Efficient Evolution of Neural Networks through Complexification

Kenneth O. Stanley

Report AI-TR-04-314 August 2004

kstanley@cs.utexas.net
http://www.cs.utexas.edu/users/kstanley/

Artificial Intelligence Laboratory
The University of Texas at Austin
Austin, TX 78712

The Dissertation Committee for Kenneth Owen Stanley
certifies that this is the approved version of the following dissertation:

Efficient Evolution of Neural Networks through Complexification

Committee:

Risto Miikkulainen, Supervisor

Kenneth A. De Jong

Joydeep Ghosh

Benjamin J. Kuipers

Raymond J. Mooney

Bruce W. Porter

Efficient Evolution of Neural Networks through Complexification

by

Kenneth Owen Stanley, B.S.E., M.S.C.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2004

Acknowledgments

My advisor, Risto Miikkulainen, has been the greatest role model, source of inspiration and guidance, and teacher throughout my education. None of my success would have been possible without his years of painstaking efforts to teach me how to test, present, and articulate ideas.

I am also most grateful to my mother who spent many years of my childhood reading to me and teaching me to be curious, to my father who took time to talk to me about science and philosophy, and to my sister Sharon who has always patiently listened to my ideas.

My first experience in writing a large scale neuroevolution system before NEAT was at Hewlett-Packard Laboratories, and I would like to thank Jaap Suermondt, Evan Kirshenbaum, and Bin Zhang for that experience.

I would also like to thank everyone at the DMC Lab who made the NERO project possible. In particular I am grateful to Alex Cavalli, Aliza Gold, Matt Patterson, and Aaron Thibault for having the courage to initiate such a project. Bobby Bryant deserves great credit for acting as Lead Programmer and keeping the project afloat. The project's volunteers, many of whom are undergraduates, also deserve my thanks for helping to make my idea a reality: Michael Chrien, Ryan Cornelius, Jonathan Perry, Brad Walls, and other art and programming volunteers, including those who have recently joined the project. I am especially grateful to Philip Flesher, who served for a time as Lead Programmer, and spent many hours working with me to get rtNEAT integrated with Torque.

I am also thankful to those who have provided friendship, support, and inspiration including Adrian Agogino, David Callner, Catie Chang, Harold Chaput, Jason Coletta, Debra Davis, Mark Gengenbach, Chris Glover, Tino Gomez, Nate Kohl, Scott McNeil, Jill Nickerson, Lina Panackal, Joseph Reisinger, Joe Rossetti, Jeremy Sears, Mike Shao, Riccardo Signorelli, Alex Strehl, Rob Turknett, Tal Tversky, and Ed Zane.

I am thankful as well for the support and guidance of my committee, Kenneth De Jong, Joydeep Ghosh, Benjamin Kuipers, Raymond Mooney, and Bruce Porter.

Finally, I wish to thank all those in the NEAT community who have made NEAT more than just a single person's research project. I am very thankful for all the feedback I have received from researchers, users, and enthusiasts around the world since NEAT was first published. In particular, I wish to thank John Arrowwood, Mat Buckland, Chris Christenson, Mattias Fagerlund, Colin Green,

Darren Izzard, Derek James, Christian Mayr, Tyler Streeter, Philip Tucker, and Ugo Vierucci for sharing their own work and ideas on NEAT.

This research was supported in part by the National Science Foundation under grant IIS-0083776, the Texas Higher Education Coordinating Board under grant ARP-003658-476-2001, the DMC Lab at the IC^2 Institute, and Toyota Corporation.

KENNETH O. STANLEY

The University of Texas at Austin
August 2004

Efficient Evolution of Neural Networks through Complexification

Publication No. _____

Kenneth Owen Stanley, Ph.D.
The University of Texas at Austin, 2004

Supervisor: Risto Miikkulainen

Artificial neural networks can potentially control autonomous robots, vehicles, factories, or game players more robustly than traditional approaches. Neuroevolution, i.e. the artificial evolution of neural networks, is a method for finding the right topology and connection weights to specify the desired control behavior. The challenge for neuroevolution is that difficult tasks may require complex networks with many connections, all of which must be set to the right values. Even if a network exists that can solve the task, evolution may not be able to find it in such a high-dimensional search space. This dissertation presents the NeuroEvolution of Augmenting Topologies (NEAT) method, which makes search for complex solutions feasible. In a process called complexification, NEAT begins by searching in a space of simple networks, and gradually makes them more complex as the search progresses. By starting minimally, NEAT is more likely to find efficient and robust solutions than neuroevolution methods that begin with large fixed or randomized topologies; by elaborating on existing solutions, it can gradually construct even highly complex solutions. In this dissertation, NEAT is first shown faster than traditional approaches on a challenging reinforcement learning benchmark task. Second, by building on existing structure, it is shown to maintain an "arms race" even in open-ended coevolution. Third, NEAT is used to successfully discover complex behavior in three challenging domains: the game of Go, an automobile warning system, and a real-time interactive video game. Experimental results in these domains demonstrate that NEAT makes entirely new applications of machine learning possible.

Contents

List of Figures

Chapter 1

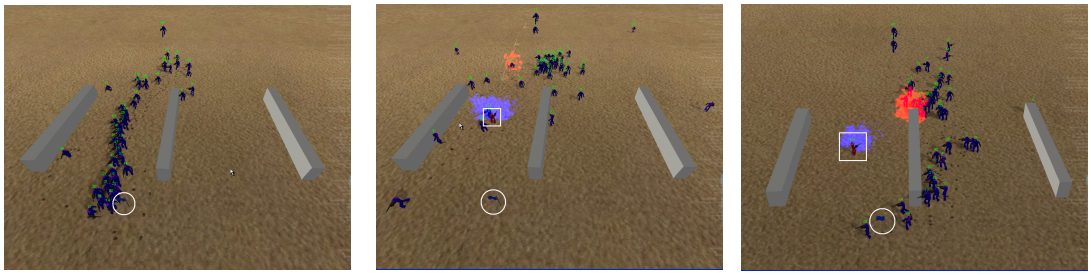
Introduction

A fundamental motivation of machine learning (ML) is to discover solutions to significant real-world problems. An important class of such problems requires discovering behavior policies for autonomous agents such as vehicles, robots, and game characters automatically. Consider for example the challenging goal of creating a video game in which characters learn *on their own* to adapt to the player's behavior (figure 1.1). As soon as human players begin to exploit characters' weakness, they could change their strategies and the game would become challenging again. Such a technology would allow the game to remain interesting far longer than today's games, and such games could even be used effectively to train people in various interactive real world tasks.

ML is necessary for such a system: Without learning, developers would need to script all possible contingencies into the system *a priori*, and have the system switch among them in reaction to players' behavior. In addition, many sophisticated strategies are difficult to program or even envision, and would only be possible to achieve through learning. Learning adds flexibility not only in games, but in many real-world scenarios, such as automated driving, military tactics, and robot control.

Sophisticated behaviors are difficult to discover in part because they are likely to be extremely complex, perhaps requiring the optimization of thousands or even millions of parameters. Searching through such high-dimensional space is intractable even for the most powerful methods. This dissertation describes a method for discovering complex neural network-controlled behaviors by gradually building up to a solution in an evolutionary process called *complexification*. The high-dimensional space of the final solution is only encountered at the very end of the search.

This chapters begins by motivating complexification, then briefly describes the approach, and concludes with an overview of the results and contributions of the dissertation.



(a) Robots approach flag (b) Player attacks on left (c) Robots learn new approach

Figure 1.1: **Video game characters adapt to player's actions (*color figure*).** The robots in these screenshots are video game characters that spawn from the top of the screen and must approach the flag (circled) at the bottom left. (a) The robots first learn to take the left hallway since it is the shortest path to the flag. (b) A human player (identified by a square) attacks inside the left hallway and decimates the robots. (c) Even though the left hallway is the shortest path to the flag, the robots learn that they can avoid the human enemy by taking the right hallway, which is protected from the human's fire by a wall. These screenshots are taken from the NERO video game (Chapter 9), in which robots learn to adapt to the player's tactics as the game is played. Machine learning is necessary for such an application to work.

1.1 Motivation

Neuroevolution (NE), the artificial evolution of neural networks using genetic algorithms, has shown great promise in complex reinforcement learning (RL) problems (Floriano and Mondada 1994; Gomez and Miikkulainen 2003; Gruau et al. 1996; Harvey 1993; Moriarty et al. 1999; Nolfi et al. 1994; Potter et al. 1995; Aharonov-Barki et al. 2001; Whitley et al. 1993). Neuroevolution searches through the space of behaviors for a network that performs well at a given task. This approach to solving complex control problems is an alternative to statistical techniques that attempt to estimate the utility of particular actions in particular states of the world (Kaelbling et al. 1996; Sutton and Barto 1998). NE is a promising approach to learning behavioral policies and finds solutions faster than leading RL methods on many benchmark tasks (Gomez 2003; Moriarty and Miikkulainen 1997).

In traditional NE approaches, a topology is chosen for the evolving networks before training begins (Montana and Davis 1989; Saravanan and Fogel 1995; Wieland 1991). Usually, the network topology is a single hidden layer of neurons, with each hidden neuron connected to every network input and output. Evolution searches the space of connection weights of this fully-connected topology by allowing high-performing networks to reproduce. The weight space is explored through the crossover of network weight vectors and through the mutation of single networks' weights. Thus, the goal of fixed-topology NE is to optimize the connection weights that determine the functionality of a network.

However, connection weights are not the only aspect of neural networks that contribute to their behavior. Their topology, or *structure*, also affects how they function. What the appropriate topology is for any particular behavior is generally not known. However, topology is an important

factor because it determines the size of the solution, and hence the size of the space in which the solution can be found. Searching in too large a space may be intractable; In fact, if the solution contains many dimensions, even searching in the space of the actual solution may be intractable. On the other hand, searching in too small a space may fail because the solution may not exist in that space.

Determining the right topology is important also because many common structures that the neural networks may need to represent and process are defined by an indefinite number of parameters. For example, the number of parts in electronic circuits and robot controllers can vary (Miller et al. 2000a; Stanley and Miikkulainen 2004). Moreover, although theoretically two neural networks with different numbers of connections and nodes can represent the same function (Cybenko 1989), they may not be equally efficient to run nor equally easy to discover. Thus, it is not clear what network topology is appropriate for solving a particular problem. Methods that search in fixed spaces must rely on heuristics to determine the appropriate topology *a priori*.

In neuroevolution, the topology is defined by the network's genetic encoding, and the size of the encoding, i.e. the number of genes, is a crucial factor determining the network topology. In highly complex domains the heuristics for determining the appropriate size are not very useful, and it becomes increasingly difficult to solve such domains with fixed-length encodings. For example, how many nodes and connections are necessary for a neural network that controls a robotic maid? The answers to such questions can hardly be based on empirical experience or analytic methods, since little is known about the solutions. One possible approach is to simply make the genetic encoding extremely large, so that the space it encodes is extremely large and a solution is likely to lie somewhere within. Yet the larger the encoding, the higher dimensional the space that evolution needs to search. Even if a robotic maid lies somewhere in the 10,000 dimensional space of a 10,000 gene encoding, searching such a space may take prohibitively long.

Even more problematic are open-ended problems where behaviors and strategies are meant to increase in sophistication indefinitely and there is no known final solution. For example, in competitive games, it is not possible to estimate the complexity of the "best" possible player in order to decide the size of a fixed-length genome; Similarly, many artificial life domains are aimed at evolving increasingly complex artificial creatures for as long as possible (Maley 1999), which is difficult with a fixed encoding for two reasons: (1) When a good strategy is found in a fixed-length encoding, the entire representational space is used to encode it. Thus, the only way to improve it is to *alter* the strategy, thereby sacrificing some of the functionality learned over previous generations. (2) Fixing the size of the encoding in such domains arbitrarily limits how complex the evolved controller can be, defeating the purpose of the experiment.

In order to discover solutions to difficult real-world problems and to open-ended problems, a method is needed that can automatically estimate the right number of dimensions for the solution. *Even if* that solution exists in high-dimensional space, search should spend the majority of time in lower-dimensional space building up a foundation for the final solution. Such a method is developed

and evaluated in this dissertation.

1.2 Approach

The NeuroEvolution of Augmenting Topologies (NEAT) method for evolving artificial neural networks is designed to take advantage of structure as a way of minimizing the dimensionality of the search space. Evolution starts with a population of small, simple genomes and systematically elaborates on them over generations by adding new genes. Each new gene expands the search space, adding a new dimension that previously did not exist. That way, evolution begins searching in a small space that is easily optimized, and adds new dimensions as necessary. This approach is more likely to discover highly complex phenotypes than an approach that begins searching directly in the intractably large space of complete solutions. In fact, natural evolution utilizes this strategy, occasionally adding new genes that make the phenotype more complex (Martin 1999; Section 2.4.4). In biology, this process of incremental elaboration is called *complexification*, which is why this term is used to describe the computational approach in this dissertation as well.

Evolving structure incrementally presents several technical challenges: (1) Is there a genetic representation that allows disparate topologies to cross over in a meaningful way? (2) How can topological innovation that needs a few generations to be optimized be protected so that it does not disappear from the population prematurely? (3) How can topologies be minimized *throughout evolution* without a contrived fitness function that measures complexity explicitly?

NEAT meets these challenges through three technical components: (1) Keeping track of which genes match up with which among differently sized genomes throughout evolution; (2) speciating the population so that solutions of differing complexity can exist independently; and (3) starting evolution with a uniform population of small networks. These components work together in complexifying solutions as part of the evolutionary process. The resulting method can evolve a diverse population of increasingly complex topologies separated into unique species. This approach results in powerful evolution that can solve benchmark problems faster than previous methods, and also makes entirely new applications possible.

1.3 Contributions and Impact

The main contribution of this dissertation is a principled method for evolving increasingly complex neural network topologies. Several experiments demonstrate the benefits of NEAT and complexification, and others suggest how the approach can be used to solve significant real-world problems.

First, NEAT is compared to both traditional reinforcement learning techniques and other neuroevolution methods in the challenging task of balancing two poles on a moving cart. The results establish that NEAT is able to take advantage of topology in order to speed up the search, resulting in highly efficient problem solving.

Second, the most significant benefit of complexification is that it allows *continual coevolution*, i.e. continual innovation in a competition. Because NEAT can complexify, it continually elaborates on its solutions, leading to increasingly sophisticated strategies. Thus, NEAT can evolve strategies and behaviors that would be difficult or impossible to discover in any other way.

Third, two new techniques are introduced that expand neuroevolution to novel domains: (1) The neural model is enhanced in two alternative ways to allow neural networks to adapt over their lifetime: First, networks adapt through synaptic plasticity and second they adapt using activation state changes. The two properties turn out to have different strengths and apply to different kinds of tasks. (2) A real-time version of NEAT is developed that allows evolution to occur while a user interacts with the system. This technique makes a new genre of video games possible and creates new opportunities for training and educational software.

NEAT is tested in two real-world applications in addition to video games, demonstrating that the technology can have a significant and versatile impact in practice. NEAT evolves Go-playing neural networks that can defeat the leading public domain Go program on a 7×7 board, and produces networks that can warn the driver before crashing a car on a simulated road, a technology that may one day save lives.

The key to all these contributions is complexification. Interestingly, the process of complexification is not limited to neural networks; complexity is ubiquitous in many important structures from biological organisms to space stations. NEAT is a first step towards an automated method for discovering complex structures across domains, and opens up exciting avenues for future research.

1.4 Overview of the Dissertation

The dissertation is divided into five main parts: Foundations (Chapter 2), the NEAT method (Chapter 3), Evaluation (Chapters 4, 5, and 6), Applications (Chapters 7, 8, and 9), and Discussion and Conclusion (Chapters 10, and 11).

Chapter 2 reviews prior work in neuroevolution, focusing on three major challenges for evolving a population of diverse network topologies: (1) How can networks in a population of diverse topologies be crossed over and compared? (2) How can innovative solutions be protected? (3) How can the size of the search space be minimized?

Chapter 3 presents the NEAT method as the solution to these challenges: Historical markings on genes ensure that topologies remain compatible, speciation is used to to protect innovation, and the search space is minimized by starting with small networks and incrementally adding complexity.

Chapter 4 focuses on performance evaluation. NEAT is first tested on the XOR problem to determine whether it can evolve topology when necessary, and then compared to traditional reinforcement learning and other neuroevolution methods on the challenging task of balancing two poles on a moving cart. These comparisons establish that NEAT is efficient at solving well-known

problems. Finally, a series of system ablations and a new species visualization technique confirm that all components of NEAT contribute to its performance, i.e. that NEAT is a principled and cohesive methodology.

In **Chapter 5**, a coevolutionary arms race is demonstrated by evolving robot controllers to compete in a duel. Complexification allows strategies to be continually elaborated, leading to novel solutions that would be difficult to discover or even foresee.

Chapter 6 compares two methods for evolving networks that adapt. First, the neural model in NEAT is expanded to allow networks to adapt by changing connection weights over their lifetime. Hebbian rules are evolved that determine how the weights should adapt in response to stimulus. Second, recurrent networks with static weights are trained to alter their behavior based on their internal state. The plastic and static networks are compared in a food foraging/avoidance task; both are found to solve the problem but have different strategies. The ability to adapt is important because ultimately the goal of neuroevolution is to evolve neural networks that work like brains in biological organisms, which frequently must adapt to new situations.

In **Chapters 7, 8, and 9**, NEAT is tested in three major applications: the game of Go, automobile driver warning, and a real-time video game. These applications illustrate that NEAT can evolve effective behaviors in diverse domains, and that it makes new kinds of applications possible. The neural networks evolved for Go control a roving eye that scans the board and makes a move even though it cannot see the whole board at once. Automobile warning networks predict when a car will crash based on the driver's recent behavior. The real-time version of NEAT makes an entirely new genre of video game possible.

Chapter 10 discusses and reviews the major contributions of NEAT and complexification, including protecting innovation, and suggests three avenues for future expansion and research: evolving non-neural structures, expanding the neural model, and evolving structures that develop from a single cell like embryos in nature. **Chapter 11** reviews the major contributions of the dissertation and their significance to artificial intelligence and machine learning.

In order to make it possible to replicate the experiments and to apply NEAT to other domains, **Appendix A** gives NEAT system parameters used in experiments throughout the dissertation and **Appendix A.4.1** gives equations of motion for pole balancing. Finally, NEAT source code, demos, and a tutorial are available through `http://nn.cs.utexas.edu/keyword?neat`.

Chapter 2

Foundations

This chapter will review issues in evolutionary computation, neural networks, and biology necessary for understanding NEAT. After briefly reviewing encoding issues in genetic algorithms and the architecture of feedforward and recurrent neural networks, the focus will shift into neuroevolution, specifically the simultaneous evolution of topology and weights. These systems will be referred to as TWEANNs (Topology and Weight Evolutionary Artificial Neural Networks). After discussing TWEANNs and some of the key issues in the field, the next section will examine problems with these systems and pose principles of TWEANN design that address the problems. Finally, the chapter concludes with a review of issues in competitive coevolution that NEAT attempts to address.

2.1 Genetic Algorithms and Genetic Encoding

Genetic algorithms (De Jong 1975; Goldberg 1989; Holland 1975; Mitchell 1996) are a class of computational search algorithms inspired by evolution in nature. The object of GAs is to search through a parameter space for a set of parameters that optimize some performance criterion. GAs are based on natural selection, where the more fit individuals are selected to procreate. In a GA, a population of solutions is maintained where each solution represents a point in the search space. Usually the search space is conceived of as a finite-dimensional parameter space. These parameters can be binary, discrete, real, or any other searchable encoding scheme. A string of parameters is called a *genotype*. A genotype is transformed into a *phenotype* through a genesis procedure. The phenotype is then evaluated on some fitness criteria on a task. The phenotypes with higher fitness are allowed to mate their genomes to create offspring in the hopes that the combination of two good sets of parameters will produce an even better parameterization. Mutations, which occur during reproduction, cause random perturbations of parameters. Mutations ensure that the search covers new areas of the search space.

It is important to note that the general theme of GAs can vary significantly. The way the genotype is encoded can significantly impact the number of evaluations it takes to find a solution and

can even make finding certain solutions impossible. One area where NEAT diverges from standard GAs is that NEAT is able to change the space in which it is searching by adding or subtracting dimensions from that space. In other words, NEAT is able to change the number of parameters it is trying to optimize. Thus, principles of standard GAs that work in finite spaces are not necessarily applicable to NEAT's form of search, and NEAT can find both smaller and larger solutions than algorithms that search a space of a fixed number of dimensions.

An algorithm called *Messy GA* (Goldberg et al. 1989; Whitley et al. 1997) also allows variable length chromosomes as NEAT does. However, Messy GAs do not grow structure. Rather, they allow underspecification or overspecification of a phenotype. They are intended for *subset feature selection problems* where some subset of a global set of features represents a solution. Therefore NEAT is not a Messy GA even though both methods use variable-length chromosomes.

GAs are particularly useful on problems with little domain knowledge because they require no a priori analysis of the hypothesis space or of the problem domain. For example, they can search for neural networks in problems with sparse reinforcement even when the contribution of any output towards the goal is unknown. GAs are also useful for problems with local minima in the error surface that are likely to trap gradient descent methods. Because GAs sample multiple points on the error surface, they can search different parts of parameter space at the same time.

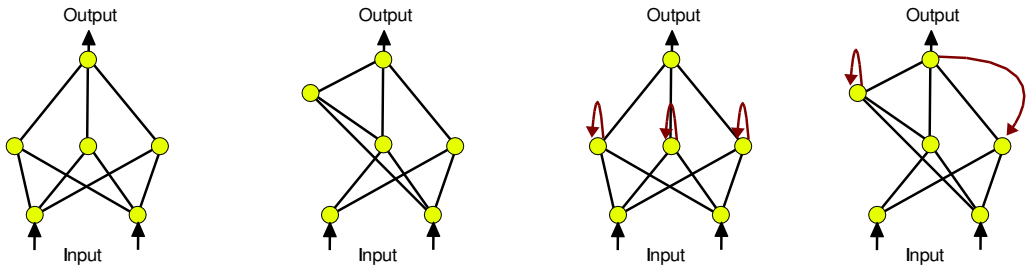
In NEAT, connection weights are coded in the genome as real numbers. Holland (1975) showed with the Schema Theorem that genetic algorithms tend to converge to better and better solutions. The Schema theorem was based on binary encoding, but Wright (1991) extended it to cover real-number encoding as well. Wright showed that real number encodings are better for some continuous optimization problems. Herrera and Lozano (1998) presented an extensive review of genetic operators over real-coded chromosomes, including different types of crossover. Thus, real numbers are an appropriate and well-understood form of representation for GAs in continuous domains.

In summary, genetic algorithms are a useful method for the type of sparse reinforcement problems for which NEAT is intended. The use of real number encoding of genes in a genome is supported by past research and is natural in a GA that optimizes connection weights since they are represented as real numbers.

2.2 Artificial Neural Networks

This section introduces some basic concepts in neural networks, the phenotype being optimized by neuroevolution methods.

Artificial Neural Networks are computational processing structures motivated by the structure and function of biological nervous systems in natural living beings (Haykin 1994). Neural networks are able to approximate any continuous function in theory (Cybenko 1989), making them very powerful tools for control and prediction.



(a) Feedforward (b) Feedforward non-standard (c) Recurrent (d) Recurrent non-standard

Figure 2.1: **Neural network architectures.** (a) A fully-connected single hidden layer feedforward network. (b) A partially connected feedforward network. Notice there is no clear layering scheme. (c) A recurrent network. Each hidden unit feeds activation back into itself. (d) An unusual recurrent network. This network is not fully-connected and has two recurrent connections, one going from the output unit back to a hidden node. Red lines represent recurrent connections, black are feedforward. This figure shows that many different types and topologies of networks are possible, each with its own functionality.

Neurons, also called *nodes* are the fundamental processing unit in neural networks. Many neurons are connected together to form a full network. A network can be conceived of in terms of layers. There is an input layer of "sensors" that receive input from the external world. These sensor neurons pass their activation forward over weighted connections to other neurons. The last layer in a neural network is the output layer, the output of which is used by the system containing the neural network. In between the input and output layers are hidden neurons, which are in "hidden layers." However, the inner neurons do not necessarily have to be in strict layers. Figure 2.1a shows a feedforward neural network. Although neural networks are traditionally thought of as strictly layered and fully-connected, NEAT does not follow this tradition because it evolves its own networks topologies. Evolved topologies likely are not fully connected and do not have clear layers, such as the network in figure 2.1b.

Each neuron computes a weighted sum of its inputs. The sum passes through an *activation function*, σ, which squashes the activation into a range between 0 and 1. Thus for every neuron j, its output y_j for a given input vector $\mathbf{x}$ is given by:

$$y_j = \sigma\left(\sum_i w_{ij} x_i\right), \tag{2.1}$$

where w_{ij} is the connection weight from node i to node j. The function σ is usually the nonlinear sigmoid function $\frac{1}{1+e^{-x}}$, but can be modified to suit the needs of a system.

A network where activation starts at the inputs and flows forward to the outputs is called a *feedforward* network. Networks can also have feedback connections that go backward instead of forward. Networks with feedback connections are called *recurrent* (figure 2.1c). Lin et al. (1996) and Ring (1994) describe how recurrent networks can be useful for learning temporal dependencies.

NEAT can evolve unusual recurrent structures (figure 2.1d) because it constructs network topology on its own. It is useful to be able to evolve specific recurrent topologies because it is not always necessary to have a recurrent connection for every hidden unit. Many times, the temporal dependencies in a problem can be captured with only one or two recurrent connections. Thus, it is a waste of parameter space to employ unnecessary recurrent connections.

Neural networks can be trained using gradient descent methods such as backpropagation (Rumelhart et al. 1986). However, such methods can be trapped at local minima. In addition, they are not well suited to sparse reinforcement domains because (1) feedback is not available for every output iteration, and (2) backpropagation requires output targets, which are not available in reinforcement learning tasks. Training recurrent networks is slow and less reliable than training feedforward networks (Bengio et al. 1994).

Artificial neural networks and the networks of neurons in living brains differ in many ways. One of them is that the weights of the connections between neurons do not change during the lifetime performance of a static network. This point is important because ultimately it is desirable to evolve networks with *synaptic plasticity*, where the rules and loci of changeable connection weights are evolved (Floreano and Urzelai 2000). These kinds of *adaptive* neural networks are simply higher dimensional parameterizations than evolution only involving topology and weights. An example of a weight update rule is the *Hebb Rule*:

$$\Delta w = \eta(1 - w)xy\,, \tag{2.2}$$

where x is the activity of the incoming neuron, y is the activity of the outgoing neuron, and η is the learning rate. This rule strengthens connections between neurons that tend to fire together. One can imagine a complex topology of interconnected Hebbian synapses of varying learning rates, which is as a whole able to adapt to a changing environment. Other rules can also be attached to specific synapses, while other synapses can be simple static feedforward weights. Chapter 6 describes an elaboration of NEAT that allows it to evolve Hebbian connections and networks that can adapt to change.

The next section describes a different method for training neural networks called *neuroevolution*, which avoids local minima better than backpropagation and can be applied to sparse reinforcement domains.

2.3 Neuroevolution

Neuroevolution (NE) is a combination of neural networks and genetic algorithms where neural networks are the phenotype being evaluated. The genotype is a compact representation that can be translated into an artificial neural network. NE searches for neural networks that optimize some performance measure. NE can search for virtually any kind of neural network whether it be simple feedforward, recurrent, or even adaptive networks.

The chromosomes in NE can represent any parameter of neural networks, from the connection weights to the topology to the activation functions. Since the choice of encoding affects the search space of solutions, it is a pivotal aspect of the design of an NE system.

While some NE methods evolve only the connection weights of the network, *Topology and Weight Evolving Neural Networks (TWEANNs)* evolve both weights and network topologies (Yao 1999). Fixed-topology methods require a human to decide the right topology for a problem. In contrast, TWEANNs can discover the right topology on their own. In addition, topology evolution can be used to increase efficiency by keeping networks as small as possible, which is a strategy employed by NEAT. Therefore, TWEANNs are an important class of NE methods and they face several specific challenges and difficulties that fixed-topology NE methods do not.

We begin by looking at three fixed-topology NE systems and then turn to a discussion of TWEANNs.

2.3.1 Fixed-Topology NE Systems

NEAT will be evaluated on some problems that other NE systems have tackled (Chapter 4). Most of these systems evolve fixed topologies, so they optimize connection weights only. The three highest-performing NE systems to date are presented here because they are NEAT's closest competition. They have performed better than any other systems on benchmark tasks.

Symbiotic, Adaptive Neuroevolution (Moriarty 1997; Moriarty and Miikkulainen 1996) is a neuroevolution system that evolves populations of neurons instead of populations of networks. The neurons are combined to form the fully connected hidden layer of networks where they are evaluated. The neurons receive the average fitness of the networks in which they were included. SANE maintains diversity (in the neuron population) because a dominant neural phenotype is likely to end up in the same network more than once. Because several different types of neurons are usually necessary to solve a problem, networks with too many copies of the same neuron are likely to fail. The dominant phenotype then loses fitness and becomes less dominant.

Enforced Subpopulations (Gomez and Miikkulainen 1997, 1998, 1999; Gomez 2003) improves on SANE by forcing neurons to specialize on specific subtasks. Each unit in the network is assigned a separate subpopulation. Recombination occurs between neurons in the same subpopulation only. Unlike in SANE, the species in ESP do not need to organize themselves since they are enforced from the start. Also, neurons only play one role in ESP, whereas in SANE they may be evaluated in different roles depending on the context of other neurons they happen to be joined with. As a result, ESP allows recurrent networks to be evolved. ESP may work well because it makes sure neurons get the credit they deserve, unlike other neuroevolution techniques where bad neurons can share in the fitness of a good network, or good neurons can be brought down by their poorly configured neighbors. It also works by decomposing the task, breaking the search into smaller, more manageable parts.

