F., Andre, D., and Keane, M. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann.

Kumar, S. (2004). *Neural Networks: a classroom approach*.

Kung, S. Y. and Taur, J. S. (1995). Decision-based neural networks with signal/image classification applications. *IEEE Transactions on Neural Networks*, 6(1):170–181.

Lehmann, K. A. and Kaufmann, M. (2005). Evolutionary algorithms for the self-organized evolution of networks. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 563–570, New York, NY, USA. ACM Press.

Levitan, S. and Reggia, J. A. (2000). A computational model of lateralization and asymmetries in cortical maps. *Neural Computation*, 12:2037–2062.

Li, M., Azarm, S., and Aute, V. (2005). A multi-objective genetic algorithm for robust design optimization. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 771–778, New York, NY, USA. ACM Press.

Liao, G. C. and Tsao, T. P. (2006). Application of a fuzzy neural network combined with a chaos genetic algorithm and simulated annealing to short-term load forecasting. *IEEE Transactions on Evolutionary Computation*, 10(3):330–340.

Luke, S. and Spector, L. (1996). Evolving graphs and networks with edge encoding: Preliminary report. In Koza, J. R., editor, *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996*, pages 117–124, Stanford University, CA, USA. Stanford Bookstore.

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

Martin, W. N., Lienig, J., and Cohoon, J. P. (1997). Island (migration) models: evolutionary algorithms based on punctuated equilibria. In Bäck, T., Fogel, D. B., and Michalewicz, Z., editors, *Handbook of Evolutionary Computation*, pages 101–124. Institute of Physics Publishing and Oxford University Press, Bristol, New York.

Mataric, M. J. (1994). Reward functions for accelerated learning. In *Proceedings of the 11th International Conference on Machine Learning*, pages 181–189.

Mehrotra, K., Mohan, C. K., and Ranka, S. (1997). *Elements of Artificial Neural Networks*. MIT Press, Cambridge, MA.

Miller, G. F., Todd, P. M., and Hegde, S. U. (1989). Designing neural networks using genetic algorithms. In *Proceedings of 3rd International Conference on Genetic algorithms (ICGA89)*, pages 379–384.

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

Møller, M. F. (1993). A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6(4):525–533.

Monroy, G. A., Stanley, K. O., and Miikkulainen, R. (2006). Coevolution of neural networks using a layered pareto archive. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 329–336, New York, NY, USA. ACM Press.

Montana, D. and Davis, L. (1990). Training feedforward neural networks using genetic algorithms. In *Proceedings of 11th International Joint Conference on Artificial Intelligence*, pages 370–374. Morgan Kaufmann.

Moriarty, D. E., Schultz, A. C., and Grefenstette, J. J. (1999). Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, 11:199–229.

Mountcastle, V. (1998). *The Cerebral Cortex*. Harvard University Press.

Paine, R. W. and Tani, J. (2004). Evolved motor primitives and sequences in a hierarchical recurrent neural network. In et al, K. D., editor, *GECCO (1)*, volume 3102 of *Lecture Notes in Computer Science*, pages 603–614. Springer.

Palacios-Durazo, R. A. and Valenzuela-Rendón, M. (2004). Similarities between co-evolution and learning classifier systems and their applications. In et al, K. D., editor, *Genetic and Evolutionary Computation – GECCO-2004, Part I*, volume 3102 of *Lecture Notes in Computer Science*, pages 561–572, Seattle, WA, USA. Springer-Verlag.

Pérez-Ortiz, J., Calera-Rubio, J., and Forcada, M. (2001). A comparison between recurrent neural architectures for real-time nonlinear prediction of speech signals. In Miller, D., Adali, T., Larsen, J., Hulle, M. V., and Douglas, S., editors, *Neural Networks for Signal Processing XI, Proceedings of the 2001 IEEE Neural Networks for Signal Processing Workshop (NNSP 2001)*, pages 73–81. IEEE Signal Processing Society.

Popovici, E. and Jong, K. D. (2006). The effects of interaction frequency on the optimization performance of cooperative coevolution. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 353–360, New York, NY, USA. ACM Press.

Porto, V. W. (1997). Evolutionary programming. In Bäck, T., Fogel, D. B., and Michalewicz, Z., editors, *Handbook of Evolutionary Computation*, pages 54–64. Institute of Physics Publishing and Oxford University Press, Bristol, New York.

Potter, M. and de Jong, K. (2000). Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1):1–29.

Pujol, J. C. F. and Poli, R. (1998). Evolving the topology and the weights of neural networks using a dual representation. *Applied Intelligence*, 8(1):73–84.

Radio, M. J., Reggia, J. A., and Berndt, R. S. (2001). Learning word pronunciations using a recurrent neural network. In *Proceedings of International Joint Conference on Neural Networks (IJCNN '01)*, volume 1, pages 11–15.