Recently, Igel (2003) successfully applied a special Evolutionary Strategy (ES) called CMA-

ES to the evolution of fixed-topology neural networks. Igel's method keeps track of correlations between changes of different weights in the network and fitness. Based on this information, the CMA-ES changes the covariance matrix of the weight mutation distribution so that it becomes more biased towards what were so far the most promising directions of search. This method has proven effective on benchmark tasks (see Chapter 4).

All three systems have shown promise in reinforcement learning problems, and will be compared with NEAT in Chapter 4. While SANE and ESP are designed to quickly evolve neurons that cooperate to form a network, NEAT concentrates on taking advantage of structure to gain speed. Thus, NEAT represents a different philosophy. Tracking correlations as in the CMA-ES is also different than NEAT, although it may be possible to combine the two methods in the future.

2.3.2 TWEANNs

The purpose of this section is to give some background on TWEANN encoding and then review several prototypical methods that have been developed to evolve ANN topology and weights simultaneously.

Before introducing TWEANN methods, it is important to note how direct encoding and indirect encoding in TWEANNs differ. Genomes in TWEANNs encode both the topology and connection weight values of a network. TWEANN designers must decide whether arbitrary topologies should be encoded directly or indirectly. Direct encoding schemes specify in the genome every connection and node that will appear in the phenotype (Pujol and Poli 1997; Zhang and Muhlenbein 1993; Braun and Weisbrod 1993; Opitz and Shavlik 1997; Yao and Liu 1996; Angeline et al. 1993; Krishnan and Ciesielski 1994; Maniezzo 1994; Lee and Kim 1996; Dasgupta and McGregor 1992). In contrast, indirect encodings usually only specify rules for constructing a phenotype (Bongard and Pfeifer 2001; Gruau et al. 1996; Hornby and Pollack 2002; Mandischer 1993). These rules can be layer specifications or growth rules through cell division. The main idea of indirect encoding is that every connection and node are not specified in the genome, although they can be derived from it. TWEANNs that use indirect encoding include *artificial embryogeny* (AE) methods that evolve phenotypes that develop from a small embryonic starting structure. Setion 10.5 includes an in-depth discussion of AE and its potential combination with NEAT.

The first three methods described below use direct encoding. The last method uses indirect encoding. Each method is described based on an actual system that implements the method.

Binary Encoding

The simplest TWEANN representation is binary encoding. In one such implementation, Dasgupta and McGregor (1992) use such an encoding in their method, called sGA (Structured Genetic Algorithm). sGA is notably simple, allowing it to operate almost like a standard GA. A bit string represents the connection matrix of a network. A "1" in a location in the matrix represents a connection from the node with the same number as the row of the matrix to the node with the same

number as the column. Feedforward connections are therefore all represented in the upper-right triangle of the matrix. Recurrent connections are in the lower left. However, current implementations of sGA do not use recurrent connections, so the lower left triangle is always all zeros. A "low-level" bit string is evolved along with the "high-level" bits representing the connectivity matrix. The low-level bits are used to represent the actual weights of connections. Because the genotype is just two bit strings, those strings can be mated using the standard binary crossover operator from GAs.

Although binary encoding such as in sGA is simple, it has several limitations. First, the size of the connectivity matrix grows as the square of the number of nodes. Thus, the representation blows up for a large number of nodes. Since the solution networks probably are not fully-connected, a good proportion of the genome is wasted. Because the bit string must have the same size for all organisms, the maximum number of nodes (and hence connections as well) must be chosen by the human running the system. If it turns out that more nodes are needed, the entire experiment must be restarted with a larger matrix.

The genomes in the initial population of sGA are random bit strings. Each genome specifies a random topology. Thus, a significant percentage of the initial population is *infeasible*. Infeasibility means that a network phenotype has no paths from all the inputs to the outputs. In some cases, there are no paths to the outputs from *any* inputs. Because of this problem, a significant amount of effort is wasted in ridding the population of infeasible networks. It is such a serious problem that the fitness function has to include a measure of infeasibility.

Using a linear string of bits to represent a graph structure makes it difficult to ensure that crossover of bit strings will yield useful combinations. Thus, a significant chunk of offspring are likely to be defective, or might even introduce new infeasible networks into the population. Part of the problem is that crossing over bit strings does not consider what those strings represent. This problem is the reason most TWEANNs use more sophisticated encoding schemes.

Graph Encoding

Most TWEANNs use encodings that represent the graph structure of networks more naturally than bit strings. A graph structure as an encoding allows meaningful crossover and mutation since graphs can be analyzed and subgraphs can be assembled into new genomes. Pujol and Poli (1997) use a dual representation scheme to allow different kinds of crossover in their Parallel Distributed Genetic Programming (PDGP) system. The first representation is a graph structure. The second is a linear genome of node definitions specifying incoming and outgoing connections. The idea is that different representations are appropriate for different kinds of operators.

As in sGA, the number of nodes in the PDGP network is limited to the number of nodes in the two-dimensional grid that represents the graph version of the genome. Subgraph-swapping crossovers and topological mutations use the grid, while point crossovers and connection parameter mutations use the linear representation. Just as sGA, PDGP starts with an initial population of randomly-connected networks.

PDGP uses graph encoding so that subgraphs can be swapped in crossover. Subgraph swapping is representative of a prevailing philosophy in TWEANNs that subgraphs are functional units and therefore swapping them makes sense because it preserves the structure of functional components. Subtree swapping in PDGP reflects a similar operation in Genetic Programming in which subtrees of program trees are swapped during crossover (Koza 1992). PDGP succeeds in keeping together functional subunits by moving them around as subgraphs during crossover. Although swapping subnetworks does help keep subgraphs together, many offspring do not represent good combinations of their parents' subgraphs because there is no way to know which subgraphs perform which functions, and therefore it is not clear how they should be recombined.

Crossover is a significant problem for TWEANNs because there are many way to represent the same solution, and crossing over these differing conventions can damage functionality. The general problem of crossing over different structures will be covered in more detail in Section 2.4.2.

PDGP shows that graph encoding can be useful in preserving substructures in crossover. The system is intuitively appealing because when we think about combining neural networks, we think about combining graphs. However, we cannot be sure whether the subgraphs being combined in PDGP are the right building blocks to create a functional offspring.

Non-mating

Because crossover of networks with different topologies frequently diminishes functionality, some researchers give up on crossover altogether in what is called Evolutionary Programming. Angeline et al. (1993) implemented a system called GNARL (GeNeralized Acquisition of Recurrent Links), commenting that "the prospect of evolving connectionist networks with crossover appears limited in general." GNARL, like mating TWEANNs, initializes a population with randomly connected networks and a finite maximum number of hidden nodes. The networks are represented as graphs, so that structural mutations can be assured to be coherent. Structural mutations (on topology) and parametric mutations (on weights) are the only genetic operators in the system. GNARL adds nodes to genomes without any connections and it may only add up to the maximum allowed number of nodes.

Experimental results with NEAT suggest that the prospects for evolving neural networks with crossover are not limited after all (Section 4.3). However, results from prior TWEANN systems have not justified using crossover. GNARL is a logical attempt to simplify the problem of evolving topology and weights by simply removing crossover as a factor.

There are a number of ways to add nodes to a network in TWEANNs. A new node can be added to the genetic encoding without any connections, or it can be immediately connected into the genome. GNARL, like most TWEANNs, takes the former approach. The authors of GNARL decided to add nodes in isolation because adding nodes with connections can alter a network's fitness by introducing a nonlinearity where there was not one before. By starting out with no connections to new nodes, the nodes are able to survive inside the genome without disrupting functionality and

causing fitness to decrease. In subsequent generations, it is hoped, new connections will integrate new nodes into the network and make them useful. One problem that results is that genomes end up representing many extraneous disconnected structures that do not have any contribution to the solution.

The main contribution of GNARL is to show that TWEANNs do not require crossover to work, and cast some doubt on whether crossover is necessary in TWEANNs. However, ablation studies with NEAT in Section 4.3 show that crossover indeed enhances the performance of NEAT.

Indirect Encoding

Gruau's Cellular Encoding (CE) method (Gruau 1993, 1994; Gruau et al. 1996) is an AE system that uses indirect encoding. In CE, genomes are programs written in a specialized graph transformation language called a *grammar tree*. The transformations are motivated by nature in that they specify *cell divisions*. Different kinds of connectivities can result from a division, so there are several kinds of cell divisions possible. CE has one major advantage: Its genetic representations are compact. Genes in CE can be reused multiple times during the development of a network, each time requesting a cell division at a different location. Reusing genes saves space in the genome because every connection and node in a network does not need to be explicitly specified in the genome. CE is a landmark system, showing that developmental rules can encode the development of networks from a single cell, much as organisms in nature begin as a single cell that differentiates as it splits into more cells, which in turn split into even more cells.

Figure 2.2 depicts how the grammar tree is translated into a developing network. The grammar tree contains developmental instructions at each node that split one cell into two cells, change the values of links between cells, and remove existing links between cells. Developing cells in CE read from different parts of the grammar tree at the same time. A *reading head* for each cell indicates from which part of the grammar tree it is reading. When a cell encounters an *end* instruction, its state is finalized and it stops reading. CE uses a First In First Out (FIFO) queue of cells in order to keep track of which cells are currently executing instructions, and in what order they should be executed. A cell at the front of the queue executes the instruction to which its reading head points, moves its head to the subsequent instruction in the grammar tree, and then goes to the end of the queue. Sometimes cells encounter instructions to divide (there are a variety of cell division methods such as parallel and sequential splitting), in which case the original cell moves its reading head down the left subtree, and the new cell moves its head down the right subtree. Thus, cell divisions allow different cell lineages to follow different developmental pathways.

Compact representations in CE are desirable because they save space, but they do not solve other problems. Crossover is still a problem for CE just as other systems, except that it is harder to analyze how crossover disrupts subfunctions in CE encoding since they are not represented explicitly. In addition, it is not clear how a mutation resulting in an extra cell division enhances the search.

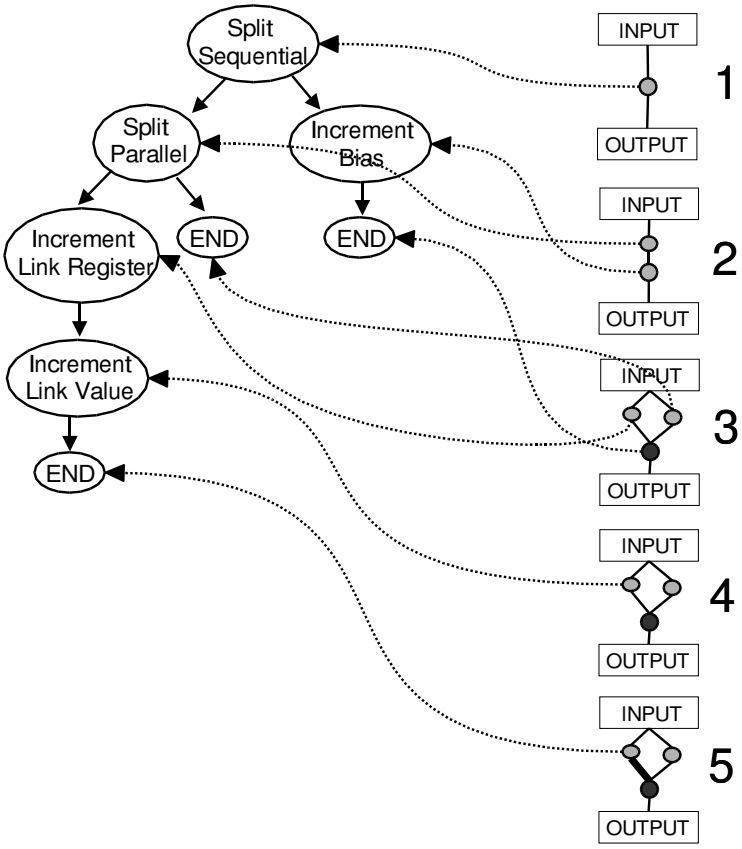


Figure 2.2: **Cellular Encoding (CE) example .** The grammar tree is shown at left and the network growth is shown from top to bottom in five steps at right. The network begins at step 1 as a single cell. At each step, each network cell is reading from its own part of the tree. Dotted lines identify the location in the tree from which each cell is reading at each step. When a cell splits, its descendent cells take separate paths in the tree. The "Increment Link Register" instruction is the way a cell knows to which link it should apply any subsequent link-based instructions. In the example, such an instruction occurs immediately following the link register instruction, causing a link's value to increase, represented in the network by a thickening line. Darker cells have a higher activation bias. This example is based on others given by Gruau (1993). While Gruau uses abbreviated instructions, this figure spells them out entirely to make the example easy to follow. Gruau (1993) proved that Cellular Encoding grammar trees can describe any network topology.

Experimental results confirm CE's weakness in practice (Section 4.2.3). CE is interesting because it represents an early attempt to actually implement development from genotype to phenotype. I believe that developmental TWEANNs will be an important field in the future when extremely large networks are impractical to represent directly (Section 10.5). In order to implement an indirect TWEANN system that is anything more than ad hoc, we need to understand the principles that underly how phenotypic substructures emerge. Without understanding these principles, it

is difficult to implement growth rules that will lead to *useful* structures. In CE, representing structure through cell divisions does not necessarily improve solution performance. In the future, analysis of the kinds of structures that develop in systems like NEAT should lead a to new understanding that can in turn lead to more sophisticated developmental systems in the tradition of CE.

The remaining background sections address three general problems with TWEANNS and some possible solutions: (1) the problem of mating different topologies, (2) the problem of protecting innovation, and (3) the problem of searching in an unnecessarily large space. Finally, the chapter concludes with an overview of competitive coevolution and complexification.

2.4 Encoding Structure

This section examines problems that arise when genomes encode structure in addition to weights. In particular, because there are many ways to encode the same functionality, it is difficult to compare or cross over different solutions.

2.4.1 Direct vs. Indirect Encoding

Some TWEANNs encode structure with bit strings. Others use graphs. Some encodings are indirect. What is the best way to encode structure in a genome? The methods currently used have several limitations that NEAT addresses.

The first question is whether to use direct or indirect encoding. NEAT currently uses direct encoding because, as Braun and Weisbrod (1993) argue, indirect encoding requires "more detailed knowledge of genetic and neural mechanisms." In other words, because indirect encodings do not directly map to their phenotypes, they implicitly restrict the search to the class of topologies to which they can be expanded. We need to be sure that indirect encodings do not restrict phenotype networks to some suboptimal class of topologies. However, it is difficult to ensure that indirect encodings are sufficiently expressive to cover the class of useful topologies, because there is not yet any theory specifying what kinds of topologies are useful. Therefore, indirect encodings currently may arbitrarily bias the search towards kinds of topologies that may or may not be sufficiently expressive. In the future, after more is known about the ways different developmental properties affect how structures evolves, it will be possible to design more well-founded indirect genetic codes. At that point, it may be possible to combine NEAT with an indirect encoding scheme. For now, however, the most straightforward implementation of NEAT is with a direct encoding.

One limitation in most TWEANNs, whether direct or indirect, is that the genome can only grow up to some bound. Thus, there is a cap on the number of possible hidden nodes in a network. It may seem that this limitation is somehow natural or necessary. However, it is a shortfall for all these methods, since they are supposed to relieve humans of the responsibility of figuring out how many hidden units are necessary. To ask a user to decide a priori the maximum number of nodes needed

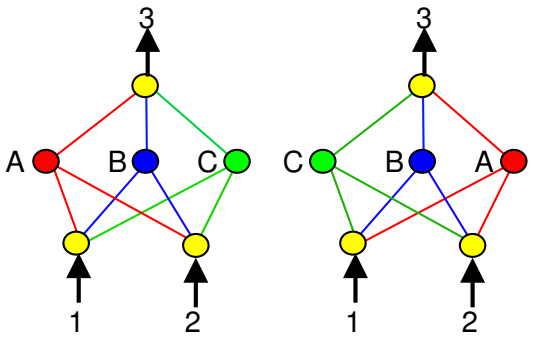


Figure 2.3: **Competing conventions problem (*color figure*).** The two networks compute the same exact function even though their hidden units appear in a different order and are represented by different chromosomes. These are 2 of the 6 possible permutations of hidden units. The neurons with their respective connections are color coded to make it easier to see how they correspond. Because the same neurons occupy different positions in the two networks, it is difficult to combine them without losing functionality.

does not relieve the user from the problem of preconceiving the complexity of the task. NEAT puts no bound on the number of nodes or connections a genome can represent.

The way structure is encoded also affects what kind of offspring can result from crossover. TWEANNs usually do not address the fact that there is no way to know which substructures of two different parents fit together in useful ways. The next section begins by highlighting this problem and then suggest a solution motivated by biology.

2.4.2 Competing Conventions

One of the main problems with NE is *Competing Conventions*, also known as the *Permutations Problem* (Radcliffe 1993).[1] This problem is magnified in TWEANNs because of their variable structure. Thus, in TWEANNs, it is called the *Variable Length Genome Problem*. This section describes competing conventions and the variable length genome problem in detail. They are used as an introduction to the general analytic challenges that varied topologies present.

Competing conventions refers to having more than one way to express a solution to a weight optimization problem with a neural network. In fact, there are potentially numerous symmetric solutions. When genomes representing the same solution do not have the same encoding, crossover is likely to produce damaged offspring because the encodings are not compatible.

Consider a simple single-layer feedforward network with three hidden neurons A, B, and C (figure 2.3). If the network is fully-connected, then each hidden unit has two incoming connections and one outgoing connection. All of the connections in the network form a vector

$$w_{A,1}w_{A,2}w_{A,3}w_{B,1}w_{B,2}w_{B,3}w_{C,1}w_{C,2}w_{C,3},$$

[1] A good discussion of competing conventions can be found in Whitley (1995).

where $w_{n,i}$ is the weight connecting hidden node n to input or output i, where $i = 1, 2$ are inputs and $i = 3$ is the output. The hidden nodes do not have to be ordered $[A, B, C]$. They could be ordered in any permutation and the network would still compute the same function. For example, if the nodes were ordered $[C, B, A]$, we would have the vector

$$w_{C,1}w_{C,2}w_{C,3}w_{B,1}w_{B,2}w_{B,3}w_{A,1}w_{A,2}w_{A,3},$$

which represents the same exact solution as the previous vector. Thus, there are 3!=6 vectors representing equivalent solutions, or competing conventions. The number of competing conventions gets higher with more hidden units, since there are in general $n!$ permutations of n hidden units.

The reason this explosion of competing conventions is such a serious problem is that permuted representations of similar solutions are easy to damage with crossover. For example, crossing $[A, B, C]$ and $[C, B, A]$ can result in $[C, B, C]$, a representation that has lost one third of the information that both of the parents had. The problem is that many times solutions are not genetically compatible even if they are functionally equivalent.

Lest there be doubt that this problem is serious, consider the simple example of the genome $[A, B, C]$ representing a 3 hidden unit architecture. There are $3! = 6$ permutations of the three hidden nodes, each representing the same solution. Let us enumerate every possible crossover among these 6 permutations. Since each permutation can crossover with any other permutation (including itself), there are $6 \times 6 = 36$ possible mating pairs of permuted structures representing competing conventions. For a 3 node representation, there are 4 possible crossing points using customary singlepoint crossover: $[1A2B3C4]$. Since there are 36 possible mate pairings, and each can cross over in 4 different points, there are $36 \times 4 = 144$ total possible crossover products (offspring). If we enumerate every one of these 144 potential products, 48 of them display severe loss of genetic information and redundancy. In other words, 48 out of 144 possible offspring are missing one of the hidden nodes A, B, or C. However, 72 of the 144 products are trivial (meaning that the offspring is a duplicate of one of the two parents because the crosspoint was on one end of the genome.) If we only look at nontrivial crossovers, where an offspring is not simply a duplicate of a parent, then the odds of an offspring with severe genetic damage is $\frac{48}{72} = 66.6\%$! With the odds of producing an undamaged offspring less than chance, the risk from competing conventions needs to be minimized.

Hancock (1992) suggested that the problem might be alleviated because the permuted solutions means that there are multiple optima in the search space. Nevertheless, researchers continue to consider the problem of competing conventions serious. Radcliffe (1993) suggested using multiset theory to find a non-redundant representation of similar solutions. Multiset theory allows different conventions to be grouped together as long as they represent the same solution. Thierens (1996) introduced a way to order neurons in order to avoid different permutations. However, these solutions generally involve simplifying assumptions about the topology and layers of the networks, and are also computationally expensive.

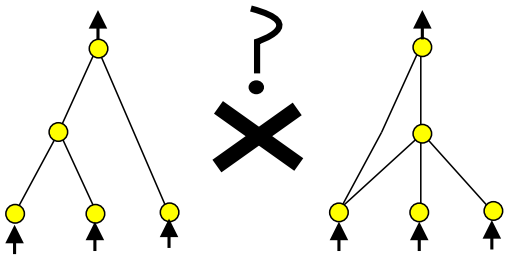


Figure 2.4: **Mating different topologies.** When two parents have different topologies, it is not obvious how their offspring should be formed. When determining which nodes and connections the offspring should inherit, it would be helpful to know which subnetworks from the parents perform the same functions, and which represent disjoint concepts. Unfortunately, this information is not readily apparent from the two different topologies. This problem is an example of the variable-length genome problem, which is a challenge for TWEANNs that aim to gain an advantage from combining different solutions.

2.4.3 The Variable Length Genome Problem

The proposed solutions to the competing conventions problem are not applicable to TWEANNs because TWEANNs do not assume anything about what topologies that can be expected. Radcliffe (1993) goes as far as calling an integrated scheme combining connectivity and weights the "Holy Grail in this area." Unfortunately, the problem of competing conventions is exacerbated by the variety of topologies of potential mates that such an open philosophy fosters.

At first glance, there appears to be a trade-off between representing structure freely and maintaining compatibility. The more open the representation, the less likely different genotypes will be compatible in useful ways (Figure 2.4). The more restrictive the representation, the fewer phenotypes are possible.

Past attempts at evolving varying topologies confronted the apparent trade-off between flexibility of representations and compatibility of genotypes in one of two ways. The first approach is to implement complex crossover algorithms that use topologically motivated analysis and/or constraints to minimize the damage that occurs when crossing over genomes representing two disparate structures (Braun and Weisbrod 1993; Dasgupta and McGregor 1992; Gruau et al. 1996; Krishnan and Ciesielski 1994; Mandischer 1993; Maniezzo 1994; Opitz and Shavlik 1997; Pujol and Poli 1997; Zhang and Muhlenbein 1993). The problem with this idea is that sometimes there is simply no combination of genetic parts that preserves functionality between different structures. Crossover generally involves combining subgraphs of network topologies, with the idea that a subgraph represents a functional unit. However, the competing conventions problem shows that this assumption is not likely to be true, since a functional unit in one part of a topology in one parent may not be located in the same place in another parent, so the *wrong* substructures are likely to be exchanged. In addition, totally different topologies are often not based on that same substructures at all, so that

no meaningful combination of substructures exists. It is as if one were to suggest that a human could mate with a sloth, if only the correct analysis of their structures were possible, and yield something superior to both human and sloth! Clearly, the power of recombination has its limits.

Even if some kind of analysis could combine seemingly incompatible structures, the kind of analysis required to extract meaningful functional units and combine them in useful ways is computationally complex. Graph matching algorithms do exist (Gold and Rangarajan 1996) but such algorithms do not help with TWEANN crossover because they do not indicate which subgraphs match up with which subgraphs when graphs differ. Theoretically, it would be better if topological analysis could be avoided altogether, due to the complexity of graph analysis. NEAT aims to achieve this goal.

The second approach to the problem of variable-length genomes is to do away with crossover entirely, leaving all progress to mutation (Angeline et al. 1993; Lee and Kim 1996; Yao and Liu 1996). While this idea may seem radical, it is reasonable if mating does more harm than good. Nevertheless, theoretically, it would be better if mating did more good than harm, so that it could enhance the performance of neuroevolution as predicted by the Schema Theorem.

The competing conventions problem and variable-length genome problem are mating-specific problems, but they set the stage for a broader discussion of issues involving diverse topologies. The relationship between different structures is not only important in crossover. It is also important in measuring how compatible different genomes are and ensuring that certain topologies are not combined. Thus, TWEANNs would benefit greatly from a simple and efficient approach to combining and comparing different topologies.

It turns out that there does not have to be a trade-off between freely representing topologies and keeping them compatible. NEAT tackles variable-length genomes in a novel way. The system tracks genes through evolution in order to tell which genes match up. Thus, permutation of vectors is not an issue, because each feature in a vector is labeled with its historical origin. For those structures that are completely incompatible, NEAT has a compatibility operator that can be used to prevent them from ever mating. The compatibility operator is also based on the gene tracking labels, which allow NEAT to tell how much history two genomes have in common. With these tools, it is possible to both prevent incompatible genotypes from mating and ensure that compatible genotypes mate in a way which maintains their functional subunits without damage. Thus competing conventions may still exist in the population, but they do not cause NEAT to explore useless recombinations. The next section motivates NEAT's approach by analyzing the mechanisms behind biological crossover.

2.4.4 Biologically Motivation: Artificial Synapsis

The main intuition behind NEAT comes from the fundamental problem with representing different structures: Representations will not necessarily match up. Sometimes, a genome can be longer than one with which it is crossing over. Sometimes, genes in the same exact position on different chromosomes may be expressing completely different traits. Sometimes, genes expressing the same

trait may appear at different positions on different chromosomes. How can these complications be resolved?

The answer is that genes that express the same feature need to be matched up, or *aligned* before crossover occurs. Of course one would not want to align a gene expressing a hair protein with another gene expressing eye pigment. How can such misalignments be avoided? In NEAT, the answer is that genes can retain a marker of their historical origin. No matter how far in the future the progeny have diverged from the originating parents, perhaps even to the point of being different species, NEAT can still tell without any ambiguity which genes come from the same historical origin and therefore are likely to express the same trait. In other words, NEAT has a built in method for testing gene *homology*:

Principle of Homology Marking genes with a number representing their order of appearance, i.e a *historical marking*, makes it possible to identify homology between genes.

Nature faces an analogous problem with gene alignment and uses an analogous method in crossover. Genomes in nature are not of fixed-length either. Consider a human and a single-celled organism. Somewhere along the line, new genes were added to life's original genomes (Darnell and Doolittle 1986). This process is called *gene amplification* (Watson et al. 1987) or *gene duplication* (Amores et al. 1998; Carroll 1995; Force et al. 1999; Martin 1999). If new genes could just randomly insert themselves in positions on the genome without any indication of which gene is which, life would never have succeeded, because the variable-length genome problem would damage a significant percentage of offspring. There needed to be some way to keep crossover orderly, so the *right* genes crossed with the right genes. Watson et al. (1987) describe nature's solution:

> . . . it is obvious what mechanism most precisely aligns DNA molecules crossing over because we can hardly imagine any other: Complementary base pairing between strands unwound from two different chromosomes puts the chromosomes in exact register. Crossing over thus generates *homologous recombination*; that is, it occurs between 2 regions of DNA containing identical or nearly identical sequences

Nature uses homology to align genes, or, as Watson et al. say, to put them in exact register. Just as in NEAT, nonequivalent strings of genes can be crossed over by pairing up genes that are homologous, as Watson et al. say later on:

> . . . crossing over also can be detected between homologous segments in nonequivalent regions, as long as the recombinant survives. . . . In the laboratory, recombination between homologous segments on different DNAs is now used to construct new genetic variants.

This mechanism has been observed in *E. coli*, and is now well understood (Sigal and Alberts 1972; Radding 1982). A special protein called *RecA protein* takes a single strand of DNA

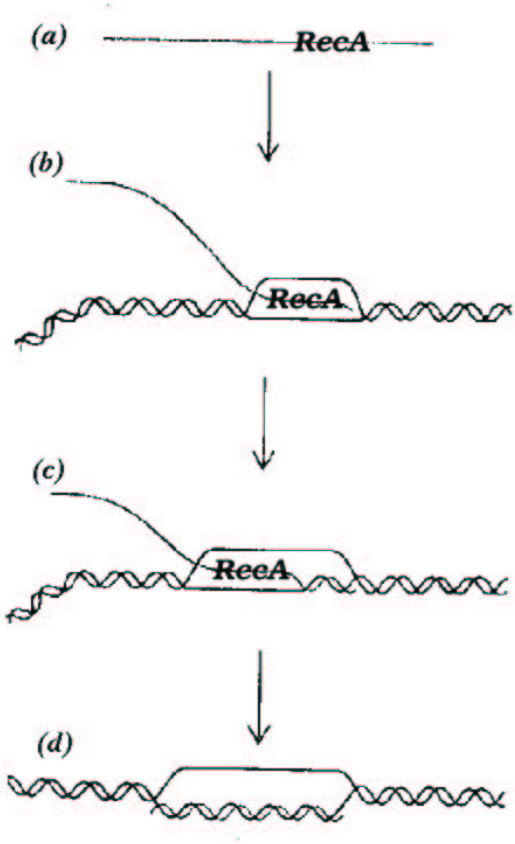


Figure 2.5: **Matching homologous segments of DNA.** (a) RecA protein binds to a single strand of DNA that was cut from its original helix. (b) RecA protein, still attached to the single strand, melts regions of duplex DNA, removing attachments. (c) RecA protein searches for homology. Once found, RecA protein begins annealing. (d) RecA protein is displaced, leaving a ternary complex of 3 strands. The old complementary strand is soon displaced. This process, called *synapsis*, ensures that only homologous genes are aligned during crossover of variable-length genomes. Graphic is modified from Watson et al. (1987), p. 317, with permission from Pearson Education, Inc.

and anneals it to another strand by sticking together parts that are homologous (figure 2.5). The second strand is still attached to its original complementary strand, which is displaced by the newly annealed strand. For a brief time there is a ternary complex of 3 strands before the old complementary strand is displaced. The new single strand is actually cut out of its original helix and transfered to the other helix. The process by which RecA protein puts homologous molecules in register is called *synapsis*. In experiments in vitro, researchers have found that RecA protein will not displace annealed fragments when they are not homologous (Radding 1982).

One may wonder why NEAT uses historical markings on genes in lieu of genetic homology.

Why not just use structural homology as nature does? In fact, some evolutionary algorithms do attempt to match up homologous substructures, such as program trees (Koza 1992), during crossover (**?**). However, unlike in program trees, in TWEANNS the network structure is a sparsely connected graph, and it is difficult to determine which substructures align in arbitrary graphs (Gold and Rangarajan 1996). In addition, it is important not to confuse the "structure" in a genome with the structure of a phenotype. In biological organisms, the DNA itself has a structure that can be used to match up homologous regions. However, in TWEANNs, even if a genome *represents* a graph, the *genome* still has no analogous structure if it is linear. Nature does not put genetic regions into exact register by analyzing the phenotypic features they express and then going back and lining up areas that express the same features. The homology is in the code itself. Where the connections match up is exactly what we *do not know* in the first place, so trying to match up genomes by the connections they describe is begging the question.

However, by keeping track of when every gene in the system first appeared, it is possible to know exactly which genes match up without the need for any topological analysis of the phenotype. Such historical marking is the RecA protein of NEAT, and the resultant matching up of corresponding genes is *artificial synapsis*. This approach is shown effective for TWEANNs in this dissertation. However, it may be an effective alternative to homologous crossover in genetic programming as well (**?**).

Matching up genes on variable length genomes is only one problem faced by TWEANNs. Another problem is how to protect new innovations long enough so they can reach their potential, as will be discussed next.

2.5 Protecting Innovation

Whether at a university, a company, or in society at large, a reasonable way to encourage innovation is to give new ideas a chance to reach their potential before counting them out. If every risky project or unusual proposal were denied at the outset, many great ideas would never achieve fruition. It turns out that the general philosophy that innovative ideas need protection applies to TWEANNs as well, as this section will explain.

2.5.1 The Problem: New Structure Can Damage Fitness

In TWEANNs, innovation means adding new structure to networks through structural mutation. Frequently, adding new structure initially causes the fitness of a network to decrease. For example, adding a new node introduces a nonlinearity where there was none before. It is unlikely a new node just happens to express a useful function as soon as it is introduced. Some generations are required to optimize the new structure and make use of it. Unfortunately, because of the initial loss of fitness caused by the new structure, it is unlikely that the innovation will survive in the population long enough to be optimized. The altered network will probably not be allowed to reproduce because of

its lowered fitness. Thus, it is necessary to somehow *protect* networks with structural innovations so they have a chance to make use of their new structure.

GNARL (Angeline et al. 1993) addresses this problem by adding nonfunctional structure. A node is added to a genome without any connections, in the hopes that in the future some useful connections will develop. However, creating nonfunctional structures wastes search effort and space.

Nature offers a better approach to protecting innovation. In nature, different structures tend to be in different species that compete in different niches. Thus, innovation is implicitly protected within a niche such that there is not as much need to hold back on actually implementing new ideas. If networks with innovative structures could be protected from competing with the population at large by isolating them in their own species, they would have a chance to optimize their structures without interference:

Principle of Protection of Innovation Speciating a population allows organisms to compete only within their niche, protecting their innovations from losing out to organisms in older species.

Thus, NEAT uses speciation to protect innovation, so it *never* adds extraneous structure that may or may not become integrated into a phenotype in future generations.

2.5.2 Speciation

Speciation, also known as *niching*, has been studied in GAs, but has rarely been applied to NE. Speciation is usually used for multimodal function optimization (Mahfoud 1995). In these problems, a function has multiple optima, and a GA is used to find those optima. Other kinds of problems naturally branch from multimodal function optimization, like classification problems: Different classifiers can be found in different parts of fitness space. In such problems, the population as a whole rather than individuals is used to make decisions so it is imperative to keep individuals from converging into a single solution. Speciation is also useful for preventing premature convergence, i.e. the entire population getting stuck on a suboptimal solution, by enforcing that the population remain diverse in single solution problems.

Speciation has also been applied in the cooperative coevolution of modular systems of multiple solutions (Darwen and Yao 1996; Potter and De Jong 1995). Solutions for different situations can be found in different parts of the search space, and a gating algorithm decides which species should be used for which situation.

Curiously, the idea of speciation has not been brought into the field of TWEANNs, perhaps because it is difficult to measure compatibility between networks of different topologies. Such a measure is necessary to tell whether two genomes should be in the same species or not. The variable-length genome problem makes measuring compatibility particularly problematic because networks that compute the same function can appear very different.

In TWEANNs, the goal is not necessarily to coevolve multiple solutions. However, speciation is useful for a *different reason* in TWEANNs. It can maintain species with differing base topologies, allowing very different structures to flourish simultaneously. In fact, empirical results in this dissertation indicate that evolving without speciation can significantly limit NEAT's ability to add and maintain new topology (Section 4.3). Speciation allows solutions that might initially appear less fit to procreate and compete within their own niches, ensuring that they get the chance they deserve. In addition, without speciation, interbreeding among incompatible structures will hinder evolution.

A number of general methods have been developed to maintain species in GAs. Some methods maintain different species temporally such that different solutions flourish at different times (Beasley et al. 1993). Other methods maintain species simultaneously in the population, which more accurately reflects speciation in biology. For example, De Jong (1975) introduced the idea of *crowding*. In crowding, offspring are chosen to replace *similar* organisms in the current population. That way, new solutions do not interfere with completely different solutions in the population. Thus, only a fraction of the population is replaced at each generation. Such a GA is called a *steady-state GA*. One problem with the original crowding method is that it does not encourage diversity beyond the diversity of the original population, making it less useful for a system like NEAT that strives to search through new topologies that did not exist in the initial population.

Mahfoud (1994) introduced a refinement called *deterministic crowding* that uses a tournament between parents and their offspring to determine who gets to continue in the next generation. It is based on the idea that children tend to be similar to their parents. Even though this enhanced crowding method makes the population more diverse than standard crowding, it was not chosen for NEAT because NEAT is designed to work as a steady-state *or* a generational GA. I did not want the system to be restricted to a single reproductive scheme. In addition, although deterministic crowding maintains diversity it does not prevent incompatible topologies from crossing over, as can happen in TWEANNs. If incompatible topologies do mate, the tournament is likely a waste of time.

Holland (1975) introduced the method of *fitness sharing*. Under this method, individuals sharing a niche are forced to *share* the payoff of the niche. Fitness sharing can be implicit or explicit. In implicit fitness sharing, solutions that overlap in their performance are forced to share the payoff of the regions where they overlap. This idea works well for classifier systems, where solutions that classify the same examples correctly share the payoff for those examples, encouraging them to diversify. Implicit sharing may be useful for classification, but it is not always clear how performances might overlap in non-classification problems. More appropriate to TWEANNs is explicit fitness sharing (Goldberg and Richardson 1987). The explicit version forces similar *individuals* to share their payoff. This method is well suited for TWEANNs, assuming that there is some definition of "similarity." Since NEAT provides such a definition (Section 3.3), explicit fitness sharing was chosen as the speciation method in NEAT.

The general definition of explicit fitness sharing is as follows. An individual i shares fitness

with the rest of the population according to a sharing function sh that evaluates the degree to which two individuals i and j must share depending on their distance d. If f_i is i's fitness before sharing, then after sharing it becomes f'_i:

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} \mathrm{sh}(\delta(i,j))}. \tag{2.3}$$

The sharing function sh is set to 0 when distance δ is above some threshold t. Otherwise, sh returns a nonzero value that depends on how close i and j are. The function sh can be chosen by the experimenter to obtain the desired dynamics. Yin and Germay (1993) cluster organisms into species before applying sharing. NEAT does this as well by using a compatibility measure to determine whether two organisms are in the same species or not. There are many advantages to preclustering species. When species are clustered, they can be identified and tracked, which helps in visualizing progress and in keeping track of stagnating subpopulations. It also means that the sharing function only has to operate over individuals in the same species (as opposed to the entire population).

NEAT's use of explicit fitness sharing is one of its main features. The important point is not that fitness sharing is a new idea or better implemented in NEAT than before; it is that NEAT is the first time that fitness sharing has ever been used in TWEANNs in support of protecting topological innovation. Most importantly, what really sets NEAT apart is that it provides a distance metric between different topologies that has never been available before, so that now, for the first time, it will be possible to speciate and evolve topology and weights in an orderly, well-specified manner according to the Principle of Protecting Innovation.

2.6 Initial Populations and Topological Innovation

Many TWEANN methods follow the philosophy that the initial population should start with random topologies. A random initial population ensures that the topologies are diverse from the start. However, random initial populations cause several problems for TWEANNs, and therefore may not be the best way to start evolution.

Randomizing initial topologies can produce infeasible networks. However, there is a more subtle problem with starting randomly that is more serious. As this dissertation will show, it is desirable to find minimal solutions since they reduce the number of parameters being searched. Starting out with random topologies does not lead to minimal solutions, since the population already has many unnecessary nodes and connections. None of these nodes or connections have had to withstand a single evaluation, meaning their configuration is not justified. Effort to minimize networks would have to be spent getting rid of apparatus that should not have been there in the first place, and recombining different topologies does not push towards minimization on its own. Since there is no fitness cost in creating larger networks, they will dominate as long as they have high fitness.

It is possible to incorporate network size into the fitness function, and some TWEANNs actually do this (Zhang and Muhlenbein 1993). However, if the population began with no hidden nodes and grew structure only as it benefited the solution, there would be no need for ad hoc fitness tampering to minimize networks. In fact, if the population started with minimal structure instead of a random population, it would be pointless to add penalties to the fitness function for extra structure. Starting with a minimal population and growing structure is an important design principle in NEAT.

As if to confirm that starting with random topologies leads to unnecessarily large networks, Pujol and Poli (1997) added a pruning algorithm to their PDGP system "in order to find solutions with minimum complexity." PDGP runs this completely separate pruning algorithm *after* it finds a solution. Evidently, PDGP alone is not evolving minimal networks.

Why is it so important to evolve minimal networks? TWEANN researchers generally follow the argument of Zhang and Muhlenbein (1993), who claim, "Finding a minimal network for particular applications is important because the speed and accuracy of learning are dependent on the network complexity." Although there may be some truth to this claim, it is slightly misleading. After all, weight optimization is done by the time the solution is output from a TWEANN anyway, so what learning is left to be done? Some local optimization in weight space may indeed be easier (using backpropagation perhaps) with a minimal architecture, but this benefit seems overstated, considering that most TWEANNs are presented as an *alternative* to backpropagation.

What *really* matters about topology is *not* the final topology, but rather the *intermediate topologies* that the system must search through in order to get to the final solution. The number of connections specified by a chromosome measures the dimensionality of the search space currently being explored. Higher dimensional spaces take longer to search than lower dimensional spaces (Empirical results supporting this assertion are presented in Sections 4.2.3 and 5.6). If the search can be kept to a minimum dimensionality over an entire run, the method will be searching in the simplest possible spaces. TWEANNs should be able to take advantage of this capability, though prior systems did not. NEAT is designed to build from a minimal starting point such that topological additions only survive if they end up enhancing fitness. Therefore, NEAT minimizes the search space for an entire run.

Another reason why starting randomly is dangerous is that adding or subtracting connections from a network can radically change the fitness landscape being searched. The relationships between different-dimensional spaces has rarely been analyzed in TWEANN research. However, Zhang and Muhlenbein (1993) do provide an insightful anecdote on this topic:

> For XOR with a minimal network architecture ($d = 9$) all global minima are isolated; no neighbors are a global optimum. But for the enlarged search space ($d = 13$), there is a chance of 19.2% that another global optimum can be reached by one bit mutation

where d is the number of connections. This interesting observation gives us a clue about how TWEANNs should behave. In certain minimal spaces, the probability of finding a global optimum

from a random point in the search space may be low enough to make the search difficult. However, Zhang and Muhlenbein (1993) also mention that in a different experiment with the goal of evolving an OR network, growing the network actually reduces the number of global optima in the search space by 0.2%. Just by adding a connection, the probability of finding a global optimum can go up significantly in some problems, but the search space can also be made worse in others. Thus, a general principle emerges combining the idea of starting out minimally with growing incrementally:

Principle of Topological Innovation A population should start out with minimal structure and grow incrementally to improve the likelihood of finding a global optimum.

The point is that combining structure minimization and smart growth is the right combination for TWEANNs. Therefore, initial populations should be minimal rather than consist of random topologies with unjustified topological elaborations. Prior TWEANNs did not follow this principle; instead, they simply permute through different combinations of random structures, shifting haphazardly through different non-minimal spaces.

2.6.1 Biological Complexification

In this dissertation, complexification refers to expanding the dimensionality of the search space while preserving the values of the majority of dimensions. In other words, complexification *elaborates* on the existing strategy by adding new structure without changing the existing representation. Thus the strategy does not only become different, but the number of possible responses to situations it can generate increases (figure 2.6).

As mentioned in 2.4.4, biological genomes also experience a process of growth over generations through gene duplication. Gene duplication is a special kind of mutation in which one or more parental genes are copied into an offspring's genome more than once. The offspring then has redundant genes expressing the same proteins. Gene duplication has been responsible for key innovations in overall body morphology over the course of natural evolution (Amores et al. 1998; Carroll 1995; Force et al. 1999; Martin 1999). Because this process of complexification has produced important innovations in biological organisms, and NEAT tries to capitalize on the same mechanism, it is useful to review some of its natural characteristics

A major gene duplication event occurred around the time that vertebrates separated from invertebrates. The evidence for this duplication centers around *HOX genes*, which determine the fate of cells along the anterior-posterior axis of embryos. HOX genes are crucial in determining the order and timing of the overall pattern of development in growing embryos. In fact, differences in HOX gene regulation account for a great deal of the diversity of body plans among arthropods and tetrapods (Carroll 1995). Invertebrates have a single HOX cluster while vertebrates have four, suggesting that cluster duplication significantly helped to elaborate vertebrate body-plans by providing more space in the DNA for representing the organization of body parts (Amores et al. 1998; Holland et al. 1994; Sidow 1996; Nadeau and Sankoff 1997; Postlethwait et al. 1998). The additional HOX

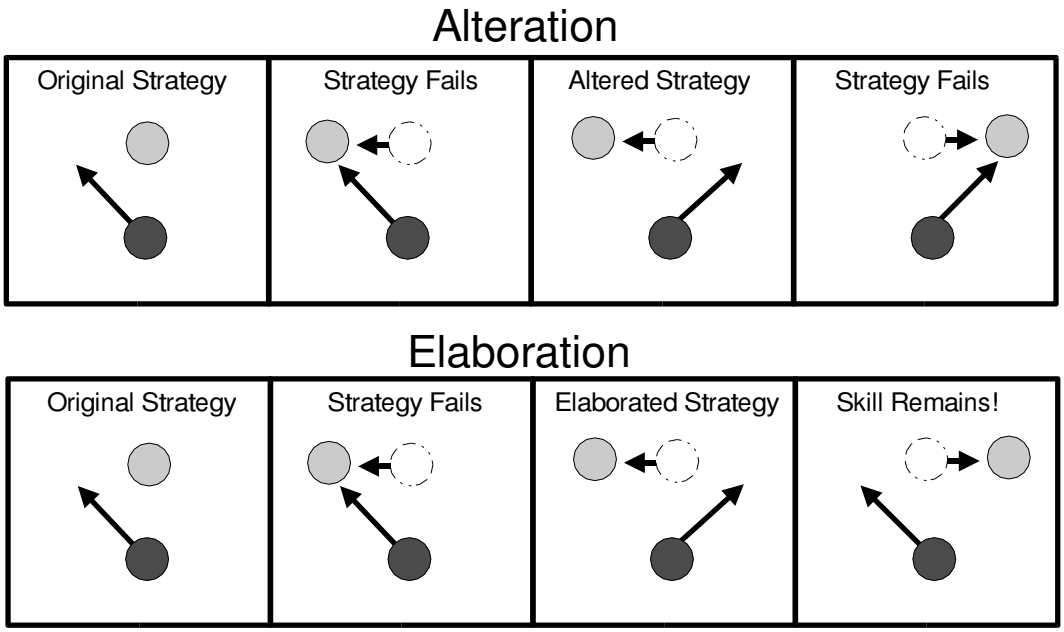


Figure 2.6: **Alteration vs. elaboration example.** The dark robot must evolve to avoid the lighter robot, which attempts to cause a collision. In the alteration scenario (top), the dark robot first evolves a strategy to go around the left side of the opponent. However, the strategy fails in a future generation when the opponent begins moving to the left. Thus, the dark robot alters its strategy by evolving the tendency to move right instead of left. However, when the light robot later moves right, the new, altered, strategy fails because the dark robot did not retain its old ability to move left. In the elaboration scenario (bottom), the original strategy of moving left also fails. However, instead of altering the strategy, it is *elaborated* by adding a new ability to move right as well. Thus, when the opponent later moves right, the dark robot still has the ability to avoid it by using its original strategy. Elaboration can be achieved through complexification.

genes took on new roles in regulating how vertebrate anterior-posterior axis develops, considerably increasing body-plan complexity. Although Martin (1999) argues that the additional clusters can be explained by many single gene duplications accumulating over generations, as opposed to massive whole-genome duplications, researchers agree that some form of gene duplication did make body-plan elaboration possible.

A detailed account of how duplicate genes can take on novel roles was given by Force et al. (1999): Base pair mutations in the generations following duplication *partition* the initially redundant regulatory roles of genes into separate classes. Thus, the embryo develops in the same way, but the genes that determine the overall body-plan are confined to more specific roles, since there are more of them. The partitioning phase completes when redundant clusters of genes are separated enough so that they no longer produce identical proteins at the same time.

After partitioning, mutations within the duplicated cluster of genes affect different steps in

development than mutations within the original cluster. In other words, duplication creates more points at which mutations can occur. In this manner, developmental processes complexify.

Gene duplication is a possible explanation how natural evolution expanded the size of genomes throughout evolution, and provides inspiration for adding new genes to artificial genomes as well. Expanding the length of the size of the genome has motivated previous work in evolutionary computation (Cliff et al. 1993; Koza 1995; Harvey 1993; Lindgren and Johansson 2001). NEAT advances this idea by making it possible to search a wide range of increasingly complex network topologies that are protected in their own niches simultaneously. When evolving neural networks, this process means adding new neurons and connections to the networks. In NEAT, while the process of biological duplication is not strictly followed, it serves as inspiration for the structure-adding mutations that expand the NEAT genome.

Complexification is especially powerful in open-ended domains where the goal is to continually generate more sophisticated strategies (Chapter 5 describes such open-ended evolution with NEAT). Competitive coevolution is a particularly important such domain, as will be reviewed in the next section.

2.7 Competitive Coevolution and Complexification

In *coevolution*, fitness is only measured relative to other members of the population, i.e. it is evaluated through an interaction between two or more individuals. In *cooperative coevolution* several individuals for example play on a team together in order to determine their fitnesses. In contrast, in *competitive coevolution*, individuals play *against* each other, and winning increases fitness. Competitive coevolution is generally set up such that two or more populations of individuals evolve simultaneously in an environment where an increased fitness in one population leads to a decreased fitness for another. Ideally, competing solutions will continually outdo one another, leading to an "arms race" of increasingly better solutions (Dawkins and Krebs 1979; Rosin 1997; Van Valin 1973).

Competitive coevolution has traditionally been used in two kinds of problems. First, interactive behaviors have been evolved with competitive coevolution that are difficult to express in terms of an absolute fitness function. For example, Sims (1994) evolved simulated 3D creatures that attempted to capture a ball before an opponent did, resulting in a variety of effective interactive strategies. Second, coevolution has been used to gain insight into the dynamics of game-theoretic problems. For example, Lindgren and Johansson (2001) coevolved iterated Prisoner's Dilemma strategies in order to demonstrate how they correspond to stages in natural evolution.

In any competitive coevolution experiment, interesting strategies will only evolve if the arms race continues for a significant number of generations. In practice, it is difficult to establish such an arms race. Evolution tends to find the simplest solutions that can win, meaning that strategies can switch back and forth between different idiosyncratic yet uninteresting variations (Floreano

and Nolfi 1997; Darwen 1996; Rosin and Belew 1997). Several methods have been developed to encourage the arms race (Rosin and Belew 1997; Angeline and Pollack 1993; Ficici and Pollack 2001; Noble and Watson 2001). For example, a "hall of fame" or a collection of past good strategies can be used to ensure that current strategies remain competitive against earlier strategies. Recently, Ficici and Pollack (2001) and Noble and Watson (2001) introduced a promising method called *Pareto coevolution*, which finds the best learners and the best teachers in two populations by casting coevolution as a multiobjective optimization problem. This information enables choosing the best individuals to reproduce, as well as maintaining an informative and diverse set of opponents.

Although such techniques allow sustaining the arms race longer, they do not directly encourage *continual coevolution*, i.e. creating new solutions that maintain existing capabilities. For example, no matter how well selection is performed, or how well competitors are chosen, if the search space is fixed, a limit will eventually be reached. Also, it may be difficult to escape local optima in a fixed space without the capacity to add new dimensions.

For these reasons, complexification is a natural technique for establishing a coevolutionary arms race. Complexification elaborates strategies by adding new dimensions to the search space. Thus, progress can be made indefinitely long: Even if a global optimum is reached in the search space of solutions, new dimensions can be added, opening up a higher-dimensional space where even better optima may exist. Coevolution with NEAT is tested in a competitive robot duel domain in Chapter 5.

2.8 Conclusion

Neuroevolution is a promising technique that faces several significant challenges to evolving both topology and weights. NEAT is a TWEANN designed to address these challenges and thereby allow neuroevolution to tackle problem that were not possible to solve in the past.

In existing TWEANNs, structures do not match up in meaningful ways during crossover or comparison. NEAT solves this problem by marking genes in chromosomes with their historical origin, so that they can be matched up in the future. The idea of keeping track of which genes match up using historical marking is motivated by the use of *homology* in biology to identify areas of genomes that should line up.

Many innovative structures are lost in TWEANNs because adding structure can cause an initial loss of fitness. NEAT addresses the problem of protecting innovation by speciating the population using *explicit fitness sharing*.

Existing TWEANNS starting with a random initial population. However, starting randomly forces NE to optimize many more parameters than are necessary to express a solution. Thus, NEAT starts out with a population of minimal structures and adds structure as necessary in order to minimize the number of parameters being searched. The resulting process of gradually increasing complexity is called *complexification*.

Complexification is a powerful mechanism that has proved useful in biological evolution. Because it allows elaboration, complexification can help encourage an arms race in competitive coevolution.

These principles, none of which have been addressed in past TWEANNs, are the basis of NEAT's approach. We now turn to the details of the implementation of NEAT.

Chapter 3

NeuroEvolution of Augmenting Topologies (NEAT)

An efficient system capable of evolving complex structures at the same time as weights should be based upon the three principles discussed in the background: The system should implement a method of detecting homology between genes, it should protect innovation, and it should minimize structure in order to minimize the number of dimensions being searched. The NEAT approach follows these principles. Historical markings are used as a way of identifying homology, speciation accomplishes the protection of innovation, and minimizing structure is accomplished by starting out with a population of networks with no hidden nodes.

This section begins with an overview of the genetic encoding used in NEAT. Using the genetic encoding, structural mutations are introduced to make it clear how genomes grow in NEAT. Historical markings, which are applied whenever a genome grows, are then explained in detail. Crossover, which uses historical markings to allowing disparate topologies to be combined, is discussed next. NEAT's approach to speciation using fitness sharing is introduced as a way to protect innovation, and the last section explains growth from a minimal starting point.

3.1 Genetic Encoding

Evolving structure requires a flexible genetic encoding. In order to allow structures to complexify, their representations must be dynamic and expandable. Each genome in NEAT includes a list of *connection genes*, each of which refers to two *node genes* being connected (Figure 3.1). Each connection gene specifies the in-node, the out-node, the weight of the connection, whether or not the connection gene is expressed (an enable bit), and an *innovation number*, which allows finding corresponding genes during crossover.

Mutation in NEAT can change both connection weights and network structures. Connection weights mutate as in any NE system; each connection weight is perturbed with a fixed probability by

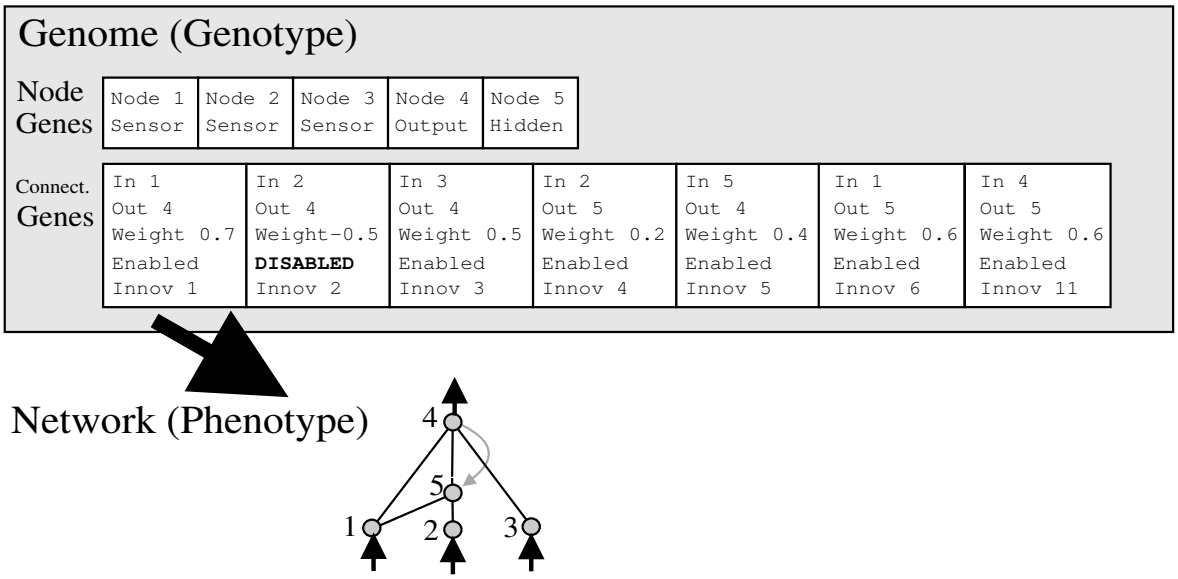


Figure 3.1: **A NEAT genotype to phenotype mapping example.** A genotype is depicted that produces the shown phenotype. There are 3 input nodes, one hidden, one output node, and seven connection definitions, one of which is recurrent. The second gene is disabled, so the connection that it specifies (between nodes 2 and 4) is not expressed in the phenotype. In order to allow complexification, genome length is unbounded.

adding a floating point number chosen from a uniform distribution of positive and negative values. Structural mutations, which form the basis of complexification, occur in two ways (Figure 3.2). Each mutation expands the size of the genome by adding genes. In the *add connection* mutation, a single new connection gene is added connecting two previously unconnected nodes. In the *add node* mutation, an existing connection is split and the new node placed where the old connection used to be. The old connection is disabled and two new connections are added to the genome. The connection between the first node in the chain and the new node is given a weight of one, and the connection between the new node and the last node in the chain is given the same weight as the connection being split. Splitting the connection in this way introduces a nonlinearity (i.e. sigmoid function) where there was none before. Because the new node is immediately integrated into the network, its effect on fitness can be evaluated right away. Preexisting network structure is not destroyed and performs the same function, while the new structure provides an opportunity to elaborate on the original behaviors.

Through mutation, the genomes in NEAT will gradually get larger. Genomes of varying sizes will result, sometimes with different connections at the same positions. Crossover must be able to recombine networks with differing topologies, which can be difficult (Radcliffe 1993). The next section explains how NEAT approaches this problem.

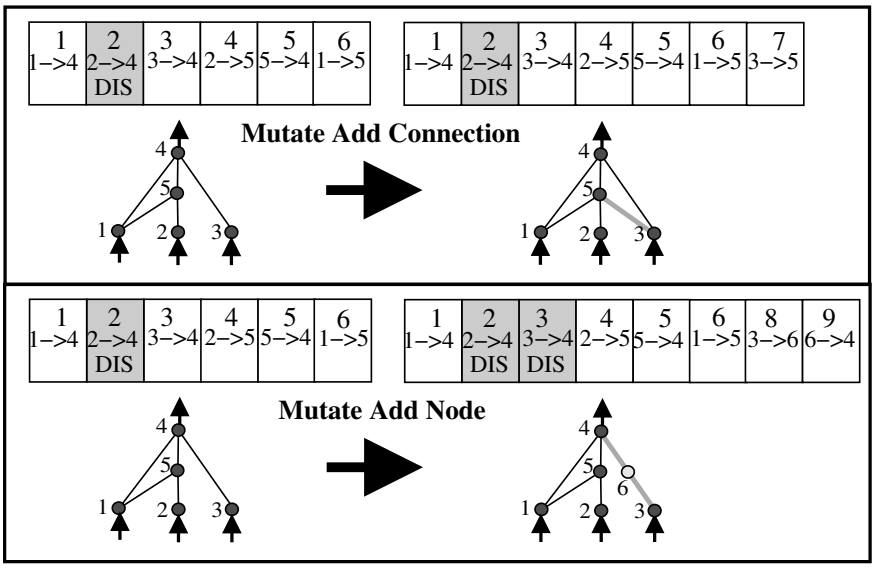


Figure 3.2: **The two types of structural mutation in NEAT.** Both types, adding a connection and adding a node, are illustrated with the genes above their phenotypes. The top number in each genome is the *innovation number* of that gene. The bottom two numbers denote the two nodes connected by that gene. The weight of the connection, also encoded in the gene, is not shown. The symbol DIS means that the gene is disabled, and therefore not expressed in the network. The figure shows how connection genes are appended to the genome when a new connection and a new node is added to the network. Assuming the depicted mutations occurred one after the other, the genes would be assigned increasing innovation numbers as the figure illustrates, thereby allowing NEAT to keep an implicit history of the origin of every gene in the population.

3.2 Tracking Genes through Historical Markings

It turns out that the historical origin of each gene can be used to tell us exactly which genes match up between *any* individuals in the population. Two genes with the same historical origin represent the same structure (although possibly with different weights), since they were both derived from the same ancestral gene at some point in the past. Thus, all a system needs to do is to keep track of the historical origin of every gene in the system.