Ranganath, H. S., Kerstetter, D. E., and Sims, S. R. F. (1995). Self partitioning neural networks for target recognition. *Neural Networks*, 8(9):1475–1486.

Reggia, J. A., Goodall, S., Shkuro, Y., and Glezer, M. (2001a). The callosal dilemma: Explaining diaschisis in the context of hemispheric rivalry via a neural network model. *Neurological Research*, 23:465–471.

Reggia, J. A., Schulz, R., Wilkinson, G., and Uriagereka, J. (2001b). Conditions enabling the evolution of inter-agent signaling in an artificial world. *Artificial Life*, 7(1):3–32.

Riedmiller, M. and Braun, H. (1993). A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Proceedings of 1993 IEEE International Conference on Neural Networks*, volume 1, pages 586–591.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). *Parallel Distributed Processing*. MIT Press, Cambridge, MA.

Ruppin, E. (2002). Evolutionary autonomous agents: A neuroscience perspective. *Nature Reviews Neuroscience*, 3(2):132–141.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):211–229.

Saravanan, N. and Fogel, D. (1995). Evolving neural control systems. *IEEE Expert*, 10:23–27.

Saunders, G. M., Angeline, P. J., and Pollack, J. B. (1993). Structural and behavioral evolution of recurrent networks. In Cowan, J. D., Tesauro, G., and Alspector, J., editors, *Neural Information Processing System*, pages 88–95. Morgan Kaufmann.

Schaal, S. and Atkeson, C. (1994). Robot juggling: An implementation of memory-based learning. *IEEE Control Systems*, 14:57–71.

Schlessinger, E., Bentley, P. J., and Lotto, R. B. (2006). Modular thinking: evolving modular neural networks for visual guidance of agents. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 215–222, New York, NY, USA. ACM Press.

Schlosser, G. and Wagner, G. (2004). *Modularity in Development and Evolution*. University of Chicago Press.

Schwefel, H. (1981). *Numerical Optimization of Computer Models*. Wiley.

Sejnowski, T. and Rosenberg, C. (1987). Parallel networks that learn to pronounce english text. *Complex Systems*, 1:145–168.

Shkuro, Y. and Reggia, J. A. (2003). Cost minimization during simulated evolution of paired neural networks leads to asymmetries and specialization. *Cognitive Systems Research*, 4(4):365–383.

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

Spears, W. M. (1993). Crossover or mutation? In Whitley, L. D., editor, *Foundations of Genetic Algorithms 2*, pages 221–237. Morgan Kaufmann, San Mateo, CA.

Srinivas, M. and Patnaik, L. M. (1991). Learning neural network weights using genetic algorithms- improving performance by search-space reduction. In *1991 IEEE International Joint Conference on Neural Networks*, volume 3, pages 2331–2336, Singapore. IEEE.

Stanley, K. O. and Miikkulainen, R. (2002a). Efficient reinforcement learning through evolving neural network topologies. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, USA, 9-13 July 2002*, pages 569–577. Morgan Kaufmann.

Stanley, K. O. and Miikkulainen, R. (2002b). Evolving neural network through augmenting topologies. *Evolutionary Computation*, 10(2):99–127.

Sutton, R. S. (1986). 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.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning An Introduction*. MIT Press.

Tesauro, G. (1994). Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219.

Tooby, J. and Cosmides, L. (2000). Toward mapping the evolved functional organization of mind and brain. In Gazzinga, M., editor, *The New Cognitive Neurosciences*, pages 1167–1178. MIT Press.

Wagner, G. (1995). Adaptation and the modular design of organisms. In Moran, F., Moreno, A., Merelo, J., and Chacon, P., editors, *Advances in Artificial Life*, pages 317–328. Springer.

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

Whitley, D., Gruau, F., and Pyeatt, L. (1995). Cellular encoding applied to neurocontrol. In *Proceedings of the Sixth International Conference on Genetic Algorithms*.

Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82.

Wolpert, D. H. and Macready, W. G. (2005). Coevolutionary free lunches. *IEEE Transactions on Evolutionary Computation*, 9(6):721– 735.

Wright, S. (1932). The roles of mutation, inbreeding, crossbreeding, and selection in evolution. In *Proceedings of the VI International Congress of Genetics*, volume 1, pages 356–366.

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447.

Yao, X. and Liu, Y. (1996). Evolutionary artificial neural networks that learn and generalise well. In *Proceedings of the 1996 IEEE International Conference on Neural Networks*, pages 159–164.

Yuen, S. Y. and Chenung, B. K. S. (2006). Bounds for probability of success of classical genetic algorithm based on hamming distance. *IEEE Transactions on Evolutionary Computation*, 10(1):1–18.

Zitzler, E. and Thiele, L. (1999). Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271.