Tracking the historical origins requires very little computation. Whenever a new gene appears (through structural mutation), a *global innovation number* is incremented and assigned to that gene. The innovation numbers thus represent a chronology of every gene in the system. As an example, let us say the two mutations in Figure 3.2 occurred one after another in the system. The new connection gene created in the first mutation is assigned the number 7, and the two new connection genes added during the new node mutation are assigned the numbers 8 and 9. In the future, whenever these genomes cross over, the offspring will inherit the same innovation numbers on each gene. Thus, the historical origin of every gene in the system is known throughout evolution.

A possible problem is that the same structural innovation will receive different innovation

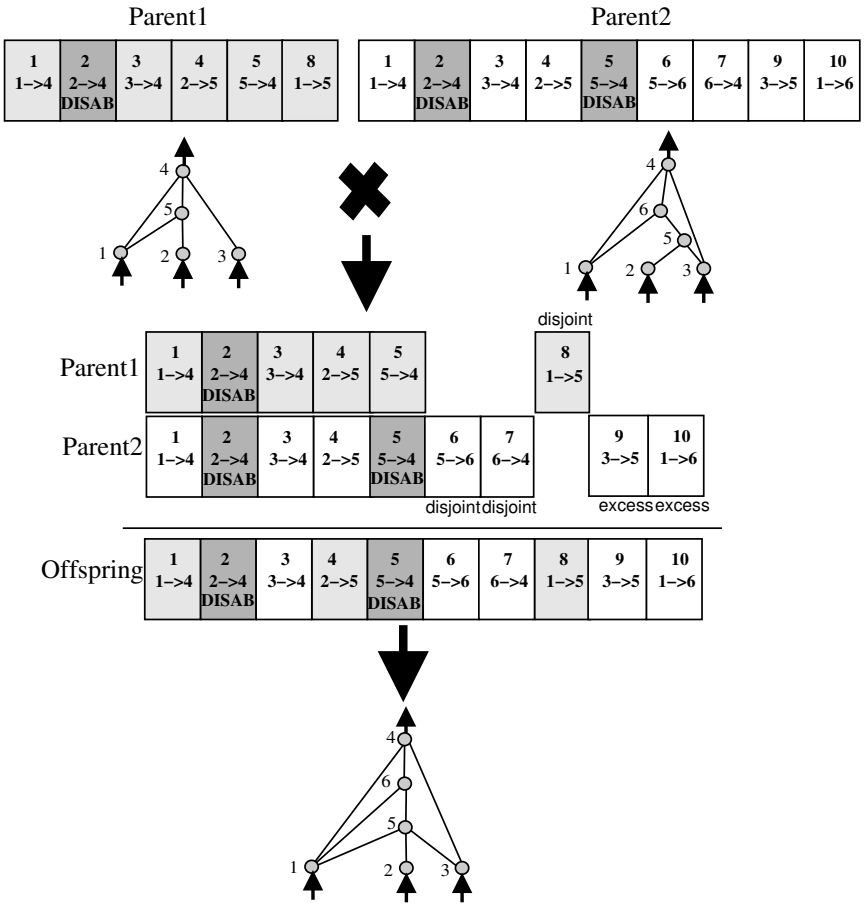


Figure 3.3: **Matching up genomes for different network topologies using innovation numbers.** Although Parent 1 and Parent 2 look different, their innovation numbers (shown at the top of each gene) tell us that several of their genes match up even without topological analysis. A new structure that combines the overlapping parts of the two parents as well as their different parts can be created in crossover. In this case, equal fitnesses are assumed, so each disjoint and excess gene is inherited from either parent randomly. Otherwise the genes would be inherited from the more fit parent. The disabled genes may become enabled again in future generations: There is a preset chance that an inherited gene is enabled if it is disabled in either parent.

numbers in the same generation if it occurs by chance more than once. However, by keeping a list of the innovations that occurred in the current generation, it is possible to ensure that when the same structure arises more than once through independent mutations in the same generation, each identical mutation is assigned the same innovation number. Extensive experimentation established that resetting the list every generation as opposed to keeping a growing list of mutations throughout evolution is sufficient to prevent innovation numbers from exploding.

Through innovation numbers, the system now knows exactly which genes match up with

which (Figure 3.3). Genes that do not share the same historical marking as any gene in other other parent genome are *disjoint*. Genes that appeared in one parent later in evolution than any genes in the other parent are *excess*. When crossing over, the genes with the same innovation numbers are lined up and crossed over in one of two ways. In the first method, matching genes are randomly chosen for the offspring genome. Alternatively, the connection weights of matching genes can be averaged (Wright (1991) reviews both types of crossover and their merits). NEAT uses both types of crossover. Disjoint and excess genes are inherited from the more fit parent, or if they are equally fit, each gene is inherited from either parent randomly. Disabled genes have a chance of being reenabled during crossover, allowing networks to make use of older genes once again.

Historical markings allow NEAT to perform crossover without analyzing topologies. Genomes of different organizations and sizes stay compatible throughout evolution. This methodology allows NEAT to complexify structure while different networks still remain compatible. However, it turns out that it is difficult for a population of varying topologies to support new innovations that add structure to existing networks, Because smaller structures optimize faster than larger structures, and adding nodes and connections usually initially decreases the fitness of the network, recently augmented structures have little hope of surviving more than one generation even though the innovations they represent might be crucial towards solving the task in the long run. The solution is to protect innovation by speciating the population, as explained in the next section.

3.3 Protecting Innovation through Speciation

NEAT speciates the population so that individuals compete primarily within their own niches instead of with the population at large. This way, topological innovations are protected and have time to optimize their structure before they have to compete with other niches in the population. In addition, speciation prevents bloating of genomes: Species with smaller genomes survive as long as their fitness is competitive, ensuring that small networks are not replaced by larger ones unnecessarily. Protecting innovation through speciation follows the philosophy that new ideas must be given time to reach their potential before they are eliminated.

Using historical markings, the system can determine how much history is shared among different genomes and use that information to divide the population into species. The distance δ between two network encodings can be measured as a linear combination of the number of excess (E) and disjoint (D) genes, as well as the average weight differences of matching genes ($\overline{W}$):

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}. \tag{3.1}$$

The coefficients c_1, c_2, and c_3 adjust the importance of the three factors, and the factor N, the number of genes in the larger genome, normalizes for genome size (N can be set to 1 unless both genomes are excessively large).

Throughout evolution, NEAT maintains a list of species numbered in the order they appeared. In the first generation, since there are no preexisting species, NEAT begins by creating species 1 and placing the first genome into that species. All other genomes are placed into species as follows: A random member of each existing species is chosen as its permanent *representative*. Genomes are tested one at a time; if a genome's distance to the representative of any existing species is less than δ_t, a compatibility threshold, it is placed into this species. Otherwise, if it is not compatible with any existing species, a new species is created and given a new number. After the first generation, genomes are first compared with species from the *previous generation* so that the same species numbers can be used to identify species throughout the run. Keeping the same set of species from one generation to the next allows NEAT to remove stagnant species, i.e. species that have not improved for too many generations. In general, because structure is added slowly and only useful innovations survive in the long run, throughout evolution topologies within the same species in the same generation tend to be similar. The problem of choosing the best value for δ_t can be avoided by making δ_t *dynamic*; that is, given a target number of species, the system can slightly raise δ_t if there are too many species, and lower δ_t if there are too few.

Let P be the entire population. The algorithm for clustering genomes into species follows:

- The Genome Loop:
  - Take next genome g from P
  - The Species Loop:
    - If all species in S have been checked, create new species s_{new} and place g in it
    - Else
      - get next species s from S
      - If g is compatible with s, add g to s
    - If g has not been placed, Species Loop
  - If not all genomes in G have been placed, Genome Loop
  - Else STOP

As the reproduction mechanism, NEAT uses *explicit fitness sharing* (Goldberg and Richardson 1987), where organisms in the same species must share the fitness of their niche. Thus, a species cannot afford to become too big even if many of its organisms perform well. Therefore, any one species is unlikely to take over the entire population, which is crucial for speciated evolution to support a variety of topologies. The adjusted fitness f'_i for organism i is calculated according to its distance δ from every other organism j in the population:

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} \mathrm{sh}(\delta(i,j))}. \tag{3.2}$$

The sharing function sh is set to 0 when distance $\delta(i,j)$ is above the threshold δ_t; otherwise, $\mathrm{sh}(\delta(i,j))$ is set to 1 (Spears 1995). Thus, $\sum_{j=1}^{n} \mathrm{sh}(\delta(i,j))$ reduces to the number of organisms in the same species as organism i. This reduction is natural since species are already clustered by compatibility using the threshold δ_t. Every species is assigned a potentially different number of offspring in proportion to the sum of adjusted fitnesses f'_i of its member organisms. The net effect of fitness sharing in NEAT can be summarized as follows. Let $\overline{F_k}$ be the average fitness of species k and $|P|$ be the size of the population. Let $\overline{F}_{tot} = \sum_k \overline{F_k}$ be the total of all species fitness averages. The number of offspring n_k allotted to species k is:

$$n_k = \frac{\overline{F_k}}{\overline{F}_{tot}}|P|. \tag{3.3}$$

The lowest performing fraction of each species is eliminated. The parents to produce the next generation are chosen randomly among the remaining individuals (uniform distribution with replacement). The highest performing individual in each species, i.e. the *species champions*, carries over from each generation. Otherwise the next generation completely replaces the one before.

The net effect of speciating the population is that structural innovation is protected. The final goal of the system, then, is to perform the search for a solution as efficiently as possible. This goal is achieved through complexification from a simple starting structure, as detailed in the next section.

3.4 Minimizing Dimensionality through Complexification

Unlike other systems that evolve network topologies and weights (Angeline et al. 1993; Gruau et al. 1996; Yao 1999; Zhang and Muhlenbein 1993), all the networks in the first generation in NEAT have the same small topology: All the inputs are directly connected to every output, and there are no hidden nodes. These first generation networks differ only in their initial random weights. Speciation protects new innovations, allowing diverse topologies to gradually accumulate over evolution. Thus, because NEAT protects innovation using speciation, it can start in this manner, minimally, and grow new structure over generations.

New structure is introduced incrementally as structural mutations occur, and only those structures survive that are found to be useful through fitness evaluations. This way, NEAT searches through a minimal number of weight dimensions, significantly reducing the number of generations necessary to find a solution, and ensuring that networks become no more complex than necessary. This gradual increase in complexity over generations is *complexification*. In other words, NEAT searches for the optimal topology by incrementally complexifying existing structure.

The process of complexification and its benefits can be summarized as follows. The system is initially searching a very low-dimensional parameter space with very few connections. Solutions are optimized in this low-dimensional space. The space may not be expressive enough to solve the problem, but locally optimal solutions appear. The system then increases the dimensionality

of the search space by adding new topology through structural mutations. This causes a shift into a new space. However, it is likely that the position of the new genome in the new space is on a promising hill, because most of its connections are already optimized to perform a useful function. The new connections, representing extra dimensions of search, can be optimized to coordinate with the older connections, which themselves may change somewhat in order to adapt to the new higher-dimensional space. Many different innovations happen at the same time, in the same generation, so NEAT is searching many different dimensional search spaces simultaneously. These different spaces are each represented by a species. Each search space is a branch off of a smaller space that was already somewhat optimized. Because solutions tend to start out in larger spaces already near their goal, much of the work of searching has been accomplished in a lower dimensional space, which means less parameters needed to be optimized simultaneously, which allows searching in higher-dimensional space than would otherwise be possible.

3.5 Conclusion

NEAT is based on three principles that work together to efficiently evolve network topologies and weights. The first principle is homology: NEAT encodes each node and connection in a network with a gene. Whenever a structural mutation results in a new gene, that gene receives a historical marking. Historical markings are used to match up homologous genes during crossover, and to define a compatibility operator.

The second principle is protecting innovation. A compatibility operator is used to speciate the population, which protects innovative solutions and prevents incompatible genomes from crossing over.

Finally, NEAT follows the philosophy that search should begin in as small a space as possible and expand gradually. Evolution in NEAT always begins with a population of minimal structures. Structural mutations add new connections and nodes to networks in the population, leading to incremental growth. Topological innovations have a chance to realize their potential because they are protected from the rest of the population by speciation. Because only useful structural additions tend to survive in the long term, the structures being optimized tend to be the minimum necessary to solve the problem.

NEAT's approach allows fast search because the number of dimensions being searched is minimized. The next two chapters demonstrate the power of this process.

Chapter 4

Performance Evaluation

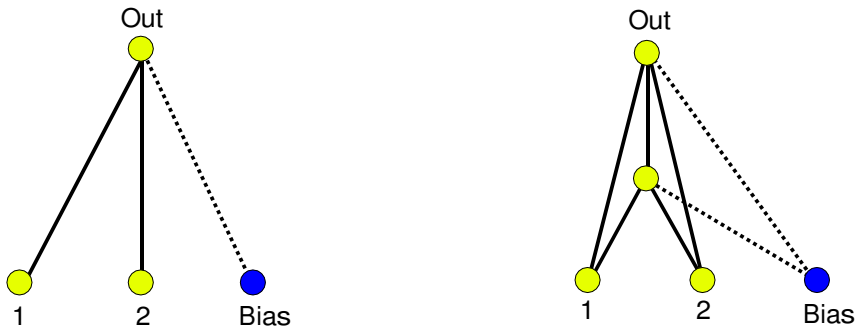
Because NEAT is designed to produce increasingly complex networks, NEAT can potentially discover neural network behaviors that would be inaccessible through search using other methods. It is primarily this potential for novel discovery that this dissertation aims to demonstrate. However, it is important first to establish that the method is *efficient* in comparison with others. In other words, if NEAT is to be applied to open-ended real-world problems, we want to know first whether it can be expected to achieve this objective in a reasonable amount of time.

Three questions must be answered in order to establish that NEAT is efficient: (1) Can NEAT evolve the necessary structures? (2) Can NEAT find solutions more efficiently than other reinforcement learning systems? (3) Are all components of NEAT necessary? The first question establishes that NEAT indeed builds topologies reliably, and it is answered through a simple XOR evolution test. The second question amounts to a systematic comparison, and it is answered through the standard benchmark of balancing two poles attached to a moving cart. The third question requires testing whether all components of NEAT contribute to its performance, which is established through a series of ablation experiments.

4.1 Evolving the Right Topology: XOR

It is important to establish that NEAT can indeed evolve appropriate topologies reliably. For this reason, NEAT is applied to the problem of building an XOR network. Although this task is simple, it requires growing hidden units, and therefore serves as a simple test for the method. The aim is to show that NEAT will grow new structure to cope with problems that require it.

XOR is a binary logical operator that only returns true if one and only one of the two inputs is true. The two inputs to XOR must be combined at a hidden unit, as opposed to only at the output node, because there is no function over a linear combination of the inputs that can separate the inputs into the proper classes. These structural requirements make XOR suitable for testing NEAT's ability to evolve structure. For example, NEAT's method for adding new nodes might be too destructive to



(a) Network from initial population (b) Phenotype of smallest possible solution

Figure 4.1: **Initial phenotype and optimal XOR.** Figure (a) shows the phenotype given to the entire initial population. Notice that there are no hidden nodes. In NEAT, a bias is a node that can connect to any node other than inputs. Figure (b) shows an optimal solution with only 1 hidden node. (A network without hidden nodes cannot compute XOR.) The bias connections are not always needed depending on the solution; All other connections are necessary. The optimal (1 hidden node) solution was found in 22 of 100 runs. The average solution had 2.35 hidden nodes with a standard deviation of 1.11 nodes. NEAT must add at least one new hidden neuron in order to solve this task, making it a good test of NEAT's ability to evolve the right structure.

allow new nodes to get into the population. Or, it could find a local champion with a wrong kind of connectivity that dominates the population so much that the systems fails to evolve the proper connectivity. Third, maybe the changing structure renders past connection weight values obsolete. If so, the algorithm would have trouble enlarging topologies that are already largely specialized. This experiment is meant to show that NEAT is not impeded by such potential obstacles, but can grow structure efficiently and consistently when needed.

To compute fitness, the distance of the output from the correct answer was summed for all four input patterns. The result of this error was subtracted from 4 so that higher fitness would mean better networks. The resulting number was squared to give proportionally more fitness the closer a network was to a solution.

The initial generation consisted of networks with no hidden units (Figure 4.1a). The networks had 2 inputs, 1 bias unit, and 1 output. The bias unit is an input that is always set to 1.0. The network can use the bias to change the activation threshold of neurons. There were three connection genes in each genome in the initial population. Two genes connected the inputs to the output, and one connected the bias to the output. Each connection gene received a random connection weight. NEAT system parameters used in XOR are described in appendix A.

On 100 runs, the first experiment showed that the NEAT system finds a structure for XOR in an average of 32 generations (4,755 networks evaluated, std=2,553). On average a solution network had 2.35 hidden nodes and 7.48 non-disabled connection genes. Since a successful network requires

at least one hidden unit (figure 4.1b) , NEAT actually found very small networks. NEAT was also very consistent in finding a solution: It did not fail once in 100 simulations. The worst performance took 13,459 evaluations, or about 90 generations (compared to 32 generations on average). The standard deviation for number of nodes used in a solution was 1.11, meaning NEAT consistently used 1 or 2 hidden nodes to build an XOR network.

The XOR problem has been used to demonstrate performance of several prior TWEANN algorithms. Unfortunately, quantitative performance comparisons are difficult in this domain because the methodologies vary widely across experiments. For example, several methods evolve network topologies using a separate hill climbing algorithm for weight optimization (Yao and Shi 1995; Zhang and Muhlenbein 1993). Although the PDGP method (Section 2.3.2; Pujol and Poli 1998) evolves weights in addition to topologies, it includes a post-processing module that spends additional generations pruning networks after a solution has already been found. The method sGA (Section 2.3.2; Dasgupta and McGregor 1992), like NEAT, evolves both topologies and weights and does not post-prune; however, the reported results do not include the average number of generations necessary for a solution. Above all, to the best of my knowledge, all previous methods applied to XOR evolution limit the size of a genome. In contrast, NEAT puts no limit on the size or complexity of networks evolved for this problem.

It is clear that NEAT solves the XOR problem without trouble and in doing so keeps the topology small. However, XOR is not a good benchmark for performance comparisons because it is such a simple task. Therefore, having established NEAT's ability to consistently evolve structure, the next section examines how it compares with other methods at more challenging problems.

4.2 Comparing Performance: Pole Balancing

There are many control learning tasks where the techniques employed in NEAT can make a difference. Many of these potential applications, like robot navigation or game playing, present problems without known solutions. The pole balancing domain is used for comparison because it is a known benchmark in the literature, which makes it possible to demonstrate the effectiveness of NEAT compared to others. It is also a good surrogate for real problems, in part because pole balancing in fact *is* a real task, and also because the difficulty can be adjusted. Earlier comparisons were done with a single pole (Moriarty and Miikkulainen 1996), but this version of the task has become too easy for modern methods. Balancing two poles simultaneously is on the other hand challenging enough for all current methods. In the most difficult version of this problem, two poles must be balanced without velocity information. This problem is non-Markovian and provides strong evidence that evolving augmenting topologies is not only interesting because it can find structures, but is also efficient in difficult control tasks.

4.2.1 Method

We set up the pole balancing experiments as described by Wieland (1991) and Gomez and Miikkulainen (1999) (Appendix A.4.1). Two poles are connected to a moving cart by a hinge and the neural network must apply force to the cart to keep the poles balanced for as long as possible without the cart going beyond the boundaries of the track. The system state is defined by the cart position (x) and velocity ($\dot{x}$), the first pole's position (θ_1) and angular velocity ($\dot{\theta}_1$), and the second pole's position (θ_2) and angular velocity ($\dot{\theta}_2$). Control is possible because the poles have different lengths and therefore respond differently to control inputs.

The Runge-Kutta fourth-order method was used to implement the dynamics of the system, with a step size of 0.01s. All state variables were scaled to $[-1.0, 1.0]$ before being fed to the network. Networks output a force every 0.02 seconds between $[-10, 10]N$. The poles were 0.1m and 1.0m long. The initial position of the long pole was 1° and the short pole was upright; the track was 4.8 meters long. The rest of the parameters are described in appendix A.

Two versions of the double pole balancing task were used: one with velocity inputs included and another without velocity information. The first task is Markovian and allows comparing with many different systems. Taking away velocity information makes the task more difficult because the network must estimate an internal state in lieu of velocity, which requires recurrent connections.

NEAT was compared to three traditional value function-based reinforcement learning (RL) methods, one policy search method, and five other NE methods. Although NE methods significantly outperform traditional RL methods, competition among NE methods has encouraged experimenters to begin to exploit idiosyncrasies of pole balancing that may not be representative of other problems. Therefore, this section presents several separate comparisons with NE methods that have been customized for pole balancing. The conclusion is that NEAT performs significantly more efficiently than traditional RL methods and is at least as powerful as the customized NE techniques. The following three sections describe the methods used in the comparisons.

Value Function Methods

Value function methods learn the value of taking specific actions in the current state of the system. The action space is discretized and a function approximator learns a Q-function that assigns a value to state-action pairs.

Q-Learning with a multilayer perceptron (Q-MLP; Watkins and Dayan 1992) uses a feed-forward artificial neural network as its function approximator. The network receives state and action variables as input and outputs a single Q-value.

Sarsa(λ) with a Case-Based function approximator (SARSA-CABA; Santamaria et al. 1998) records state-action pairs explicitly in memory as separate cases. New cases are assigned values by combining the values of neighboring cases.

Sarsa(λ) with a Cerebellar Model Articulation Controller (CMAC; Albus 1975; Singh and

Sutton 1996) replaces the case-based memory of SARSA-CABA with a Cerebellar Model Articulation Controller, which divides the state-action space into a set of overlapping tilings. Q-values are computed by summing values from different overlapping tilings that contain the same feature value.

Value function performance results are reported by Gomez (2003), who used SARSA implementations by Santamaria et al. (1998). Gomez (2003) implemented the Q-Learning system from which Q-Learning results are reported.

Policy Search Methods

Like NE, Value and Policy Search (VAPS; Meuleau et al. 1999) searches the space of policies (as opposed to states and actions) for a solution. However, while NE searches through a space of neural networks, VAPS searches for finite state automaton graphs. In addition, NE evolves a solution while VAPS searches using stochastic gradient descent. In principle, this technique can represent policies that work in non-Markovian environments. VAPS' performance results in pole balancing were reported by Meuleau et al. (1999).

Neuroevolution Methods

NEAT was also compared to published results from five other NE systems. The first two represent standard population-based approaches. Saravanan and Fogel (1995) used Evolutionary Programming, which relies entirely on mutation of connection weights, while Wieland (1991) used both mating and mutation.

The second two systems, SANE (Moriarty and Miikkulainen 1996) and ESP (Gomez and Miikkulainen 1999), evolved populations of neurons and a population of network blueprints that specifies how to build networks from the neurons that are assembled into fixed-topology networks for evaluation. The topologies are fixed because the individual neurons are always placed into predesignated slots in the neural networks they compose. SANE maintains a single population of neurons. ESP improves over SANE by maintaining a separate population for each hidden neuron position in the complete network (Section 2.3.1).

Like NEAT, Cellular Encoding (CE; Gruau *et al.*, 1996) evolves both topologies and weights (Section 2.3.2). The success of CE was first attributed to its ability to evolve structures. However, ESP, a fixed-topology NE system, was able to complete the non-Markovian pole-balancing task five times faster simply by restarting with a random number of hidden nodes whenever it got stuck. Our experiments will show that NEAT's evolution of structure can significantly outperform CE.

CMA-ES (Igel 2003) is an evolution strategy (ES) technique (Beyer and Paul Schwefel 2002) used to evolve fixed-topology networks. It keeps track of which mutations lead to the most gains and uses this information to guide the direction of future mutations (Section 2.3.1). According to Igel (2003), CMA-ES has the advantage of allowing small population sizes; it successfully evolves a very small population of of between 13 and 19 networks. However, subsequent experi-

Method	Evaluations	CPU time (seconds)
Q-MLP	**10,582**	153
SARSA-CABA	Timed Out	–
SARSA-CMAC	Timed Out	–
VAPS	Timed Out	–
Ev. Programming	**307,200**	–
Conventional NE	**22,100**	73
SANE	**12,600**	37
ESP	**3,800**	22
NEAT	**3,600**	31

Table 4.1: **Double pole balancing with velocity information.** The number of evaluations and CPU time required for a solution are shown for both NE methods (bottom five entries) and non-NE methods (top four). Evolutionary programming results were obtained by Saravanan and Fogel (1995). All other results are from Gomez (2003) and averaged over 50 runs; Because the SARSA and VAPS methods could not solve the task within 12 hours, they were timed out (Gomez 2003). NEAT results were averaged over 120 runs with standard deviation of 2,704 evaluations. Although standard deviations for other methods were not reported, if we assume similar variances, all differences are statistically significant ($p < 0.001$), except that between NEAT and ESP. NEAT solves the task in fewer evaluations than any other method, and is significantly faster than traditional RL methods.

mentation with NEAT showed that NEAT also solves the pole balancing problem faster with very small populations (Section 4.2.4). Thus, small populations may fit the pole balancing problem better than previously thought, making comparisons with prior experiments, which all used populations of 100 networks or more, impossible. Therefore, NEAT is first compared with results using standard population sizes, and then compared with CMA-ES by giving NEAT a very small population as well.

4.2.2 Double Pole Balancing with Velocities

The criteria for success on this task was keeping both poles balanced for 100,000 time steps, or 30 minutes of simulated time. A pole was considered balanced between -36 and 36 degrees from vertical. Fitness on this task was measured as the number of time steps that both poles remained balanced.

The populations used by NE methods in this comparison included 100 or more networks: Evolutionary Programming used 2048; conventional NE, 100; SANE and ESP, 200; and NEAT, 150.

Table 4.1 shows that NEAT takes the fewest evaluations to complete this task (the difference between NEAT and ESP is not statistically significant). NEAT took three times fewer evaluations than Q-Learning, while traditional RL techniques could not solve the task. The conclusion is that NEAT is significantly more successful at solving this benchmark than traditional RL methods.

The fixed-topology NE systems evolved networks with 10 hidden nodes because that number was thought to be appropriate for this task. However, NEAT's solutions always used between 0 and 4 hidden nodes. Thus, it is clear that NEAT's minimization of dimensionality is working on this problem and finding the smallest known solutions. This result is important because it shows that NEAT performs as well as ESP while finding more minimal solutions.

4.2.3 Double Pole Balancing Without Velocities

Gruau *et al.* (1996) introduced a special fitness function for this problem to prevent the system from solving the task simply by moving the cart back and forth quickly to keep the poles wiggling in the air. (Such a solution would not require computing the missing velocities.) The fitness penalizes oscillations. It is the sum of two fitness component functions, f_1 and f_2, such that $F = 0.1f_1 + 0.9f_2$. The two functions are defined over 1000 time steps:

$$f_1 = t/1000, \tag{4.1}$$

$$f_2 = \begin{cases} 0 & \text{if } t < 100, \\ \frac{0.75}{\sum_{i=t-100}^{t}(|x^i|+|\dot{x}^i|+|\theta_1^i|+|\dot{\theta}_1^i|)} & \text{otherwise,} \end{cases} \tag{4.2}$$

where t is the number of time steps the poles remain balanced during the 1,000 total time steps. The denominator represents the sum of offsets from center rest of the cart and the long pole. It is computed by summing the absolute value of the state variables representing the cart and long pole positions and velocities. Thus, by minimizing these offsets (damping oscillations), the system can maximize fitness. Because of this fitness function, swinging the poles is penalized, forcing the system to internally compute the hidden state variables.

Under Gruau *et al.*'s criterion for a solution, the champion of each generation is tested on generalization to make sure it is robust. This test takes a lot more time than the fitness test, which is why it is applied only to the champion. In addition to balancing both poles for 100,000 time steps, the winning controller must balance both poles from 625 different initial states, each for 1,000 times steps. The number of successes is called the *generalization performance of the solution*. In order to count as a solution, a network needs to generalize to at least 200 of the 625 initial states. Each start state is chosen by giving each state variable (i.e. x, $\dot{x}$, θ_1, and $\dot{\theta}_1$) each of the values 0.05, 0.25, 0.5, 0.75, 0.95 scaled to the range of the input variable ($5^4 = 625$).

Gomez (2003) reported results for ESP and conventional NE (CNE) with both the standard fitness (i.e. number of time steps before falling) and Gruau's damping fitness with the generalization test. Table 4.2 shows that NEAT and ESP are the fastest systems on this task, with no significant difference between the two with either fitness measure. None of the other NE methods nor any of the value function methods could solve this task.

NEAT takes 34 times fewer evaluations than Gruau's original benchmark, showing that the way in which structure is evolved has significant impact on performance. While NEAT and ESP can

Method	Evaluations	
	Standard Fitness	**Damping fitness**
CE	–	**840,000**
Conventional NE	**76,906**	**87,623**
ESP	**20,456**	**26,342**
NEAT	**20,918**	**24,543**

Table 4.2: **Double pole balancing without velocity information.** CE is Cellular Encoding of Gruau et al. (1996). CNE and ESP results were reported by Gomez (2003). All results are averages over 50 simulations with a population of 150 except CE, which was run only once with a population of 16,384 networks. All differences in number of evaluations are significant ($p < 0.001$) except those between NEAT and ESP. NEAT and ESP solve this task in the fewest evaluations. NEAT is 34 times faster than CE, the only other TWEANN to solve the task.

solve the task with approximately the same number of evaluations, ESP was started with a network with an appropriate number of hidden nodes. Past comparisons showed that if ESP is forced to start with a random number of hidden neurons, NEAT gains a significant advantage (Stanley and Miikkulainen 2002b), suggesting that the capacity to find the right level of complexity for the task is beneficial. There was no significant difference in the ability of any of the four methods to generalize. Finally, both NEAT and ESP are several times faster than conventional neuroevolution with a fixed topology.

4.2.4 Pole Balancing with Very Small Populations

Having a common benchmark such as double pole balancing is useful because it allows measuring whether various improvements to learning algorithms are actually significant in practice. However, such competition has the unfortunate side effect that it is becoming increasingly difficult to separate the contribution of particular methods from the experimental methodologies used to compare them.

When Igel (2003) evaluated their CMA-ES with very small populations of 13 to 19 networks, they obtained solutions in significantly fewer evaluations than other NE methods. However, when the population size in NEAT was similarly reduced, its performance also increased significantly, suggesting that population size has a powerful effect on the number of evaluations required for a solution in pole balancing. This result makes sense since both double pole balancing with and without velocity inputs may be difficult to solve at the start, but once a species is near a solution, the extra evaluations accumulated by other species in a large population are not necessary to solve the problem. Smaller populations can therefore make evolution dramatically faster in this particular task. However, it is important to note that in more difficult or open-ended domains, the situation is likely different, because early in evolution it is not clear which species is close to a solution, and so a number of species must be maintained over the long term. Small populations would actually be harmful in such tasks.

Second, Igel's (2003) algorithm also continued exploring even after the damping fitness criteria was satisfied. The reason for this modification is that passing Gruau's generalization test does not mean an optimal damping controller has been found, yet the generalization test used to determine whether the task has been solved requires an optimal damping controller. Third, Igel found that the bias unit in the Markovian version of the task actually makes it harder, so they removed the bias. All of these modifications are useful in pole balancing, although they may not be useful in general. They are also modifications that can be applied to the other learning algorithms, probably improving their performance as well. Therefore Igel's (2003) results are not comparable with the earlier ones, but require a separate comparison.

In the double pole balancing task with velocity information provided, Igel reported that the CMA-ES solves the task in between 884 and 25,853 evaluations, depending on the particular fixed topology and CMA-ES parameter settings. When NEAT is evolved on this same task with the bias removed and a population of 16, it can solve it in 642 evaluations (averaged over 50 runs). This result is over five times faster than the 3,600 evaluations taken by NEAT when evolving with a population of 150, suggesting that NEAT obtains a similar advantage from evolving small populations as the CMA-ES. Moreover, NEAT solves the task in fewer evaluations than any of the CMA-ES topologies, showing again that the ability to efficiently evolve network topologies along with weights provides an advantage.

NEAT's results also improve when the population size is reduced to 16 in the non-Markovian version of the task. In that case, NEAT finds a solution to the generalization test in an average of 6,929 evaluations over 10 runs. CMA-ES solves this version of the task in between 6,061 and 25,254 evaluations depending on the fixed topology and system parameters. The result again demonstrates that determining the appropriate topology automatically is a significant advantage.

Once the right topology is found, CMA-ES performance rivals that of NEAT. CMA-ES works by keeping track of correlations between mutations on specific connection weights and changes in fitness. Interestingly, tracking correlations among parameters is possible in NEAT just as it is in the CMA-ES since NEAT keeps track of which connection is which among different networks using historical marking (Section 3.2). Thus, the strengths of CMA-ES and NEAT may prove complementary and therefore may be possible to combine in the future.

4.2.5 Pole Balancing Conclusion

In sum, the pole balancing results demonstrate both that NEAT can evolve structure when necessary, and that NEAT gains a significant performance advantage from doing so. NEAT significantly outperforms traditional RL methods, and it is at least as powerful as the other NE methods without the need for choosing an appropriate starting topology. The chapter now turns to understanding how the system works, and whether it indeed solves the problems with evolving a population of diverse topologies raised in Chapter 2.

4.3 Experimental Analysis of NEAT

Chapter 3 argued that NEAT's strong performance is due to historical markings, speciation, and incremental growth from minimal structure. In order to verify the contribution of each component, a series of ablation studies were performed. In addition, a species visualization technique was used to depict the dynamics of the system.

4.3.1 Ablations Method

Ablations are meant to establish that each component of NEAT is necessary for its performance. For example, it is possible that growth from minimal structure is not really important; maybe the rest of the system, speciation and historical markings, is sufficient for NEAT's optimal performance. This hypothesis will be checked by ablating both growth and starting from minimal structure from the system. On the other hand, perhaps the situation is the opposite, and speciation buys nothing: protecting innovation might not be as important as I have argued. This hypothesis will be checked by ablating speciation from the system. Finally, NEAT is designed to be able to make use of crossover even though genomes in NEAT have different sizes. This point is more controversial than it might seem. For example, Angeline *et al.* (1993) claimed that crossover in TWEANNs does more harm than good. This hypothesis will be checked by ablating crossover from the system.

The reason why historical markings are not ablated directly is that without historical markings NEAT would reduce to a conventional NE system. Historical markings are the basis of every function in NEAT: Speciation uses a compatibility operator that is based on historical markings, and crossover would not be possible without them. All other system components can be ablated systematically.

Ablations can have a significant detrimental effect on performance, potentially to the point where the system cannot solve the task at all. Therefore, double pole balancing *with* velocities was used as the task for ablation studies with the normal population size of 150. The task is complex enough to be interesting, yet still not too hard, so that ablated systems work as well. Thus, it is possible to compare the ablated versions of the system to the unablated system.

All settings were the same as in the double pole balancing with velocities experiment. Results are averages over 20 runs, except nonmating and full NEAT, which are averages over 120 runs (nonmating NEAT was fast enough to allow many runs).

4.3.2 Ablations Results

Table 4.3 shows the results of all the ablations, in terms of average evaluations required to find a solution. If a solution could not be found in $1,000$ generations, evolution was restarted. Averages include trials that failed to find a solution. A failure rate denotes how often such failures occurred for each ablation. The main result is that the system performs significantly worse ($p \leq 0.001$) for every ablation. The next sections explain how each ablation was performed.

Method	Evaluations	Failure Rate
No-Growth NEAT (Fixed-Topologies)	**630,239**	80%
Non-speciated NEAT	**75,600**	25%
Initial Random NEAT	**30,927**	5%
Nonmating NEAT	**5,557**	0
Full NEAT	**3,600**	0

Table 4.3: **NEAT ablations summary.** The table compares the average number of evaluations (including restarts) for a solution in the double pole balancing with velocities task. Each ablation leads to a weaker algorithm, showing that each component is necessary.

No-Growth Ablation

Since populations in NEAT start out with no hidden nodes, simply removing growth from the system would disable NEAT by barring hidden nodes from all networks. NE systems where structures are fixed start with a fully-connected hidden layer of neurons (Wieland 1991). Therefore, to make the experiment fair, the no-growth ablation was also allowed to start with a fully-connected hidden layer. Every genome specified 10 hidden units like the fixed topology methods in this task (Saravanan and Fogel 1995; Wieland 1991). Without growth, the system was only able to use weight differences to speciate the population. Given 1,000 generations to find a solution, the ablated system could only find a solution 20% of the time! Including restarts, it takes 175 times more evaluations than full NEAT to find a solution! Clearly, speciation and historical markings alone do not account for full NEAT's performance.

Initial Random Ablation

TWEANNs other than NEAT typically start with a random population (Angeline et al. 1993; Gruau et al. 1996; Yao 1999). The structures in these systems can still grow (in most cases, up to some bound). It is still possible that although growth is necessary, starting minimally is not.

This question was examined by starting evolution with random topologies as in other TWEANNs. Each network in the initial population received between 1 and 10 hidden neurons with random connectivity (as implemented by Pujol and Poli 1998). The result is that random-starting NEAT was 8.5 times slower than full NEAT on average, including restarts. The random-starting system also failed to find a solution within 1000 generations 5% of the time. This result suggests that starting randomly forces NE to search higher-dimensional spaces than necessary, thereby wasting time. If topologies are to grow, they should start out as small as possible.

Non-Speciated Ablation

Speciation is intended to protect innovation and allows search to proceed in many different spaces simultaneously. To test this function, speciation should be ablated from the system. However, if nothing else is changed in the system, no structural innovations can survive, causing all networks to be stuck in minimal form.

To make the speciation ablation more meaningful, the non-speciated NEAT must be started with an initial random population. This way, a variety of structures exist in the population from the beginning, bypassing the need for speciation to evolve a structurally diverse population. The resulting non-speciated NEAT was able to find solutions, although it failed in 25% of the attempts. It took 21 times longer on average than full NEAT.

The reason for the dramatic slowdown is that without speciation, the population quickly converges to whatever topology happens to initially perform best. Thus, a lot of diversity is drained immediately (within 10 generations). On average, this initially best-performing topology has about 5 hidden nodes. Thus, the population tends to converge to a relatively high-dimensional search space, even though the smaller networks in the initial population would have optimized faster. The smaller networks do not get a chance because being small offers no immediate advantage in the initially random weight space. Of course, once in a while small networks are found that perform well, allowing a solution to be found more quickly. Whether or not such networks are found early on accounts for the large standard deviation of 41,704 evaluations.

The result shows that growth without speciation is not sufficient to account for NEAT's performance. For growth to succeed, it requires speciation, because speciation gives different structures a chance to optimize in their own niches.

Nonmating NEAT

For the last ablation, mating was removed from NEAT. This ablation tests the claim that crossover is a useful technique in TWEANNs. If NEAT's method of crossover works, then NEAT should perform significantly better with mating and mutation than with mutation alone.

A total of 120 simulations were run with crossover disabled but all other settings the same as before. It took on average 5,557 evaluations to find a solution without mating, compared to 3,600 with mating enabled. The difference is statistically significant ($p = 0.001$). Thus, it is clear that mating *does* contribute when it is done right. However, the nonmating version of NEAT is still significantly faster than the other ablations.

4.3.3 Ablation Conclusions

An important conclusion is that all of the parts of NEAT work together (figure 4.2). None of the system can work without historical markings because all of NEAT's functions utilize historical markings. If growth from minimal structure is removed, speciation can no longer help NEAT find

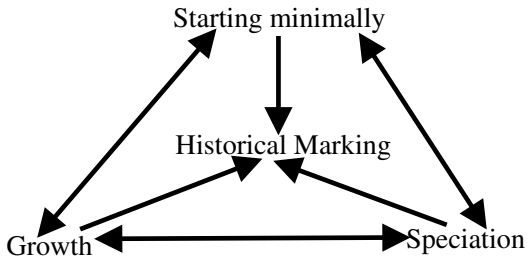


Figure 4.2: **Dependencies among NEAT components.** Strong interdependencies can be identified among the different components of NEAT. An arrow pointing from component A to B indicates that A is dependent on B. No two components can maintain the performance of NEAT when one component is removed. All components of NEAT require historical markings in order to work. The conclusion is that NEAT is a cohesive system that is more than the sum of its parts.

spaces with minimal dimensionality. If speciation is removed, growth from minimal structures cannot proceed because structural innovations do not survive. When the system starts with a population of random topologies without speciation, the system quickly converges onto a non-minimal topology that just happens to be one of the best networks in the initial population. Thus, each component is necessary to make NEAT work.

4.3.4 Visualizing Speciation

The ablation studies demonstrate that speciation is a necessary part of the overall system. To understand how innovation takes place in NEAT, it is important to understand the dynamics of speciation. How many species form over the course of a run? How often do new species arise? How often do species die? How large do the species get? These questions can be answered by depicting speciation visually over time.

Figure 4.3 depicts a typical run of the double pole balancing with velocities task. In this run, the task took 29 generations to complete, which is slightly above average. In the visualization, successive generations are shown from top to bottom. Species are depicted horizontally for each generation, with the width of each species proportional to its size during the corresponding generation. Species are divided from each other by white lines, and new species always arrive on the right hand side. A species becomes red when the fitness of its most fit member is one standard deviation above the mean fitness of the run, indicating that the species is a highly promising one. A species becomes yellow when it is two standard deviations above the mean, suggesting that the species is very close to a solution. Thus, it is possible to follow any species from its inception to the end of the run.

Figure 4.3 shows that the initial minimal topology species was the only species in the population until the 5th generation. Recall that species are computed by the system according to a compatibility distance metric, indicating that before generation 7, all organisms were sufficiently

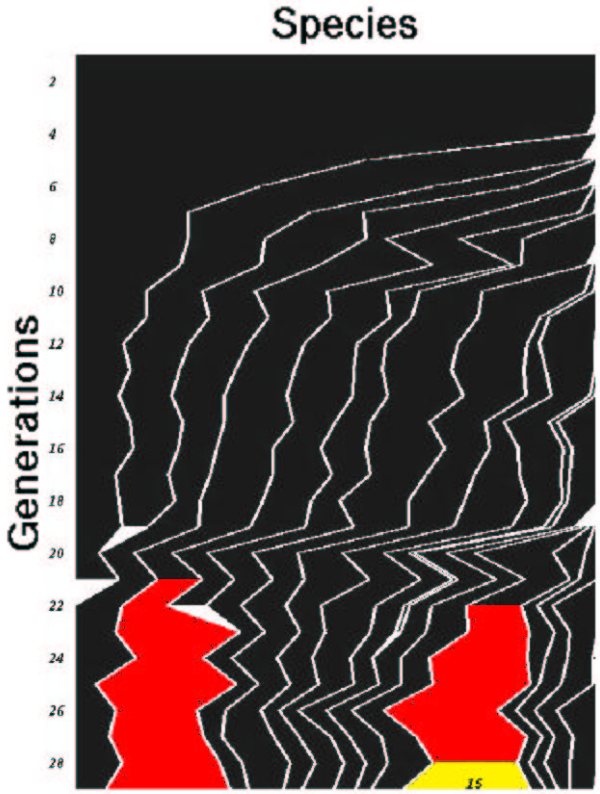


Figure 4.3: **Visualizing speciation during a run of the double pole balancing with velocity information task.** The visualization shows how species grow and shrink over generations. Each species is divided from the others by white lines. A horizontal slice through the figure shows how big each species was during a particular generation. A species is colored red when its maximum fitness is one standard deviation above the mean fitness of the run, and yellow when it is two standard deviations above mean, i.e. when it is about to find the solution. Two species begin to close in on a solution soon after the 20th generation. Around the same time, some of the oldest species become extinct (represented by white triangles). The figure shows how fitness sharing keeps the sizes of different species relatively stable and how no one species is able to take over the population even after significant gains in fitness.

compatible to be grouped into a single species. The visualization shows how the initial species shrinks dramatically in order to make room for the new species.

A few species can be observed becoming extinct during this run. When a species becomes extinct, a white triangle appears between the generation it expired and the next generation. Thus, in this run, the initial species finally became extinct at the 21st generation after shrinking for a long time. It was unable to compete with newer, more innovative species. The second species to appear in the population met a similar fate in the 19th generation.

In the 21st generation a structural mutation in the second-oldest surviving species connected

the long pole angle sensor to a hidden node that had previously only been connected to the cart position sensor. This gave networks in the species the new capability to combine these observations, leading to a significant boost in fitness (and reddening of the species in figure 4.3). The innovative species subsequently expanded, but did not take over the population. Nearly simultaneously, in the 22nd generation, a younger species also made its own useful connection, this time between the short pole velocity sensor and long pole angle sensor, leading to its own subsequent expansion. In the 28th generation, this same species made a pivotal connection between the cart position and its already established method for comparing short pole velocity to long pole angle. This innovation was enough to solve the problem within one generation of additional weight mutations. In the final generation, the winning species was 11 generations old and included 38 neural networks out of the population of 150.

Most of the species that did not come close to a solution survived the run even though they fell significantly behind around the 21st generation. This observation is important, because it visually demonstrates that innovation is indeed being protected. The winning species does not take over the entire population. Maintaining a variety of solutions is also useful in applications where the desired optimal behavior changes over evolution. Chapter 9 describes a real-time game based on NEAT where such changes frequently occur.

Ablation studies confirm that NEAT's components depend strongly on each other, and the speciation visualization is useful in understanding the dynamics of the system. Together, they show that the system works together as a cohesive whole with speciation dynamics progressing as intended.

4.4 Conclusion

This chapter began by asking two questions. First, can NEAT evolve the necessary structure? Results on XOR show that NEAT indeed does so consistently and reliably. The second question was can NEAT find solutions more efficiently than other RL systems? It turns out that all the advanced NE methods, NEAT, ESP, and CMA-ES, significantly outperform the traditional RL methods. However, of these only NEAT can efficiently find the appropriate topology for the task. This is a significant advantage in applying NE to new tasks, and makes it a leading method for reinforcement learning problems in the real world. Ablation studies further confirm that each component of NEAT is necessary for the whole system to work.

The main promise of NEAT, aside from its efficiency, is that through complexification it can find solutions that would otherwise be unreachable with other methods. Having established that NEAT indeed performs well on a major benchmark task, the next chapter shows how NEAT utilizes complexification to maintain continuing innovation in an evolutionary arms race.

Chapter 5

Coevolutionary Complexification

While the previous chapter demonstrated that a complexifying system is efficient, this chapter focuses on open-ended problems that have no explicit fitness function; instead, fitness depends on comparisons with other agents that are also evolving. The goal is to discover creative solutions beyond a designer's ability to define a fitness function. This goal is important because there is no independent fitness measure or benchmark in many domains, such as the game of Go or competitive military scenarios: Fitness is only defined in terms of other players. In such *coevolutionary* domains it is difficult to continually improve solutions because evolution tends to oscillate between idiosyncratic yet uninteresting solutions (Floreano and Nolfi 1997). However, complexification encourages continuing innovation by elaborating on existing solutions (Section 2.7). Thus, NEAT is specifically designed to encourage continual innovation.

In order to demonstrate the power of complexification in coevolution, NEAT is applied to the competitive robot control domain of *Robot Duel*. This domain combines predator/prey interaction and food foraging in a novel head-to-head competition. There is no known optimal strategy in the domain but there is substantial room to come up with increasingly sophisticated strategies. The main results were that (1) evolution did complexify when possible, (2) complexification led to elaboration, and (3) significantly more sophisticated and successful strategies were evolved with complexification than without it. These results imply that complexification allows establishing a coevolutionary arms race that achieves a significantly higher level of sophistication than is otherwise possible.

5.1 The Robot Duel Domain

A domain is needed to test the hypothesis that the complexification process allows discovering more sophisticated strategies, i.e. strategies that are more effective, flexible, and general, and include more components and variations than do strategies obtained through search in a fixed space. To demonstrate this effect, we need a domain where it is possible to develop a wide range increasingly

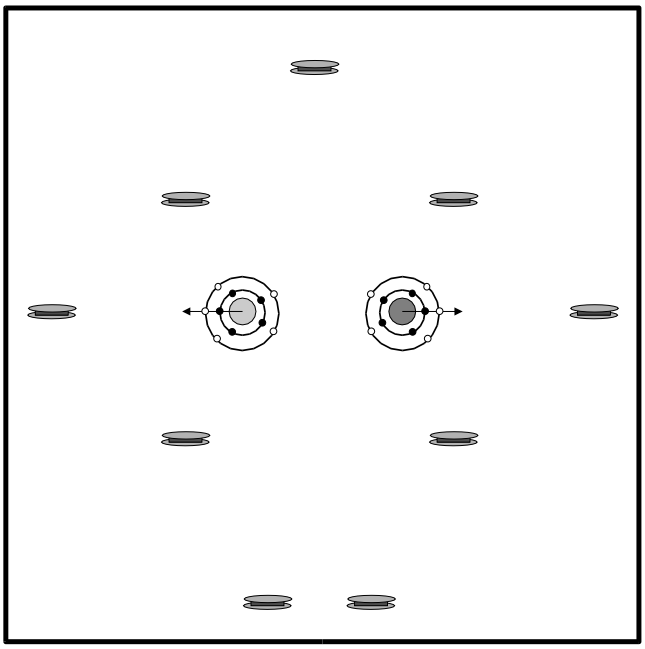
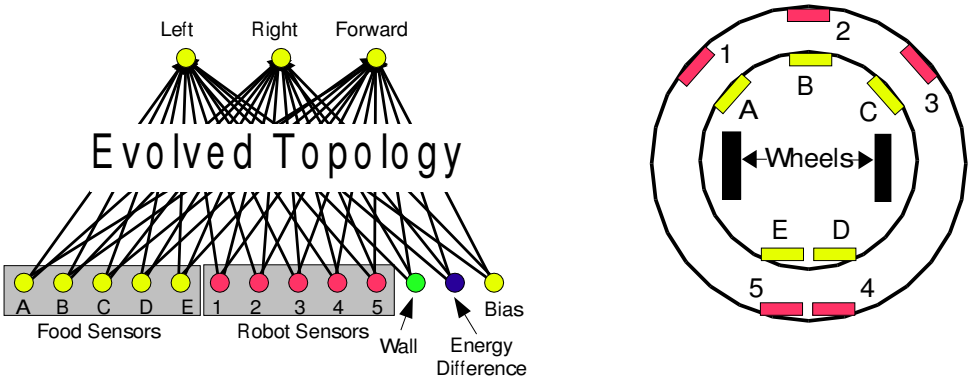


Figure 5.1: **The Robot Duel Domain.** The robots begin on opposite sides of the board facing away from each other as shown by the arrows pointing away from their centers. The concentric circles around each robot represent the separate rings of opponent sensors and food sensors available to each robot. Each ring contains five sensors. The robots lose energy when they move around, and gain energy by consuming food (shown as small sandwiches). The objective is to attain a higher level of energy than the opponent, and then collide with it. Because of the complex interaction between foraging, pursuit, and evasion behaviors, the domain allows for a broad range of strategies of varying sophistication. Animated demos of such evolved strategies are available at **nn.cs.utexas.edu/pages/research/neatdemo.html**.

sophisticated strategies, and where sophistication can be readily measured. A coevolution domain is particularly appropriate because a sustained arms race should lead to increasing sophistication.

In choosing the domain, it is difficult to strike a balance between being able to evolve complex strategies and being able to analyze and understand them. Pursuit and evasion tasks have been utilized for this purpose in the past (Gomez and Miikkulainen 1997; Miller and Cliff 1994; Reggia et al. 2001; Jim and Giles 2000; Sims 1994), and can serve as a benchmark domain for complexifying coevolution as well. While past experiments evolved either a predator or a prey, an interesting coevolution task can be established if the agents are instead equal and engaged in a duel. To win, an agent must develop a strategy that outwits that of its opponent, utilizing structure in the environment.

This chapter introduces such a duel domain, in which two simulated robots try to overpower each other (figure 5.1). The two robots begin on opposite sides of a rectangular room facing away from each other. As the robots move, they lose energy in proportion to the amount of force they apply to their wheels. Although the robots never run out of energy (they are given enough to survive the entire competition), the robot with higher energy wins when it collides with its competitor. In



(a) Neural Network Controller (b) Robot Sensors and Wheels

Figure 5.2: **Robot neural networks (*color figure*).** Five food sensors and five robot sensors detect objects around the robot. A single wall sensor indicates proximity to walls, and the energy difference sensor tells the robot how its energy level differs from that of its opponent. This difference is important because the robot with lower energy loses if the robots collide. The three motor outputs are mapped to forces that control the left and the right wheel.

addition, each robot has a sensor indicating the difference in energy between itself and the other robot. To keep their energies high, the robots can consume food items, which are always placed in a horizontally symmetrical pattern around the middle of the board.

The robot duel task supports a broad range of sophisticated strategies that are easy to observe and interpret. The competitors must become proficient at foraging, prey capture, and escaping predators. In addition, they must be able to quickly switch from one behavior to another. The task is well-suited to competitive coevolution because naive strategies such as forage-then-attack can be complexified into more sophisticated strategies such as luring the opponent to waste its energy before attacking.

The simulated robots are similar to Kheperas (Mondada et al. 1993; figure 5.2). Each has two wheels controlled by separate motors. Five rangefinder sensors can sense food and another five can sense the other robot. Finally, each robot has an energy-difference sensor, and a single wall sensor.

The robots are controlled with neural networks evolved with NEAT. The networks receive all of the robot sensors as inputs, as well as a constant bias that NEAT can use to change the activation thresholds of neurons. They produce three motor outputs: two to encode rotation either right or left, and a third to indicate forward motion power. These three values are then translated into forces to be applied to the left and right wheels of the robot.

The state s_t of the world at time t is defined by the positions of the robots and food, the energy levels of the robots, and the internal states (i.e. neural activation) of the robots' neural networks, including sensors, outputs, and hidden nodes. The subsequent state s_{t+1} is determined by

the outputs of the robots' neural network controllers, computed from the inputs (i.e. sensor values) in s_t in one step of propagation through the network. The robots change their position in s_{t+1} according to their neural network outputs as follows. The change in direction of motion is proportional to the difference between the left and right motor outputs. The robot drives forward a distance proportional to the forward output on a continuous board of size 600 by 600. Robots and food both span a diameter of approximately 15. The robot first makes half its turn, then moves forward, then completes the second half of its turn, so that the turning and forward motions are effectively combined. If the robot encounters food within a distance of 20, it receives an energy boost, and the food disappears from the world. The loss of energy due to movement is computed as the sum of the turn angle and the forward motion, so that even turning in place takes energy. If the robots are within a distance of 20, a collision occurs and the robot with a higher energy wins (see appendix A.4.2 for the exact parameter values used).

Since the observed state o_t taken by the sensors does not include the internal state of the opponent robot, the robot duel is a partially-observable Markov decision process (POMDP). Since the next observed state o_{t+1} depends on the decision of the opponent, it is necessary for robots to learn to predict what the opponent is likely to do, based on their past behavior, and also based on what reasonable behavior is in general. For example, it is reasonable to assume that if the opponent robot is quickly approaching and has higher energy, it is probably trying to collide. Because an important and complex portion of s is not observable, memory, and hence recurrent connections, are crucial for success.

This complex robot-control domain allows competitive coevolution to evolve increasingly sophisticated and complex strategies, and can be used to demonstrate and understand complexification, as will be described next.

5.2 Experiments

In order to demonstrate how complexification enhances performance, thirty-three 500-generation runs of coevolution were run in the robot duel domain. Thirteen of these runs were based on the full NEAT method. The remaining 20 runs evolved either fixed topology networks with several different topologies or simplifying networks that start out complex and lose structure over time. In these remaining 20 runs, complexification was turned off (although networks were still speciated based on weight differences), in order to see how complexification contributes evolving sophisticated strategies. The competitive coevolution setup is described first, followed by an overview of the dominance tournament method for monitoring progress.

5.2.1 Competitive Coevolution Setup

The robot duel domain supports highly sophisticated strategies. Thus, the question in such a domain is whether *continual coevolution* will take place, i.e. whether increasingly sophisticated strategies

will appear over the course of evolution. The experiment has to be set up carefully for this process to emerge, and to be able to identify it when it does.

In competitive coevolution, every network should play a sufficient number of games to establish a good measure of fitness. To encourage interesting and sophisticated strategies, networks should play a diverse and high quality sample of possible opponents. One way to accomplish this goal is to evolve two separate populations. In each generation, each population is evaluated against an intelligently chosen sample of networks from the other population. The population currently being evaluated is called the *host* population, and the population from which opponents are chosen is called the *parasite* population (Rosin and Belew 1997). The parasites are chosen for their quality and diversity, making host/parasite evolution more efficient and more reliable than one based on random or round robin tournament.

In the experiment, a single fitness evaluation included two competitions, one for the east and one for the west starting position. That way, networks needed to implement general strategies for winning, independent of their starting positions. Host networks received a single fitness point for each win, and no points for losing. If a competition lasted 750 time steps with no winner, the host received zero points.

In selecting the parasites for fitness evaluation, good use can be made of the speciation and fitness sharing that already occur in NEAT. Each host was evaluated against the four highest species' champions. They are good opponents because they are the best of the best species, and they are guaranteed to be diverse because their distance must exceed the threshold δ_t (Section 3.3). Another eight opponents were chosen randomly from a Hall of Fame composed of all generation champions (Rosin and Belew 1997). The Hall of Fame ensures that existing abilities need to be maintained to obtain a high fitness. Each network was evaluated in 24 games (i.e. 12 opponents, 2 games each), which was found to be effective experimentally. Together speciation, fitness sharing, and Hall of Fame comprise an effective competitive coevolution methodology. Complete experimental parameters are given in appendix A.

However, it should be noted that complexification does not depend on the particular coevolution methodology. For example Pareto coevolution (De Jong 2004; Ficici and Pollack 2001; Noble and Watson 2001) could have been used as well, and the advantages of complexification would be the same. However, Pareto coevolution requires every member of one population to play every member of the other, and the running time in this domain would have been prohibitively long. In order to interpret experimental results, a method is needed for analyzing progress in competitive coevolution efficiently. The next section describes such a method.

5.2.2 Monitoring Progress in Competitive Coevolution

A competitive coevolution run returns a record of every generation champion from both populations. The question is, how can a sequence of increasingly sophisticated strategies be identified in this data, if one exists? This section describes the *dominance tournament* method for monitoring progress in

competitive coevolution (Stanley and Miikkulainen 2002a) that allows us to do that.

First a method is needed for performing individual comparisons, i.e. whether one strategy is better than another. Because the board configurations can vary between games, champion networks were compared on 144 different food configurations from each side of the board, giving 288 total games for each comparison. The food configurations included the same 9 symmetrical food positions used during training, plus an additional 2 food items, which were placed in one of 12 different positions on the east and west halves of the board. Some starting food positions give an initial advantage to one robot or another, depending on how close they are to the robots' starting positions. Thus, the one who wins the majority of the 288 total games has demonstrated its superiority in many different scenarios, including those beginning with a disadvantage. We say that network a *is superior* to network b if a wins more games than b out of the 288 total games.

Given this definition of superiority, progress can be tracked. The obvious way to do it is to compare each network to others throughout evolution, finding out whether later strategies can beat more opponents than earlier strategies. For example, Floreano and Nolfi (1997) used a measure called *master tournament*, in which the champion of each generation is compared to all other generation champions. Unfortunately, such methods are impractical in a time-intensive domain such as the robot duel competition. Moreover, the master tournament only counts how many strategies can be defeated by each generation champion, without identifying which ones. Thus, it can fail to detect cases where strategies that defeat fewer previous champions are actually superior in a direct comparison. For example, if strategy A defeats 499 out of 500 opponents, and B defeats 498, the master tournament will designate A as superior to B *even if* B defeats A in a direct comparison. In order to decisively track strategic innovation, we need to identify *dominant strategies*, i.e. those that defeat *all previous* dominant strategies. This way, we can make sure that evolution proceeds by developing a progression of strictly more powerful strategies, instead of e.g. switching between alternative ones.

The *dominance tournament* method of tracking progress in competitive coevolution meets this goal (Stanley and Miikkulainen 2002a). Let a *generation champion* be the winner of a 288 game comparison between the host and parasite champions of a single generation. Let d_j be the jth dominant strategy to appear over evolution. Dominance is defined recursively starting from the first generation and progressing to the last:

- The first dominant strategy d_1 is the generation champion of the first generation;
- dominant strategy d_j, where $j > 1$, is a generation champion such that for all $i < j$, d_j is superior to d_i (i.e. wins the 288 game comparison with it).

This strict definition of dominance prohibits circularities. For example, d_4 must be superior to strategies d_1 through d_3, d_3 superior to both d_1 and d_2, and d_2 superior to d_1. We call d_n the nth dominant strategy of the run. If a network c exists that, for example, defeats d_4 but loses to d_3,

making the superiority circular, it would not satisfy the second condition and would not be entered into the dominance hierarchy.

The entire process of deriving a dominance hierarchy from a population is a *dominance tournament*, where competitors play all previous dominant strategies until they either lose a 288 game comparison, or win every comparison to previous dominant strategies, thereby becoming a new dominant strategy. Dominance tournament allows us to identify a sequence of increasingly more sophisticated strategies. It also requires significantly fewer comparisons than the master tournament (Stanley and Miikkulainen 2002a).

Armed with the appropriate coevolution methodology and a measure of success, we can now ask the question: Does the complexification result in more successful competitive coevolution?

5.3 Results

Each of the 33 evolution runs took between 5 and 10 days on a 1GHz Pentium III processor, depending on the progress of evolution and sizes of the networks involved. The NEAT algorithm itself used less than 1% of this computation: Most of the time was spent in evaluating networks in the robot duel task. Evolution of fully-connected topologies took about 90% longer than structure-growing NEAT because larger networks take longer to evaluate.

In order to analyze the results, we define *complexity* as the number of nodes and connections in a network: The more nodes and connections there are, the more complex behavior it can potentially implement. The results were analyzed to answer three questions: (1) As evolution progresses does it also continually complexify? (2) Does such complexification lead to more sophisticated strategies? (3) Does complexification allow better strategies to be discovered than does evolving fixed-topology networks? Each question is answered in turn below.

5.4 Evolution of Complexity

NEAT was run thirteen times, each time from a different seed, to verify that the results were consistent. The highest levels of dominance achieved were 17, 14, 17, 16, 16, 18, 19, 15, 17, 12, 12, 11, and 13, averaging at 15.15 (sd = 2.54).

At each generation where the dominance level increased in at least one of the thirteen runs, the number of connections and number of nodes was averaged in the current dominant strategy across all runs (figure 5.3). Thus, the graphs represent a total of 197 dominance transitions spread over 500 generations. The rise in complexity is dramatic, with the average number of connections tripling and the average number of hidden nodes rising from 0 to almost six. In a smooth trend over the first 200 generations, the number of connections in the dominant strategy grows by 50%. During this early period, dominance transitions occur frequently (fewer prior strategies need to be beaten

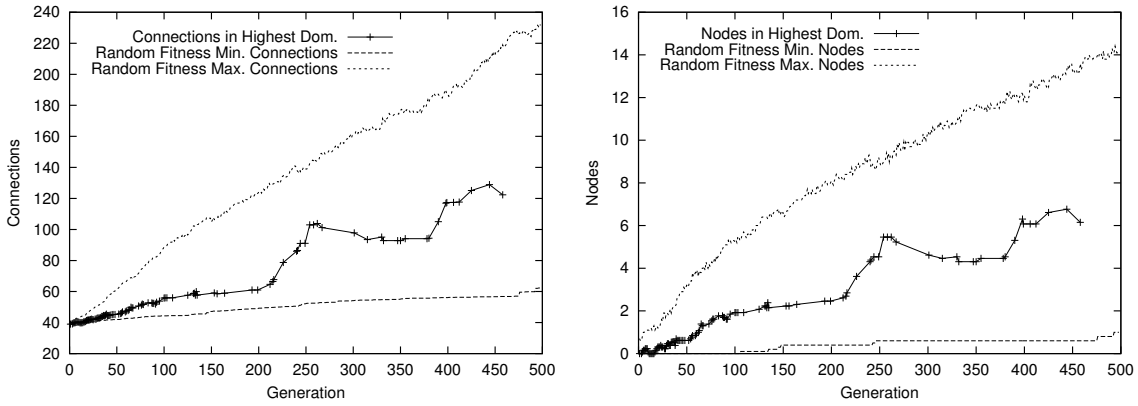


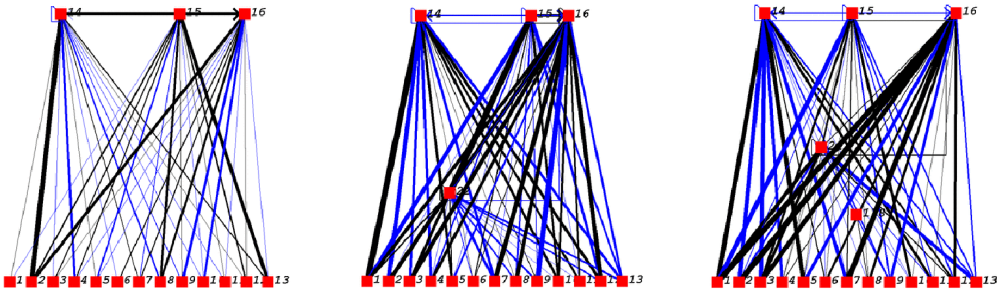
Figure 5.3: **Complexification of connections and nodes over generations.** The hashed lines depict the average number of connections and the average number of hidden nodes in the highest dominant network in each generation. Averages are taken over 13 complexifying runs. A hash mark appears every generation in which a new dominant strategy emerged in at least one of the 13 runs. The graphs show that as dominance increases, so does complexity. The differences between the average final and first dominant strategies are statistically significant for both connections and nodes ($p < 0.001$). For comparison the dashed lines depict the sizes of the average smallest and largest networks in the entire population over five runs where the fitness is assigned randomly. These bounds show that the increase in complexity is not inevitable; both very simple and very complex species exist in the population throughout the run. When the dominant networks complexify, they do so because it is beneficial.

to achieve dominance). Over the next 300 generations, dominance transitions become more sparse, although they continue to occur.

Between the 200th and 500th generations a stepped pattern emerges: The complexity first rises dramatically, then settles, then abruptly increases again (This pattern is even more marked in individual complexifying runs; the averaging done in Figure 5.3 smooths it out somewhat). The cause for this pattern is speciation. While one species is adding a large amount of structure, other species are optimizing the weights of less complex networks. Initially, added complexity leads to better performance, but subsequent optimization takes longer in the new higher-dimensional space. Meanwhile, species with smaller topologies have a chance to temporarily catch up through optimizing their weights. Ultimately, however, more complex structures eventually win, since higher complexity is necessary for continued innovation.

Thus, there are two underlying forces of progress: The building of new structures, and the continual optimization of prior structures in the background. The product of these two trends is a gradual stepped progression towards increasing complexity.

An important question is: Because NEAT searches by adding structure only, not by removing it, does the complexity always increase whether it helps in finding good solutions or not? To demonstrate that NEAT indeed prefers simple solutions and complexifies only when it is useful, we ran five complexifying evolution runs with fitness assigned randomly (i.e. the winner of each game was chosen at random). As expected, NEAT kept a wide range of networks in its population, from



(a) Generation 100 (b) Generation 267 (c) Generation 315

Figure 5.4: **Complexification of a winning species.** The best networks in the same species are shown at landmark generations. Nodes are depicted as squares beside their node numbers, and line thickness represents the strength of connections. Over time, the networks became more complex and gained skills. (a) The champion from generation 10 had no hidden nodes. (b) The addition of h_{22} and its respective connections gave new abilities. (c) The appearance of h_{172} refined existing behaviors.

very simple to highly complex (figure 5.3). That is, the dominant networks did not *have to* become more complex; they only did so because it was beneficial. Not only is the minimum complexity in the random-fitness population much lower than that of the dominant strategies, but the maximum complexity is significantly greater. Thus, evolution complexifies sparingly, only when the complex species holds its own in comparison with the simpler ones.

5.5 Sophistication through Complexification

To see how complexification contributes to evolution, let us observe how a sample dominant strategy develops over time. While many complex networks evolved in the experiments, we will follow the species that produced the winning network d_{17} in the third run because its progress is rather typical and easy to understand. Let us use S_k for the best network in species S at generation k, and h_l for the lth hidden node to arise from a structural mutation over the course of evolution. We will track both strategic and structural innovations in order to see how they correlate. Let us begin with S_{100} (figure 5.4a), when S had a mature zero-hidden-node strategy:

- S_{100}'s main strategy was to follow the opponent, putting it in a position where it might by chance collide with its opponent when its energy is up. However, S_{100} followed the opponent even when the opponent had more energy, leaving S_{100} vulnerable to attack. S_{100} did not clearly switch roles between foraging and chasing the enemy, causing it to miss opportunities to gather food.

- S_{200}. During the next 100 generations, S evolved a *resting* strategy, which it used when it had significantly lower energy than its opponent. In such a situation, the robot stopped moving, while the other robot wasted energy running around. By the time the opponent got close, its

energy was often low enough to be attacked. The resting strategy is an example of improvement that can take place without complexification: It involved increasing the inhibition from the energy difference sensor, thereby slightly modifying intensity of an existing behavior.

- In S_{267} (figure 5.4b), a new hidden node, h_{22}, appeared. This node arrived through an inter-species mating, and had been optimized for several generations already. Node h_{22} gave the robot the ability to change its behavior at once into an all-out attack. Because of this new skill, S_{267} no longer needed to follow the enemy closely at all times, allowing it to collect more food. By implementing this new strategy through a new node, it was possible not to interfere with the already existing resting strategy, so that S now switched roles between resting when at a disadvantage to attacking when high on energy. Thus, the new structure resulted in strategic elaboration.

- In S_{315} (figure 5.4c), another new hidden node, h_{172}, split a link between an input sensor and h_{22}. Replacing a direct connection with a sigmoid function greatly improved S_{315}'s ability to attack at appropriate times, leading to very accurate role switching between attacking and foraging. S_{315} would try to follow the opponent from afar focusing on resting and foraging, and only zoom in for attack when victory was certain. This final structural addition shows how new structure can improve the accuracy and timing of existing behaviors.

The analysis above shows that in some cases, weight optimization alone can produce improved strategies. However, when those strategies need to be extended, adding new structure allows the new behaviors to coexist with old strategies. Also, in some cases it is necessary to add complexity to make the timing or execution of the behavior more accurate. These results show how complexification can be utilized to produce increasing sophistication.

In order to illustrate the level of sophistication achieved in this process, we conclude this section by describing the competition between two sophisticated strategies, S_{210} and S_{313}, from another run of evolution. At the beginning of the competition, S_{210} and S_{313} collected most of the available food until their energy levels were about equal. Two pieces of food remained on the board in locations distant from both S_{210} and S_{313} (figure 5.5). Because of the danger of colliding with similar energy levels, the obvious strategy is to rush for the last two pieces of food. In fact, S_{210} did exactly that, consuming the second-to-last piece, and then heading towards the last one. In contrast, S_{313} forfeited the race for the second-to-last piece, opting to sit still, even though its energy temporarily dropped below S_{210}'s. However, S_{313} was closer to the last piece and got there first. It received a boost of energy while S_{210} wasted its energy running the long distance from the second-to-last piece. Thus, S_{313}'s strategy ensured that it had more energy when they finally met. Robot S_{313}'s behavior was surprisingly deceptive, showing that high strategic sophistication had evolved. Similar waiting behavior was observed against several other opponents, and also evolved in several other runs, suggesting that it is a robust result.

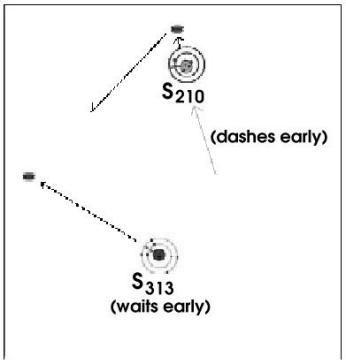


Figure 5.5: **Sophisticated endgame.** Robot S_{313} dashes for the last piece of food while S_{210} is still collecting the second-to-last piece. Although it appeared that S_{313} would lose because S_{210} got the second-to-last piece (gray arrow), it turns out that S_{210} ends with a disadvantage. It has no chance to get to the last piece of food before S_{313}, and S_{313} has been saving energy while S_{210} wasted energy traveling long distances. This way, sophisticated strategies evolve through complexification, combining multiple objectives, and utilizing weaknesses in the opponent's strategy.

This analysis of individual evolved behaviors shows that complexification indeed elaborates on existing strategies, and allows highly sophisticated behaviors to develop that balance multiple goals and utilize weaknesses in the opponent. The last question is whether complexification indeed is necessary to achieve these behaviors.

5.6 Complexification vs. Fixed-topology Evolution and Simplification

Complexifying coevolution was compared to two alternatives: standard coevolution in a fixed search space, and to simplifying coevolution from a complex starting point. Note that it is not possible to compare methods using the standard crossvalidation techniques because no external performance measure exists in this domain. However, the evolved neural networks can be compared *directly* by playing a duel. Thus, for example, a run of fixed-topology coevolution can be compared to a run of complexifying coevolution by playing the highest dominant strategy from the fixed-topology run against the entire dominance ranking of the complexifying run. The highest level strategy in the ranking that the fixed-topology strategy can defeat, normalized by the number of dominance levels, is a measure of its quality against the complexifying coevolution. For example, if a strategy can defeat up to and including the 13th dominant strategy out of 15, then its performance against that run is $\frac{13}{15} = 86.7\%$. By playing every fixed-topology champion, every simplifying coevolution champion, and every complexifying coevolution champion against the dominance ranking from every complexifying run and averaging, the relative performance of each of these methods can be measured.

This section first establishes the baseline performance by playing complexifying coevolution runs against themselves and demonstrating that a comparison with dominance levels is a meaningful

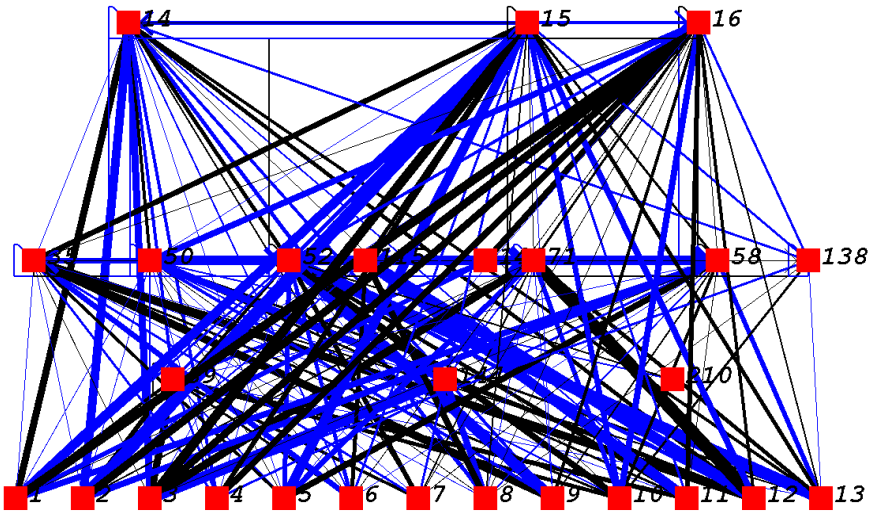


Figure 5.6: **The best complexifying network.** The highest dominant network from the sixth complexifying coevolution run was able to beat 99.6% of the dominance hierarchy of the other 12 runs. The network has 11 hidden units and 202 connections (plotted as in figure 5.4), making significant use of structure. While it still contains the basic direct connections, the strategy they represent has been elaborated by adding several new nodes and connections. For example, lateral and recurrent connections allow taking past events into account, resulting in more refined decisions. While such structures can be found reliably through complexification, it turns out difficult to discover them directly in the high dimensional space through fixed-topology evolution or through simplification.

measure of performance. Complexification is then compared with fixed-topology coevolution of networks of different architectures, including fully-connected small networks, fully-connected large networks, and networks with an optimal structure as determined by complexifying coevolution. Third, the performance of complexification is compared with that of simplifying coevolution.

5.6.1 Complexifying Coevolution

The highest dominant strategy from each of the 13 complexifying runs played the entire dominance ranking from every other run. Their average performance scores were 87.9%, 83.8%, 91.9%, 79.4%, 91.9%, 99.6%, 99.4%, 99.5%, 81.8%, 96.2%, 90.6%, 96.9%, and 89.3%, with an overall average of 91.4% (sd=12.8%). Above all, this result shows that complexifying runs produce consistently good strategies: On average, the highest dominant strategies qualify for the top 10% of the other complexifying runs. The best runs were the sixth, seventh, and eighth, which were able to defeat almost the entire dominance ranking of every other run. The highest dominant network from the best run (99.6%) is shown in Figure 5.6.

In order to understand what it means for a network to be one or more dominance levels above another, Figure 5.7 shows how many more games the more dominant network can be expected to win on average over all its 288-game comparisons than the less dominant network. Even at the

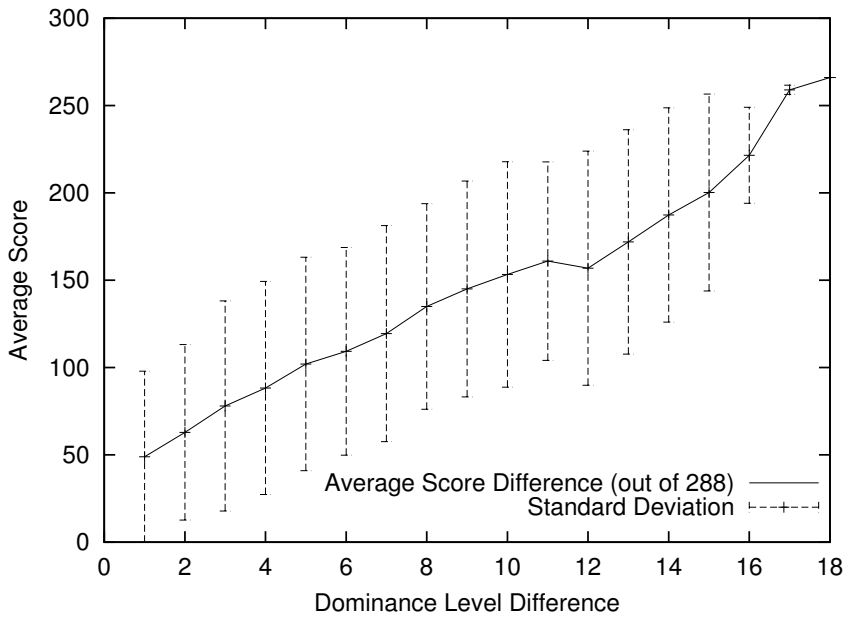


Figure 5.7: **Interpreting differences in dominance levels.** The graph shows how many games in a 288-game comparison a more dominant network can be expected to win, averaged over all runs at all dominance levels of complexifying coevolution. For example, a network one level higher wins 50 more games out of 288. A larger difference in dominance levels translates to a larger difference in performance approximately linearly, suggesting that dominance levels can be used as a measure of performance when an absolute measure is not available.

lowest difference (i.e. that of one dominance level), the more dominant network can be expected to win about 50 more games on average, showing that each difference in dominance level is important. The difference grows approximately linearly: A network five dominance levels higher will win 100 more games, while a network 10 levels higher will win 150 and 15 levels higher will win 200. These results suggest that dominance level comparisons indeed constitute a meaningful way to measure relative performance.

5.6.2 Fixed-Topology Coevolution of Large Networks

In fixed-topology coevolution, the network architecture must be chosen by the experimenter. One sensible approach is to approximate the complexity of the best complexifying network (figure 5.6). This network included 11 hidden units and 202 connections, with both recurrent connections and direct connections from input to output. As an idealization of this structure, a 10-hidden-unit fully recurrent network was used with direct connections from inputs to outputs, with a total of 263 connections. A network of this type should be able to approximate the functionality of the most effective complexifying strategy. Fixed-topology coevolution runs exactly as complexifying coevolution in NEAT, except that no structural mutations occur. In particular, the population is still speciated based

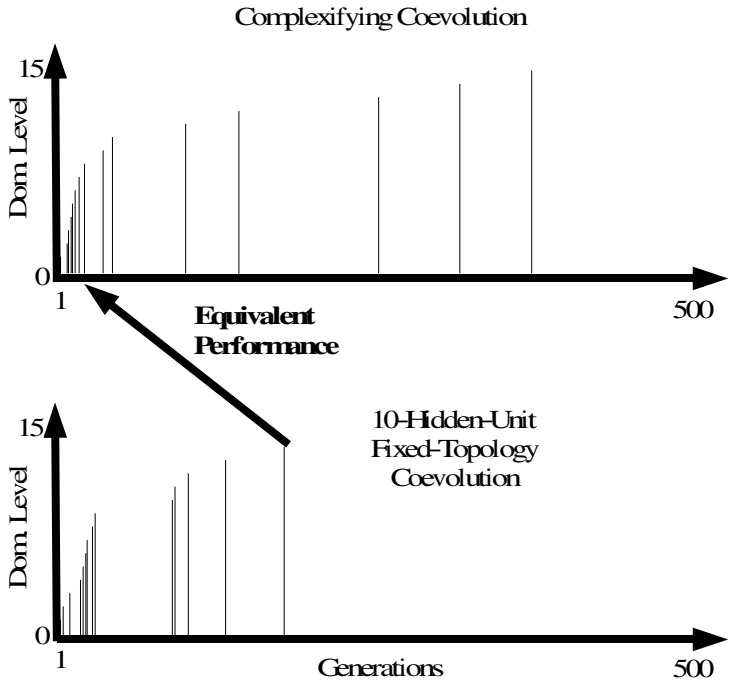


Figure 5.8: **Comparing typical runs of complexifying coevolution and fixed-topology coevolution with ten hidden units.** Dominance levels are charted on the y-axis and generations on the x-axis. A line appears at every generation where a new dominant strategy arose in each run. The height of the line represents the level of dominance. The arrow shows that the highest dominant strategy found in 10-hidden-unit fixed-topology evolution only performs as well as the 8th dominant strategy in the complexifying run, which was found in the 19th generation. (Average = 24, sd = 8.8) In other words, only a few generations of complexifying coevolution are as effective as several hundred of fixed-topology evolution.

on weight differences (i.e. $\overline{W}$ from equation 3.1), using the usual speciation procedure.

Three runs of fixed-topology coevolution were performed with these networks, and their highest dominant strategies were compared to the entire dominance ranking of every complexifying run. Their average performances were 29.1%, 34.4%, and 57.8%, for an overall average of 40.4%. Compared to the 91.4% performance of complexifying coevolution, it is clear that fixed-topology coevolution produced consistently inferior solutions. As a matter of fact, no fixed-topology run could defeat any of the highest dominant strategies from the 13 complexifying coevolution runs.

This difference in performance can be illustrated by computing the *average generation* of complexifying coevolution with the same performance as fixed-topology coevolution. This generation turns out to be 24 (sd = 8.8). In other words, only 24 generations of complexifying coevolution reach on average the same level of dominance as 500 generations of fixed-topology coevolution!

In effect, the progress of the entire fixed-topology coevolution run is compressed into the first few generations of complexifying coevolution (figure 5.8).

5.6.3 Fixed-Topology Coevolution of Small Networks

One of the arguments for using complexifying coevolution is that starting the search directly in the space of the final solution may be intractable. This argument may explain why the attempt to evolve fixed-topology solutions at a high level of complexity failed. Thus, the next experiment aimed at reducing the search space by evolving fully-connected, fully-recurrent networks with a small number of hidden nodes as well as direct connections from inputs to outputs. Considerable experimentation suggested that five hidden nodes (144 connections) was appropriate, allowing fixed-topology evolution to find the best solutions it could. Five hidden nodes is also about the same number of hidden nodes as the highest dominant strategies had on average in the complexifying runs.

A total of seven runs were performed with the 5-hidden-node networks, with average performances of 70.7%, 85.5%, 66.1%, 87.3%, 80.8%, 88.8%, and 83.1%. The overall average was 80.3% (sd=18.4%), which is better but still significantly below the 91.4% performance of complexifying coevolution ($p < 0.001$).

In particular, the two most effective complexifying runs were still never defeated by any of the fixed-topology runs. Also, because each dominance level is more difficult to achieve than the previous one, on average the fixed-topology evolution only reached the performance of the 159th complexifying generation (sd=72.0). Thus, even in the best case, fixed-topology coevolution on average only finds the level of sophistication that complexifying coevolution finds halfway through a run (figure 5.9).

5.6.4 Fixed-Topology Coevolution of Best Complexifying Network

One problem with evolving fully-connected architectures is that they may not have an appropriate topology for this domain. Of course, it is very difficult to guess an appropriate topology *a priori*. However, it is still interesting to ask whether fixed-topology coevolution could succeed in the task assuming that the right topology was known? To answer this question, networks were evolved as in the other fixed-topology experiments, except this time using the topology of the best complexifying network (figure 5.6). This topology may constrain the search space in such a way that finding a sophisticated solution is more likely than with a fully-connected architecture. If so, it is possible that seeding the population with a successful topology gives it an advantage even over complexifying coevolution, which must build the topology from a minimal starting point.

Five runs were performed, obtaining average performance score 86.2%, 83.3%, 88.1%, 74.2%, and 80.3%, and an overall average of 82.4% (sd=15.1%). The 91.4% performance of complexifying coevolution is significantly better than even this version of fixed-topology coevolution ($p < 0.001$). However, interestingly, the 40.4% average performance of 10-hidden-unit fixed topol-

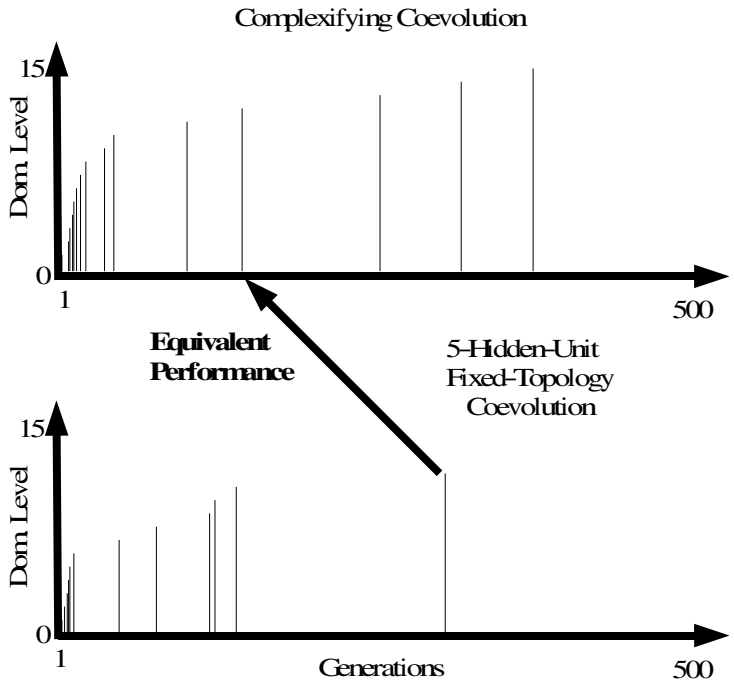


Figure 5.9: **Comparing typical runs of complexifying coevolution and fixed-topology coevolution with five hidden units.** As in figure 5.8, dominance levels are charted on the y-axis and generations on the x-axis, a line appears at every generation where a new dominant strategy arose in each run, and the height of the line represents the level of dominance. The arrow shows that the highest dominant strategy found in the 5-hidden-unit fixed-topology evolution only performs as well as the 12th dominant strategy in the complexifying run, which was found in the 140th generation (Average 159, sd = 72.0). Thus, even in the best configuration, fixed-topology evolution takes about twice as long to achieve the same level of performance.

ogy coevolution is significantly *below* best-topology evolution, even though both methods search in spaces of similar sizes. In fact, best-topology evolution performs at about the same level as 5-hidden-unit fixed-topology evolution (80.3%), even though 5-hidden-unit evolution optimizes half the number of hidden nodes. Thus, the results confirm the hypothesis that using a successful evolved topology does help constrain the search. However, in comparison to complexifying coevolution, the advantage gained from starting this way is still not enough to make up for the penalty of starting search directly in a high-dimensional space. As Figure 5.10 shows, best-topology evolution on average only finds a strategy that performs as well as those found by the 193rd generation of complexifying coevolution.

The results of the fixed-topology coevolution experiments can be summarized as follows: If this method is used to search directly in the high-dimensional space of the most effective solutions,

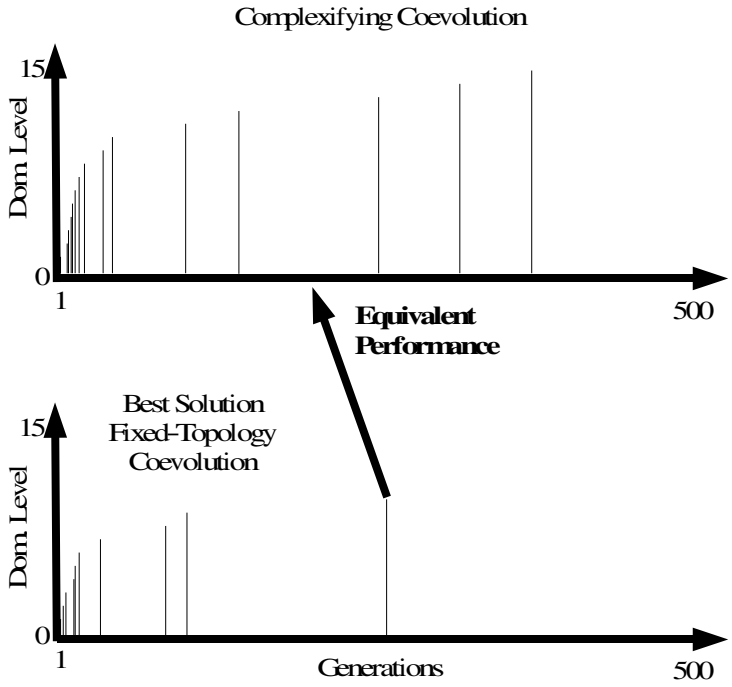


Figure 5.10: **Comparing typical runs of complexifying oevolution and fixed-topology coevolution of the best complexifying network.** Dominance levels are charted as in figure 5.8. The arrow shows that the highest dominant strategy found by evolving the fixed topology of the best complexifying network only performs as well as the dominant strategy that would be found in the 193rd generation of complexifying coevolution (Average 193, sd = 85). Thus, even with an appropriate topology given, fixed-topology evolution takes almost twice as long to achieve the same level of performance.

it reaches only 40% of the performance of complexifying coevolution. It does better if it is allowed to optimize less complex networks; however, the most sophisticated solutions may not exist in that space. Even given a topology appropriate for the task, it does not reach the same level as complexifying coevolution. Thus, fixed-topology coevolution is not competitive with complexifying coevolution with *any* choice of topology.

The conclusion is that complexification is superior not only because it allows discovering the appropriate high-dimensional topology automatically, but also because it makes the optimization of that topology more efficient. This point will be discussed further in Chapter 10.

5.6.5 Simplifying Coevolution

A possible remedy to having to search in high-dimensional spaces is to allow evolution to search for smaller structures by removing structure incrementally. This *simplifying coevolution* is the opposite

of complexifying coevolution. The idea is that a mediocre complex solution can be refined by removing unnecessary dimensions from the search space, thereby accelerating the search.

Although simplifying coevolution is an alternative method to complexifying coevolution for finding topologies, it still requires a complex starting topology to be specified. This topology was chosen with two goals in mind: (1) Simplifying coevolution should start with sufficient complexity to at least potentially find solutions of equal or more complexity than the best solutions from complexifying coevolution, and (2) with a rate of structural removal equivalent to the rate of structural addition in complexifying NEAT, it should be possible to discover solutions significantly simpler than the best complexifying solutions. Thus, a 12-hidden-unit, 339 connection fully-connected fully-recurrent network was chosen to start the search. Since 162 connections were added to the best complexifying network during evolution, a corresponding simplifying coevolution could discover solutions with 177 connections, or 25 less than the best complexifying network.

Simplifying coevolution was run just as complexifying coevolution, except that structural mutations removed connections instead of adding nodes and connections. If all connections of a node were removed, the node itself was removed. Historical markings and speciation worked as in complexifying NEAT, except that all markings were assigned in the beginning of evolution (because structure was only removed and never added). The population was speciated just as in complexifying NEAT.

The five runs of simplifying coevolution performed at 64.8%, 60.9%, 56.6%, 36.4%, and 67.9%, with an overall average of 57.3% (sd=19.8%). Again, such performance is significantly below the 91.4% performance of complexifying coevolution ($p < 0.001$). Interestingly, even though it started with 76 more connections than fixed-topology coevolution with ten hidden units, simplifying coevolution still performed significantly better ($p < 0.001$), suggesting that evolving structure through reducing complexity is better than evolving large fixed structures.

Like Figures 5.8–5.10, Figure 5.11 compares typical runs of complexifying and simplifying coevolution. On average, 500 generations of simplification finds solutions equivalent to 56 generations of complexification. Simplifying coevolution also tends to find more dominance levels than any other method tested. It generated an average of 23.2 dominance levels per run, once even finding 30 in one run, whereas e.g. complexifying coevolution on average finds 15.2 levels. In other words, the difference between dominance levels is much smaller in simplifying coevolution than in complexifying coevolution. Unlike in other methods, dominant strategies tend to appear in spurts of a few at a time, and usually after complexity has been decreasing for several generations, as also shown in Figure 5.11. Over a number of generations, evolution removes several connections until a smaller, more easily optimized space is discovered. Then, a quick succession of minute improvements creates several new levels of dominance, after which the space is further refined, and so on. While such a process makes sense, the inferior results of simplifying coevolution suggest that simplifying search is an ineffective way of discovering useful structures compared to complexification.

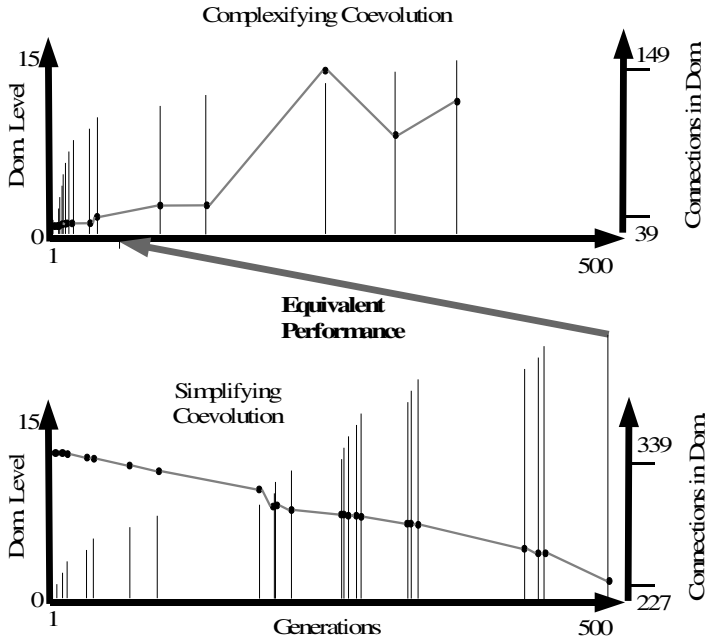


Figure 5.11: **Comparing typical runs of complexifying coevolution and simplifying coevolution.** Dominance levels are charted as in figure 5.8. In addition, the line plot shows the complexity of each dominance level in terms of number of connections in the networks with scale indicated in the y-axis at right. In this typical simplifying run, the number of connections reduced from 339 to 227 connections. The arrow shows that the highest dominant strategy found in simplifying coevolution only performs as well as the 9th or 10th dominant strategy of complexifying coevolution, which is normally found after 56 generations (sd = 31). In other words, even though simplifying coevolution finds more dominance levels, the search for appropriate structure is less effective than that of complexifying coevolution.

5.6.6 Comparison Summary

Table 1 shows how the coevolution methods differ on number of dominance levels, generation of the highest dominance level, overall performance, and equivalent generation. The conclusion is that complexifying coevolution innovates longer and finds a higher level of sophistication than the other methods.

5.7 Conclusion

The experiments in this chapter show that complexification of genomes leads to continual coevolution of increasingly sophisticated strategies. Three trends were found in the experiments: (1) As evolution progresses, solutions become more complex, (2) evolution uses complexification to elab-

Coevolution Type	Ave. Highest Dom. Level	Ave. Highest Generation	Average Performance	Equivalent Generation (out of 500)
Complexifying	15.2	353.6	91.4%	343
Fixed-Topology 10 Hidden Node	12.0	172	40.4%	24
Fixed-Topology 5 Hidden Node	13.0	291.4	80.3%	159
Fixed-Topology Best Network	14.0	301.8	82.4%	193
Simplifying	23.2	444.2	57.3%	56

Table 5.1: **Summary of the performance comparison.** The second column shows how many levels of dominance were achieved in each type of coevolution on average. The third specifies the average generation of the highest dominant strategy, indicating how long innovation generally continues. The fourth column gives the average level in the complexifying coevolution dominance hierarchy that the champion could defeat, and the fifth column shows its average generation. The differences in performance ($p < 0.001$) and equivalent generation ($p < 0.001$) between complexifying coevolution and every other method are significant. The main result is that the level of sophistication reached by complexifying coevolution is significantly higher than that reached by fixed-topology or simplifying coevolution.

orate on existing strategies, and (3) complexifying coevolution is significantly more successful in finding highly sophisticated strategies than non-complexifying coevolution. These results suggest that complexification is a crucial component of a successful search for complex solutions, which is a major goal for AI.

Chapter 6

Evolving Adaptive Neural Networks

Being able to adapt is important because the environment can change during a network's lifetime. For example, robot sensors can fail and objects in the world can appear differently in different lighting conditions. Yet the behavior of the network should remain consistent, i.e. the network should adapt to the changes in the environment. This chapter introduces two ways evolved networks can be made to adapt. First, *adaptation rules* for individual connections can be evolved, potentially allowing networks to change their behavior based on experience. This idea is inspired by synaptic plasticity in biological neurons, which means connection weights change over the network's lifetime. Second, the *state of a recurrent network* with static connection weights can be used to represent the current state of the world, and the network can adapt by changing its state. In this chapter these two methods for adaptation are compared by evolving both types of networks with NEAT. Two important results are that different kinds of adaptation mechanisms are appropriate for different tasks and evolving the topology of the network helps find the best adaptation mechanism.

6.1 Motivation

In many important control problems, the environment may change suddenly or gradually; to maintain sufficient performance, the controller needs to adapt to it online. For example, a robot may lose a sensor or an airplane may lose an engine and there may not be any opportunity to re-evolve controllers for such unexpected circumstances. Adaptation is necessary also to make the controllers general. For example, a controller evolved for driving a car should work for any car, even if the dimensions and mechanics are slightly different from its training models. Even though the same basic control policies are valid, they may need to be adapted slightly for the specific instances.

Natural organisms are constantly faced with unforeseen circumstances and generally adapt to them very well. They can do it because their nervous systems are plastic, i.e. not fixed at birth. Thus, one way to achieve adaptive solutions is to evolve neural networks with plastic synapses, i.e. plastic networks. Evolution can discover rules that determine how connection weights should

change to allow the network to adapt. Another way is to evolve static networks that adapt by changing their state, i.e. the activation levels of their neurons, through recurrent connections. These two approaches are compared in this chapter.

Several researchers have evolved plastic systems in the past. Nolfi and Parisi (1995, 1993) evolved "self-teaching" networks that used the outputs of a teaching subnetwork to train a motor control network using backpropagation. In a slightly different approach they also trained the networks with backpropagation to predict what it would see after moving around in its environment (Nolfi, Elman, and Parisi 1990, 1994). Such learning enhanced performance in a foraging task. Chalmers (1990) evolved a global learning rule (a rule that applies to every connection) and discovered that the evolved rule was similar to backpropagation. McQuesten and Miikkulainen (1997) showed that NE can benefit from parent networks teaching their offspring using backpropagation. These experiments showed that *general* learning mechanisms can improve performance of evolved neural networks.

However, the goal of evolving adaptive neural networks is different from evolving networks that simply utilize learning. The goal is to find *specific* learning mechanisms that are optimized to adapt to new or changed problems that can arise in the domain. In an early example of such an approach, Baxter (1992) evolved a network that could learn Boolean functions of one value. Each connection had a rule for changing its weight to one of two possible values. Baxter's contribution was mainly to show that local learning rules are sufficient to evolve an adaptive network.

More recently, Floreano and Urzelai (2000) showed that evolving local (node-based) synaptic plasticity parameters produces networks that can solve complex problems better than recurrent networks with static synapses. In Floreano and Urzelai's experiment, a plastic network and a static recurrent network were evolved to turn on a light by moving to a switch; the networks then had to move onto a gray square. The neural controllers had a fixed, fully-connected structure. Each connection in the plastic network included a learning rule and a learning rate, and the static network only encoded static connection weights. The sequence of two actions proved difficult to learn for the static network because it could not adapt to the sudden change in goals after the light was switched on. These networks tended to circle around the environment, attracted by both the light switch and the gray square. Plastic networks, on the other hand, completely changed their trajectories after turning on the light, reconfiguring their internal weights to tackle the problem of finding the gray square. This result shows that evolving plastic networks can be an advantage.

However, Blynel and Floreano (2002) showed that a special kind of static recurrent network called a *Continuous Time Recurrent Neural Network* (CTRNNs) can perform Floreano and Urzelai's light-switching task as well as the plastic network. CTRNNs are generalizations of discrete-time recurrent neural networks in which neurons integrate incoming activation at different rates (Section 10.4.2 discusses CTRNNs in detail). The network was able to solve the task by accurately controlling the timing of its actions. This success may have been possible because the task requires a sequence of actions that always take place in the same order with the same timing. That is, the

sequence of behaviors is the same on every trial: First go to the light switch, and then go to the light.

Furthermore, Aharonov-Barki et al. (2001) demonstrated that adapting in a variable environment is possible with static networks as well. They evolved networks to control food foraging agents that had to switch from an exploration mode to a grazing mode depending on whether or not they were in an area of the map called the *food zone*. The best strategy for an agent outside the food zone is to explore until it finds food, and then switch into a grazing pattern. Thus, the behavior of the network has to vary depending on the situation. Aharonov-Barki et al. found that a *command-neuron* evolved that used a self-recurrent connection to switch between and maintain a high or low activation level. The network explored or grazed depending on the state of the command neuron. Thus, the command neuron allowed the network to switch its behavior.

In nature, one of the most significant challenges faced by organisms is environmental change. Weather can change, lighting can change, vegetation can be either dangerous or nutritious depending on location, and animals can walk even when body parts are damaged or wounded. A signficant goal for neural networks is to adapt in tasks where the environment varies. In such tasks the correct policy depends on an aspect of the environment that varies randomly from one trial to the next and is not immediately observable but must be discovered through exploration. In fact, the foraging domain of Aharonov-Barki et al. exhibits such a changing environment depending on the location of the robot. However, the question remains, what is the right approach for adapting in an uncertain environment? As Blynel and Floreano showed, plastic networks also show promise, although their light-switching environment does not vary from trial to trial. Thus, this chapter tests both plastic networks and adaptive networks in a variable environment in order to determine how they can best be applied.

Prior experiments with adaptive networks evolved fully-connected network topologies (Floreano and Urzelai 2000; Aharonov-Barki et al. 2001; Blynel and Floreano 2002). However, network topology is important in both plastic and static recurrent networks: The specific nodes that are connected and the learning rules on those connections determine how the network behaves and adapts, and the recurrent connections determine how memory can be utilized. Thus, NEAT is an appropriate platform for comparing adaptive networks in challenging tasks since it can evolve the appropriate topology for either kind of network.

To determine whether one type of adaptation has a significant advantage over the other, plastic neural networks and static recurrent networks were evolved in a food foraging domain designed to require a policy change during the network's lifetime. For the plastic networks, NEAT was augmented with a facility for evolving local Hebbian learning rules for specific connections in the network. Thus, the connection weights could change over the network's lifetime according to different evolved rules at each connection.

Interestingly, experiments showed that while recurrent networks with static connection weights could solve the task faster and more reliably, plastic solutions were more holistic, i.e. they involved large groups of connections changing their weights together. These results suggest that different

types of adaptive networks are appropriate for different tasks: Static recurrent networks work well in simple domains requiring a single state change, but plastic networks may be better suited towards problems where the entire environment changes (such as light level). The experiments also confirm that NEAT can be used to evolve adaptive networks of both types, which may lead to more biologically-plausible evolution in the future.

The next section explains how NEAT was augmented to evolve learning parameters in addition to connection weights. The remainder of the chapter presents the food foraging domain used in the experiments, and details the results.

6.2 Plastic NEAT

Just as in non-adapting neural networks, it is necessary in a network with plastic synapses to have connections between the *right* nodes, so that the connections can be strengthened or weakened to change the relationship between the computational concepts they represent. In addition, the parameter space being searched in the space of plastic networks is much larger than for static networks: In addition to connection weights, several learning parameters must be evolved for each of several learning rules at each connection. Thus, it is important to minimize the number of connections optimized by evolution. For these reasons, NEAT is an appropriate approach to evolving such networks.

However, NEAT needs to be extended to evolve plastic structures. Specifically, genomes in NEAT must encode the rules that govern changes in connection weights. Every synapse in a plastic network must specify a modification rule that guides how it is adapted. Local learning rules were implemented in NEAT based on those used by Floreano and Urzelai (2000). These rules were based on synaptic mechanisms observed in mammals (Willshaw and Dayan 1990). The space of possible rules in Floreano and Urzelai's implementation included a *plain Hebbian rule*, which strengthens the connection proportionally to correlated activation, a *postsynaptic rule*, which works like the plain Hebbian rule in addition to weakening the connection when the postsynaptic neuron is active alone, and a *presynaptic rule*, which is like the postsynaptic rule except it weakens the connection when only the presynaptic neuron is active. The specific rule used at each connection and its respective parameters were evolved.

In order to fit these rules into the fewest parameters possible, they were streamlined in NEAT. Instead of dividing Hebbian learning into separate rules, a single general learning rule was evolved for both excitatory and inhibitory connections that combines the properties of Floreano and Urzelai's rules. That way, only two parameters are needed to express a rule, keeping the search space to a minimum. Let x and y be the activities of the incoming and outgoing neurons, respectively, and W be the current highest weight magnitude in the network. If the connection is excitatory, the change in weight magnitude Δw can be expressed as

$$\Delta w = \eta_1 (W - w) xy + \eta_2 W x (y - 1.0) \,, \tag{6.1}$$

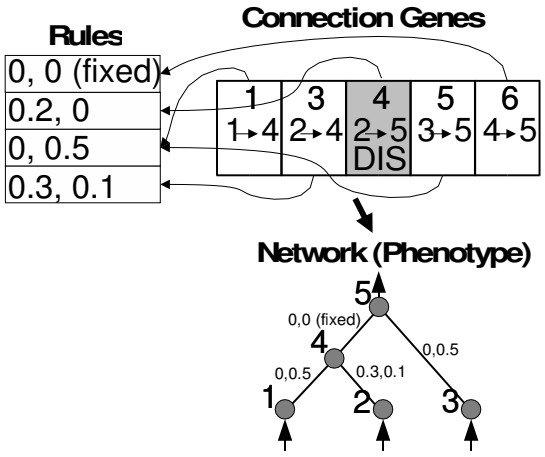


Figure 6.1: **Encoding local adaptation rules.** The connection genes are depicted as in figure 3.2, with gene 4 disabled. Each genome has its own list of four local learning rules, shown on the left. Each connection gene points to one of the four local learning rules and it is possible for more than one gene to point to the same rule (i.e. rules can be reused). The corresponding connection in the phenotype adapts according to the rule pointed to by it respective gene. This way, fewer parameters need to be optimized since rules can be reused. The connections between nodes 1 and 4 and between 3 and 5 both use the same rule.

where η_1 is the Hebbian learning rate and η_2 is the decay rate that controls how fast the connection weakens when the presynaptic node does not affect the postsynaptic node. Inhibitory connections adapt analogously as

$$\Delta w = -\eta_1(W - w)xy + \eta_2(W - w)x(1.0 - y)\ . \tag{6.2}$$

In the inhibitory case, the aim is to *weaken* the connection strength when both the incoming and and outgoing activations are high, since the connected neurons then do not have an inhibitory relationship. Hence, the first term in equation 6.2 is negative. The second term strengthens the connection when the incoming activation is high and the outgoing activation is low, increasing the strength of the inhibitory connection.

To implement equations 6.1 and 6.2, every connection in a plastic NEAT neural network has a local learning rule parameterized by η_1 and η_2, in addition to its evolved weight. Yet if every connection gene expressed its own local learning parameters, the dimensionality of the parameter space would multiply by a factor of 3, reducing the chance of finding a solution. Moreover, it is likely that many connections will utilize the same rule. It should not be necessary to rediscover such a rule for every connection that uses it.

Thus, the NEAT genetic encoding was augmented so that the rules can be reused by more than one connection gene. Instead of having all of the adaptation parameters for each connection and node expressed in every gene, each genome has a single *rule set*. This set consists of a finite number

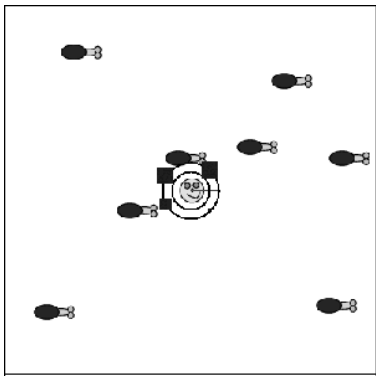


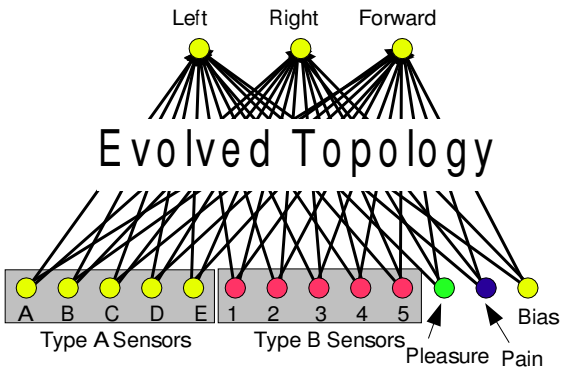
Figure 6.2: **The Dangerous Food Foraging Domain.** The robot begins in the center of the field. The concentric circles around the robot represent one ring of sensors for type A items and one ring for type B. Eight items are dispersed randomly throughout the field. In this case, the items are type B. However, they may or may not be poisonous. The robot can only find out if the items are poisonous by consuming them, after which is must stop foraging if it senses pain. Otherwise, if there is no pain, the robot must consume all the remaining food. This domain requires adaptation because the correct behavior policy depends on the robot's experience.

of rules, in our implementation one "fixed weight" rule and three evolved rules, each containing the adaptation parameters η_1 and η_2. Each gene points to one rule in the rule set. Thus, when a genome is translated into a network, the connections and nodes receive the parameters of the rule to which their genes point (figure 6.1). These pointers can change through mutation. This system accomplishes 3 objectives: (1) The number of learning parameters in a genome *does not* increase as the genome grows; (2) the same rules can be *reused* by many genes; and (3) the adaptation rules can be optimized separately from the connection genes. Thus, using the rule set in NEAT can be seen as an efficient alternative to evolving separate rules for every gene (see appendix A.4.3 for parameters specific to plastic NEAT).

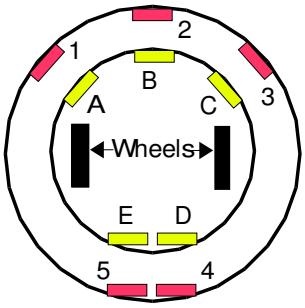
The question, then, is how networks with plastic synapses differ from static recurrent networks. The next section describes a domain designed to answer this question.

6.3 The Dangerous Foraging Domain

In order to analyze the evolution of adaptation, a domain is needed in which adaptation is necessary and where success can be readily measured. Such a domain can be constructed by making the optimal policy depend upon a hidden property of the world that can only be discovered through exploration. That way, once the hidden property is uncovered, the policy must adapt accordingly. A fixed policy, e.g. a neural network with only non-adapting feedforward connections, cannot change its policy, and thus cannot succeed in such a domain. Networks must evolve to adapt to properties native to the particular problem.



(a) Neural Network Controller



(b) Robot Sensors and Wheels

Figure 6.3: **The robot and its controller network (*color figure*).** Five type A sensors and five type B sensors detect the presence of objects around the robot. The pleasure or pain sensors activate for 20 time steps (out of a total of 750 in each trial) if the robot consumes food or poison. These signals give the robot a chance to change its behavior accordingly. The three motor outputs are mapped to forces that control the left and right wheels. Evolution must discover a network that can change the control policy of the robot when it encounters poison.

One such problem occurs in natural foraging. When an animal enters a new geographical area, some food may first appear edible yet turn out poisonous. The animal must be able to change its policy and stop gathering such food. A food foraging domain was implemented based on this dangerous natural scenario. A simulated robot begins in the center of a field with one of two types of items, type A or type B, spread randomly throughout (figure 6.2). The robot attempts several trials, and in each trial the items are either all food or all poison. In each trial, the robot must consume at least one item to find out whether it should continue foraging. After consuming an item, the neural network receives a brief pain or pleasure signal. The correct policy thereafter depends upon this signal.

Because the correct policy, whether to forage or not, cannot be fixed at the start, the network must be able to adapt. The challenge of constructing such a network is significant because it must evolve policies for foraging items of both type A and type B, but also be capable of abandoning either policy if poison is encountered.

The simulated adaptive robots are controlled just as those in the robot duel (Section 5.1). In both the dangerous foraging domain and the robot duel, the robots turn or move based on their neural network outputs and collect items on the board. However, whereas in the robot duel all items are food, robots in the dangerous foraging domain cannot know from their sensors what is food and what is poison. Five rangefinder sensors can sense type A items and five can sense type B (figure 6.3). Finally, each robot has a pleasure sensor, which activates when it consumes food, and a pain sensor, which activates when it consumes poison. The pleasure/pain sensors are used for adaptation.

6.4 Experiments

The experiments are designed to compare plastic networks to static recurrent networks in two ways: (1) Is one type of adaptive network easier to evolve than the other. (2) How do evolved solutions of the two types differ?

Accordingly, NEAT was tested in five 500-generation runs with plastic synapses and five 350-generation runs with static connection weights (the latter runs took fewer generations to converge). All runs were free to utilize recurrent connections but only the plastic runs could change connection weights. Each run took approximately 2 days to complete on a 1.8 Ghz Pentium 4 processor. The NEAT algorithm itself took less than 1% of this computation: The rest of the time was spent evaluating networks in the foraging task.

6.4.1 Experimental Setup

In each run, the population consisted of 500 NEAT networks. Each network was evaluated in eight separate trials: Two trials with edible type A items, two trials with edible type B items, two trials with poisonous type A items, and two trials with poisonous type B items. Networks were reset to their initial state before each trial, that is, internal activations were flushed to zero and synapses were reset to the initial weights defined in the genome.

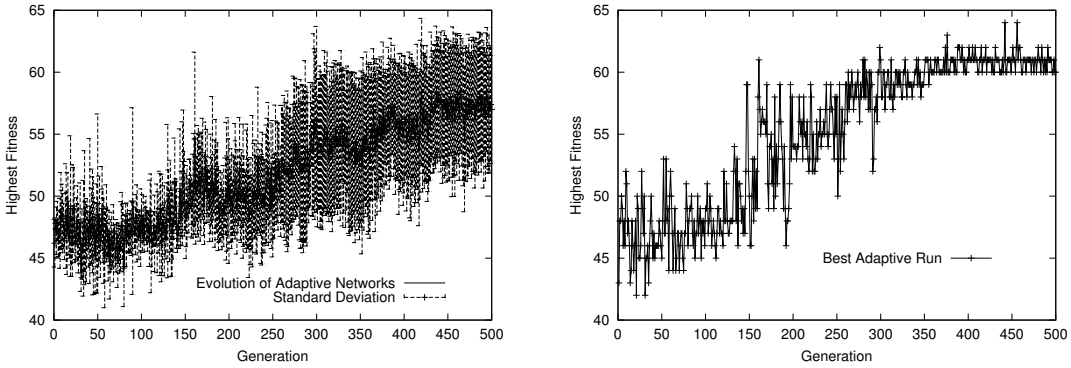
Fitness was evaluated by rewarding networks for consuming food and penalizing them for consuming poison. Since there were 8 items in each trial, and 4 trials consisted of poison, the maximum number of poison consumed is 32. Thus, in order to ensure positive fitness values, the fitness function f was defined as

$$f = 32 + e - p\,, \tag{6.3}$$

where e is the number of edible items consumed and p is the number of poisonous items. The maximum possible fitness is 64. However, in general the highest fitness that can be consistently attained is 60 because the only way to know when poison is present is by testing at least one item in the field. Thus, since there are four poison trials, the best networks need to consume four poisons to properly test for poison in each trial.

Because food is placed randomly in each trial, the fitness function is noisy. Thus, simply reaching a fitness of 60 is not sufficient to indicate that a solution has been found. Rather, a run is deemed successful when the population champion consistently reaches a fitness of 60 for several generations in a row.

NEAT system parameters used in this chapter are described in appendix A.



(a) Average Highest Fitness (b) Best Run Highest Fitness

Figure 6.4: **Average highest fitness and best run of plastic NEAT.** Three of five runs found a consistent solution. Thus, the task can be solved with plastic synapses, although not as reliably as with static synapses.

6.5 Results

While both plastic and static networks were able to solve the task, there were significant differences in their performance. This section examines how consistent and efficient both approaches are, and also what kinds of solutions they tend to produce.

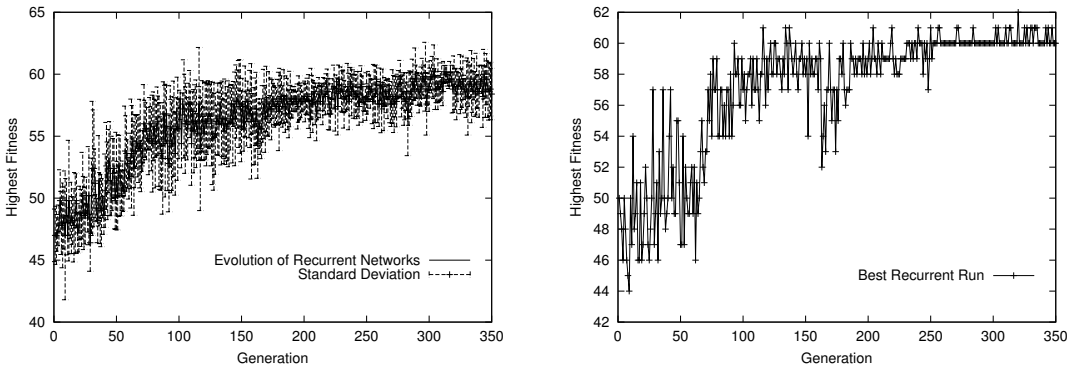
6.5.1 Evolving Plastic Neural Networks

NEAT was able to evolve networks with local learning rules to solve the task. However, solutions with plastic synapses were not always found. Figure 6.4 shows how fitness increased over generations in both the average and best runs of plastic evolution. Three of the five runs converged to consistently scoring 60 or above, whereas the other two runs never found a consistent solution.[1] The best run was able to find a consistent solution by the 350th generation. Since several runs did find solutions, these results show that local Hebbian learning can be utilized to encode dynamic policies.

6.5.2 Evolving Static Recurrent Neural Networks

Figure 6.5 shows that static recurrent networks were also able to solve the task. In fact, all five runs could consistently score 60 or above before 350 generations. The best run found a consistent solution by the 250th generation, 100 generations earlier than plastic evolution. Because static runs always found a solution while plastic runs did not, the variance in fitness was higher for plastic networks than for static networks. In fact, if restarts are taken into account, the number of generations needed to find a consistent static solution is significantly fewer than for plastic evolution ($p < 0.05$). This result demonstrates that state changes inside the networks are sufficient to change its policy.

[1]The reason networks sometimes scored above 60 is that some species evolved a small probability of doing nothing during a trial. Although this behavior sometimes led to a significant drop in fitness when a food trial was missed, in some cases the networks got lucky and skipped a poison trial.



(a) Average Highest Fitness (b) Best Run Highest Fitness

Figure 6.5: **Average highest fitness and best run of static recurrent NEAT.** All five runs discovered solutions within 350 generations, demonstrating that dynamic synapses are not necessary for adaptation in this task. The best run found a solution in only 250 generations. In contrast, the best plastic run found a solution in the same time as the average static recurrent run.

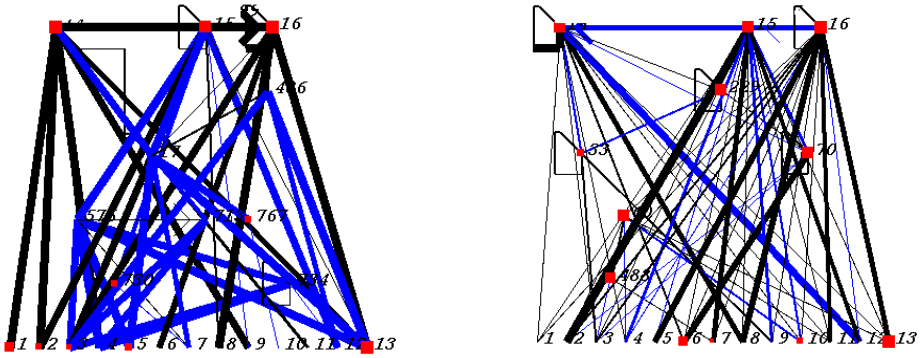
The conclusion is that for this task, evolution of static recurrent networks is more efficient than evolving plastic networks. How does this result come about?

6.5.3 Typical Solution Networks

Let us analyze solution networks from the two types of evolution in order to understand how each kind of network represents a policy that can change over time, and why static networks evolved faster and more reliably.

Figure 6.6 depicts typical solutions from each type of evolution. It turns out that the plastic solution uses a more complicated mechanism than the plastic solution. The plastic solution uses its hidden nodes as part of its policy-changing method. If hidden nodes are ablated, the network can no longer perform the task. In the plastic network, 22% (16) of the connections diverge in opposite directions depending on whether or not the robot discovers it is in a food or poison trial. Of those, 8 connections diverge in *both* type A and type B trials. In other words, an *abstraction* of the food vs. poison distinction evolved that is independent from what the objects look like. Thus, the solution is holistic in that the entire network contributes to the task in every trial. When it finds poison, the plastic network spins in place. Spinning causes the network to continually see food to which it does not react, showing that its policy has changed significantly.

In contrast, most of the structure in the static network, including recurrent connections on hidden nodes, is used to stabilize food gathering trajectories, rather than to modulate behavior depending on pain or pleasure. In fact, the network can still reliably perform the task even if all its hidden nodes are ablated, although it takes longer because its actions are less accurate. The key components of the static solution are the self-recurrent connections on the output nodes. The strong excitatory self-recurrent connection on the *left turn* output (identified in figure 6.3) keeps the node



(a) Plastic (b) Static Recurrent

Figure 6.6: **Solution network examples (*color figure*).** Typical solutions are depicted for plastic evolution and static recurrent evolution. Nodes are shown as red squares beside their node numbers, and line thickness represents the strength of connections. Black lines stand for excitatory connections, and blue lines are inhibitory connections. Loops at nodes represent self-recurrency. (a) The plastic solution is holistic, utilizing plastic synapses throughout the network. (b) The static network solves the task using recurrent connections on its outputs. While both solutions take different approaches, they are both possible through NEAT's ability to evolve network topologies.

active even when it has low input. The only mechanism that can stop the self-activation cycle is a strong inhibitory signal directly from the pain sensor, in which case the left turn output node is temporarily disabled. At the same time, the *right turn* output turns off unless food is directly in front of the robot. Thus, when the left turn output is temporarily muted, the right turn output will cause the robot to spin until it is facing away from food. At this point both turn outputs will be off, and the robot will dash forward through the open space to a wall, thus avoiding any further food gathering. Therefore, instead of spinning like the plastic network, the recurrent network drives into a wall after consuming poison. Thus, recurrent connections *are* used to represent a dynamic policy. This clever solution demonstrates the power of evolution in finding effective novel solutions.

Other plastic and static recurrent solutions followed similar patterns; plastic solutions tended to make complex internal network changes while recurrent solutions tended to rely exclusively on a "trick" using recurrent connections on output nodes. These results explain why recurrent solutions were found more easily. These networks only need to find a particular combination of recurrent connections on outputs, and then use hidden nodes to refine trajectory control. Plastic networks, on the other hand, tend to discover complex holistic solutions that genuinely change network functionality. Such holistic solutions are more difficult to evolve.

6.6 Discussion

Static recurrent networks were able to solve the dangerous food foraging task with fewer evaluations. Recurrency alone was successful in this experiment because a relatively simple solution existed in the space of static recurrent networks. This result suggests that if such a solution exists, finding it can be easier than finding a plastic solution. In addition, evolving network topology makes it possible to find such clever solutions; searching for such a solution in the fully-connected recurrent architectures used in prior research (Floreano and Urzelai 2000; Aharonov-Barki et al. 2001; Blynel and Floreano 2002) would require finding appropriate weights for many unnecessary recurrent connections.

Plastic solutions were more holistic: Many connections changed their weights concurrently in order to adjust the overall network behavior. The general conclusion is that static recurrent networks and plastic networks may be appropriate for different kinds of tasks requiring adaptation. Static recurrent networks work well in domains that require a single definitive state change from one behavior pattern to another. This conclusion is also supported by the discovery of a command neuron in a static recurrent network by Aharonov-Barki et al. (2001). On the other hand, when general properties of the environment change such as brightness or sensory acuity, it may be important for the whole network to be able to react by adjusting its weights accordingly. The fact that the same connections changed their weights in both type A and type B trials suggests that plastic networks do indeed adjust to global environmental changes. In fact, Blynel and Floreano (2002) showed that plastic networks adapt well when the environment of a robot becomes darker, further supporting this view. Thus, the comparison in this chapter solidifies an emerging picture of how static and plastic networks can serve different roles.

Why did the plastic networks, which were also able to use recurrent connections, not evolve the same style of solutions as the static networks? As connection weights became dynamic early in evolution, simple solutions based on recurrent output nodes were no longer feasible because the network weight configuration was not reliable. Thus, evolution was forced to utilize the dynamic synapses in order to master the task, leading to more holistic solutions.

Interestingly, figures 6.5 and 6.4 show that solutions occasionally exceeded a fitness of 60. Given that such a fitness requires the network not to collect a single food item in some trials, how was this result possible? That is, how can the networks know which trial contains poison without collecting at least one item? The answer is that solutions had a small probability of switching off their food gathering policies without any pain or pleasure input. Of course, doing so could lead to a major drop in fitness if it happened during a food trial. However, while such failures did occur for some members of a species, other members would *get lucky* and just happen to choose the right trial to stay put. In other words species *colluded* in order to obtain extremely high fitness. As long as a species was large enough, it could afford the slight risk of a member occasionally missing a food trial because another member would probably get lucky and make up for it. It may be necessary to adjust fitness functions to prevent such collusion in the future.

An important goal for future research is to compare adaptive networks in more complex domains so that we can begin to understand the limits of different types of adaptation. In addition, plastic networks traditionally change their connection weights based on Hebbian rules, but other kinds of learning rules are also possible (Section 10.4.1), and should be tested in the future as well.

6.7 Conclusion

Both plastic networks and static recurrent networks were evolved in a dangerous food foraging task. The only way to succeed in the task is to be able to switch off the foraging behavior in the middle of a trial. Plastic evolution found more complex holistic solutions while static networks exploited a clever strategy of switching their internal state, represented by recurrent connections. The conclusion is that different kinds of adaptation are appropriate for different problems, and that topology is an important feature of adaptive networks.

Chapter 7

Application 1: A Roving Eye for Go

The performance of AI methods in Go lags behind other board board games, which makes it a popular and challenging testbed for different techniques (Bouzy and Cazenave 2001). A unique feature of Go is that it can be played on a range of different board sizes. Ideally, a neural network evolved to play Go on a small board could continue evolving on larger boards, thereby building on prior experience. In this chapter, NEAT evolves such a neural network. The network controls a *roving eye* that voluntarily moves its narrow field of view over the board and uses evolved recurrent connections to remember what it saw. Because the roving eye's input field is the same size for any board, the same network can potentially scale to larger boards. This chapter puts the roving eye to the test, showing how it scales with increasing board size, and how NEAT uses such scaling to evolve effective players.

7.1 Motivation

In board games, human players analyze the position by scanning the board with their eyes. Even though players do not see the entire board at once during such a scan, they are able to make intelligent decisions because they *remember* what they saw. Recurrent connections, which send signals backwards through an artificial neural network, allow the network to maintain state and hence memory from one timestep to the next. Thus, NEAT can evolve a roving eye that scans the input space piece by piece, using its memory to keep track of the important parts of the input as they pass out of the visual field.

Board games are a natural application of this idea since they generally involve an input space of an entire board with more than one possible piece at each position. Among the most challenging such games is Go. Since designing the strategy of a good player by hand is very difficult in Go, machine learning is a popular approach (Enzenberger 2003). The idea is that a sufficiently powerful ML algorithm can learn strategies and tactics through experience that are otherwise difficult to formalize in a set of rules.

The ultimate goal is to train a player on the full-sized 19×19 board. However, the game is increasingly difficult on larger boards because the search space grows combinatorially and it becomes increasingly difficult to evaluate positions. Thus, current methods have been successful only on smaller boards (Lubberts and Miikkulainen 2001; Richards et al. 1998). Such results demonstrate the method has potential, with the hope that it may be possible to scale it up to larger boards in the future.

Ideally the knowledge gained on small boards could bootstrap play on larger ones. A process that could make use of such shared knowledge would be a significant step towards creating learners that scale. Previous neuroevolution work in evolving Go focused on networks with the number of inputs and outputs chosen for a particular board size, making it difficult or impossible to transfer to a larger board (Lubberts and Miikkulainen 2001; Richards et al. 1998).

A different approach is taken in this chapter; instead of a network that sees the entire board at once, a roving eye is evolved with a small visual field that can scan the board at will. The roving eye uses the recurrent structure of the neural network to integrate input information it acquires as it scans the board. While scanning, the roving eye decides when and where to place a piece. Because it has the same field size on *any* board size, a roving eye evolved on a small board can be further evolved on a larger board without losing experience. Thus, the roving eye architecture promises to scale.

The roving eye's scalability is tested by comparing an eye evolved on a 7×7 board with one first pre-evolved on a 5×5 and then further evolved on a 7×7 board. The pre-evolved eye became a significantly better player. In the next two sections prior work in machine learning and neuroevolution for Go is reviewed, and also prior implementations of roving eyes in AI. Section 7.5 describes the experimental methods and results.

7.2 Machine Learning and Go

Go is a difficult two-player game with simple rules, making it an appealing domain for testing machine learning techniques. The standard game is played on a 19×19 grid. The two players place black and white pieces alternately at gird intersection points. The game ends when both players pass, which usually happens when it becomes clear that no further gains can be made. The winner is determined by the final score. In the experiments in this chapter, Japanese scoring is used, which counts the amount of territory surrounded by each player excluding their own pieces.

Thus, the object of the game is to control more territory than the opponent. Any area completely surrounded by one player's stones is counted as that player's territory. If the opponent's stones are completely surrounded, those stones are lost and that area is counted towards the other player. If there is an open intersection, called an *eye*, in the middle of the opponent's stones, that intersection must also be filled in order to surround the stones. A structure with two eyes cannot be captured because it is not possible to fill both eyes at once. The *ko rule*, which forbids the same

board state from occurring twice, ensures that the game progresses to a conclusion.

The rules of Go are deceptively simple. Yet unlike in many other board games, such as Othello and chess, machines cannot come close to master-level performance in Go. Not only are there generally more moves possible in Go than other two-player, complete information, zero-sum games, but it is also difficult to formulate an accurate evaluation function for board positions (Bouzy and Cazenave 2001). However, the game can be simplified by playing on a smaller board (Richards et al. 1998). The smaller the board, the simpler the strategy of the game. At the smallest possible board size, 5×5, the game becomes largely a contest to control one side of the board. However, even at this very small scale, fundamental concepts must be applied, such as forming a line of connected pieces and defending the center. Although these concepts alone are not sufficient to play well on a larger board, they are nevertheless a foundation for more developed strategies, making smaller boards a viable platform for testing machine learning methods. While 5×5 Go may be much simpler than 19×19 Go, it is still related to 7×7 Go, which is related to 9×9 Go, and so on.

A number of general AI techniques that are not based on learning have been applied to Go, such as goal generation, game tree search, and pattern-matching (Bouzy and Cazenave 2001; Cazenave 2000). Gnugo 3.2 is a publicly available, open source Go playing program that includes many of these techniques (available at `www.gnu.org/software/gnugo/gnugo.html`). Gnugo is on par with current commercially available Go playing programs. However, Bayer et al. (2002) note that no existing program is even competitive with an amateur shodan, which is the designation for a technically proficient human player.

Writing a program to play Go directly is difficult because a large amount of strategic knowledge must be coded into the system. Therefore, machine learning methods that generate their own knowledge through experience are an appealing alternative. For example, Enzenberger (2003) created a program called NeuroGo that links units in a neural network corresponding to relations between intersections on a Go board. In 2001, NeuroGo ranked in the top third of computer Go playing programs (Bouzy and Cazenave 2001), showing that machine learning is competitive with hand-coded knowledge.

NE has been applied to Go on smaller board sizes in the past (Lubberts and Miikkulainen 2001; Richards et al. 1998). However, these experiments evolved neural networks for a single board size, wherein each intersection on the board was represented by a discrete set of inputs and outputs. Such representations cannot easily scale to other board sizes because the number of inputs and outputs in the network are only compatible with the single board size for which they were designed. In contrast, a roving eye neural network uses the same number of inputs and outputs regardless of the board size. The next section reviews prior research on roving eye systems outside the game-playing domain.

7.3 Roving Eyes

A *roving eye* is a general concept in machine vision, referring to a visual field smaller than the total relevant image area; such a field must move around the image in order to process its entire contents. Roving eyes are often used in robots, where they allow successful navigation even with a limited sensory radius (Hershberger et al. 2000). This type of purposeful control of a moving visual field is also sometimes called *active vision* (Dickinson et al. 1997).

Instead of hardcoding the control laws that guide the movement of the roving eye, Pierce and Kuipers (1997) showed that they can be learned through experience in the world. This research sets a precedent for the automatic design of roving eyes when the proper control or processing rules are unknown.

Most relevant to game playing are experiments where a roving eye must learn to recognize objects that are too big to process all at once. Fullmer and Miikkulainen (1992), and Nolfi (1997), trained neural networks using neuroevolution to control situated robots that had to move around an object in order to determine its identity. Fullmer and Miikkulainen evolved simple "creatures" that move around a shape on a grid and must learn to move onto only certain shapes. In Nolfi's experiment, a robot moves around an object in order to determine if it is a wall or a type of cylinder. In both cases, both the movement control and the final classification action were evolved simultaneously as neural network outputs.

Such neuroevolved shape-recognizing roving eyes provide inspiration for using a roving eye in a board game as well. However, instead of performing a simple classification, the eye must scan the board and then decide where and when to place a piece. For such a technique to work, the controller for the eye must have memory: It must relate different parts of the board to each other even though they cannot be within its field at the same time. Thus, NEAT is a natural method for evolving a roving eye. The next section describes how NEAT evolved neural networks to control a roving eye that scans a Go board.

7.4 Experimental Methods

The experiments are designed to answer two questions: (1) Is the roving eye a scalable architecture, i.e. can it play on more than one board size? (2) If so, can learning to play on a small board facilitate further evolution on a larger board? Specifically, does a proficient 5×5 player provide a better starting point for evolving a 7×7 player than evolving from scratch?

7.4.1 Evolving Against Gnugo

Roving eye neural networks were evolved to play black against Gnugo 3.2 using Japanese scoring. Gnugo is deterministic; it always responds the same way to the same moves. Thus, solutions were not evolved to be general-purpose players (although they did evolve some general skills), but rather

to defeat Gnugo. While it is possible to devise a more general-purpose competition, e.g. by using coevolution, playing Gnugo makes it easy to interpret results and clearly illustrates how the roving eye scales against a known benchmark.

Another advantage of Gnugo is that it estimates the score for every move of the game, as opposed to only the last one. This estimate makes fitness calculation more effective than one that uses only the final score of the game, because the quality of play throughout the game can be taken into account, instead of only at the end. Fitness was calculated from the cumulative score estimate as well as the final score as follows:

$$f = 100 - (\frac{2\sum_{i=1}^{n} e_i}{n} + e_f), \tag{7.1}$$

where e_i is the score estimate on move i, n is the number of moves before the final move, and e_f is the final score. This fitness equation weighs the average estimated score twice as much as the final score, emphasizing the performance over the course of the entire game over the final position. Such a fitness allows selecting promising networks even early in evolution when the network is likely to lose all its pieces. Because Gnugo returns positive scores for white, they must be subtracted from 100 to correlate higher fitnesses with greater success for black.

Neural networks were evolved to control a roving eye in both 5×5 and 7×7 evolution. In half the 7×7 evolution runs, the champion of the 5×5 run was used to seed the initial population, i.e. the 5×5 champion's connection weights were slightly mutated by adding a number from a uniform distribution to randomly selected weights to form the initial population for 7×7 evolution.

A special modification had to be made in order make Gnugo a useful opponent: Because Gnugo will pass as soon as it determines that it cannot win, Gnugo will pass in 5×5 Go as soon as black plays in the middle since there is a winning strategy for black starting from this position. Therefore, to force a continuing game, white was forced to play the intersection directly adjacent and right of center for its first move. That way, Gnugo would play a full game. In order to be consistent, white was forced to make the same initial move on the 7×7 board as well. This modification does not make the results less general since the aim is to encourage the roving eye to improve by playing full games against Gnugo, which was effectively accomplished. NEAT system parameters used in this experiment are specified in appendix A.

7.4.2 Roving Eye

The roving eye is a neural network evolved with NEAT. The neural network's sensors are loaded with the visual field of the roving eye, and its outputs determine how the eye should move, or decide that the eye should stop moving and the network should place a piece on the board.

The state of the roving eye consists of its position and heading. It can be positioned at any intersection on the board, and be heading either north, south, east, or west. The ability to see the board from different headings allows the roving eye to process symmetrical board configurations the

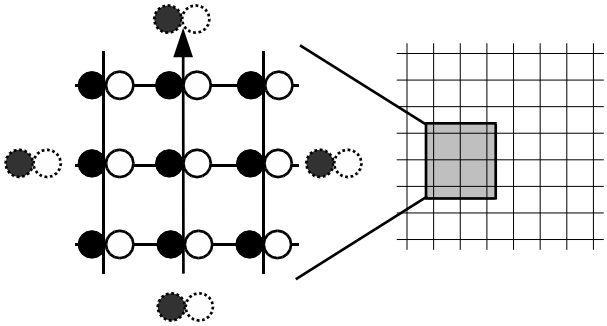


Figure 7.1: **The roving eye visual field.** At each of the nine intersections visible to the roving eye, it has one black sensor and one white sensor, depicted as filled circles. In addition, the dotted circles represent long-range front, left, right, and back sensors, which only tell how many pieces of each type are in each of those four zones outside the visual field. The arrow shows the eye's current heading. The gray square on the 7×7 board shows the size of the visual field relative to the board. The neural network controlling the roving eye also receives 2 inputs representing absolute position, an illegal move sensor that alerts the network to ko (Section 7.2), and a bias input. This architecture allows the same roving eye to operate on any board size, making scalable Go players possible.

same way. In other words, unlike full-board neural network players, the roving eye does not need to evolve separate components for symmetric positions. Instead, the eye can simply turn around to see the board from an identical perspective, which is easier to evolve.

The visual field includes nine intersections with the eye positioned at the center (figure 7.1). For each intersection, the eye receives two inputs. If the intersection is empty, both are zero. For a black or white stone, the first or second input is fully activated. For a border, both inputs are active. In addition, to help the eye decide where to look, it is given a count of how many white and black pieces are outside its visual field to its front, left, right, and back. The eye is also fed two inputs representing its absolute position on the board, a single input that specifies whether playing in the current position would be illegal due to ko, and a constant bias. Thus, in total, the eye receives 30 inputs regardless of board size (figure 7.2).

Five outputs determine the eye's next action. It can place a piece at its current position, move forward in its current heading, turn left, turn right, pause (which may give it more time to "think," i.e. process recurrent activations), or pass. If any of the movement outputs are above 0.5, the eye will move regardless of the other output values, and the movements are combined. For example, the eye can move forward and turn left at the same time. However, if both turning outputs are above 0.5, the eye defaults to turning left. If no movement outputs are active, the greatest output over 0.5 is chosen. If no output is sufficiently active, the eye pauses for one time step. If after 100 time steps the eye still does not make a choice, it is forced to pass. Finally, if the roving eye attempts to place a piece on top of another piece, the move is considered a pass.

In general, the roving eye scans the board by moving around until it finds an acceptable location, and then places a piece. In this way, the roving eye analyzes the board similarly to hu-

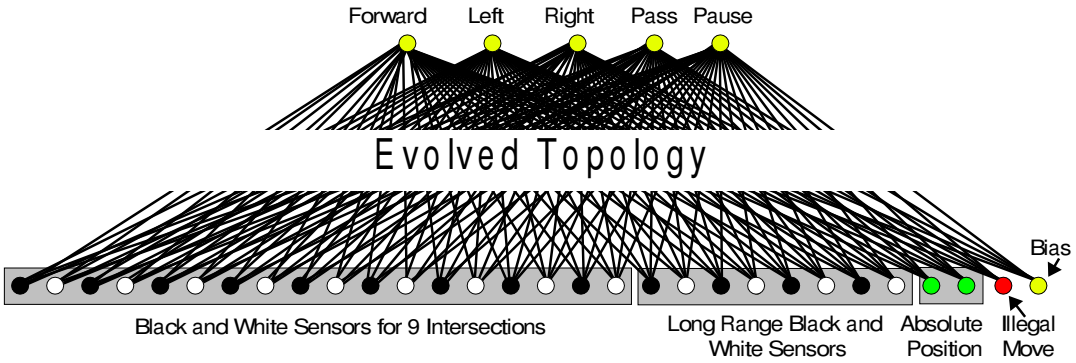


Figure 7.2: **Roving eye neural networks.** The first 18 inputs include a pair of black and white sensors for each of nine intersections the eye can see at one time. Another eight inputs detect pieces outside the immediate visual field (figure 7.1). Two more inputs specify the eye's absolute board position. The illegal move input activates if the eye is centered on a ko position. The five outputs let the eye move forward, left, or right, pass, or pause without moving. This configuration was found effective through extensive preliminary experimentation.

mans, who also focus attention on specific areas of the board while considering a move rather than processing the entire board at once.

7.5 Experiments

In every run, the roving eye was playing black, the first player to move, and Gnugo white. Ten runs of 5×5 evolution were performed. The champion of a typical run, with fitness 99, was chosen as the seed genome for 15 runs of 7×7 evolution. Another 15 runs of 7×7 evolution were started from scratch, without the seed.

7.5.1 5×5 Champion

NEAT improved significantly against Gnugo in 5×5 evolution. In early generations, NEAT rarely placed a piece without losing it soon after. By the 400th generation, NEAT was able to control the center of the board and thereby capture more area than Gnugo. NEAT learned the general principle of connectedness, and also that it is important to maintain a forward front. The champion roving eye's game is shown in figure 7.3.

Figure 7.4 shows what happens when the 5×5 champion plays directly against Gnugo on a 7×7 board. Interestingly, the champion shows some of its 5×5 capabilities, forming a semi-contiguous line. However, when its line fails to connect, and is not sufficient to cover the larger 7×7 board, the roving eye is quickly surrounded by Gnugo without making any attempt to respond.

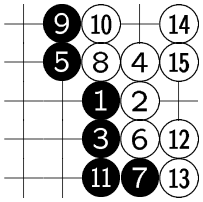
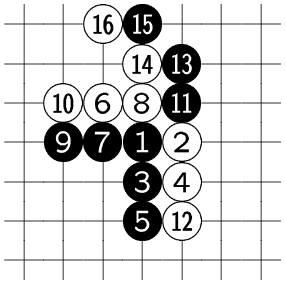
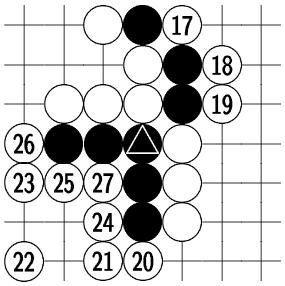


Figure 7.3: **A game by the 5×5 champion.** The roving eye (black) is able to control more of the board by pushing its border as far to the right as possible against Gnugo. Gnugo is unable to mount an attack, and instead reinforces its position by creating two eyes. If Gnugo tried to capture area to the left of the border, it would be open to capture since black surrounds the area. This roving eye was used as the starting point for evolution on the larger 7×7 board.



(a) Black attempts a border



(b) Black is surrounded

Figure 7.4: **The 5×5 champion plays Gnugo on a 7×7 board.** (a) The roving eye (black) attempts to form a border as it did in 5×5 Go (figure 7.3). However, because the board is larger, it only partially succeeds. (b) Having never experienced such a conclusion, black does nothing but pass while Gnugo easily eliminates its opponent. The game shows that the 5×5 roving eye is able to apply some of its knowledge to 7×7 Go, although its former strategy is no longer entirely successful.

This behavior makes sense because in the 5×5 game, as soon as the roving eye finished building its border, Gnugo would not attack and the game would end (see figure 7.3). However, in 7×7 Go, this strategy is no longer sufficient. The question is whether the knowledge contained in the 5×5 champion will benefit further NEAT evolution on the 7×7 board.

7.5.2 Evolving 7×7 Players

Evolution on the 7×7 board is indeed significantly more effective starting from the pretrained 5×5 champion than starting from scratch (figure 7.5). Not only does initial fitness rise significantly faster over the first few generations, but it remains higher throughout the run, suggesting that starting raw may never reach the performance level of a roving eye that has already learned about Go on a smaller board. This result establishes both that (1) the roving eye can be used on more than one board size, and (2) evolving on a smaller board captures information that is useful on a larger board.

During 7×7 evolution, NEAT was able to reorganize and expand the original 5×5 strategy

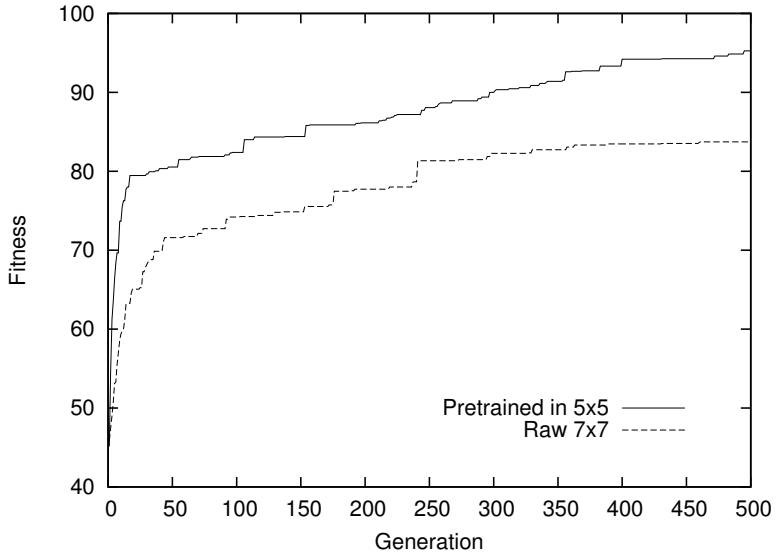
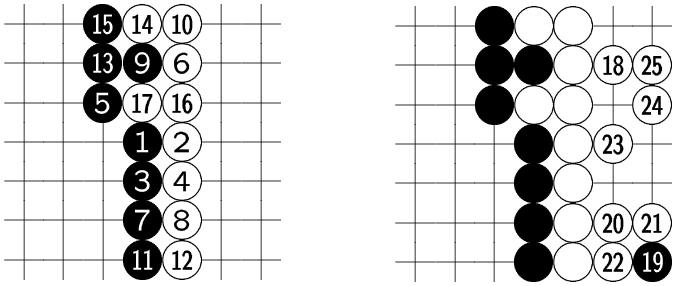


Figure 7.5: **Average fitness on a 7×7 board over generations.** The average fitness (equation 7.1) of the best roving eye in each generation is shown over 15 runs of raw 7×7 evolution, and 15 runs of 7×7 evolution pretrained in 5×5 Go. Pretrained evolution has significantly ($p < 0.05$) higher fitness in generations $2-237$, $383-451$, and $495-500$, showing that pre-evolving on a smaller board leads to both a higher start and end to evolution.



(a) Black controls more area (b) White does not respond

Figure 7.6: **Typical 7×7 champion pretrained in 5×5.** The figure shows two halves of a game played by a 7×7 champion. that descended from the 5×5 champion (figure 7.3). (a) The roving eye has learned to form a clean border with more territory enclosed than Gnugo. (b) The roving eye plays one piece in the corner, because it has evolved to exploit Gnugo's tendency to occasionally place unnecessary pieces (which lowers its score under Japanese rules). Gnugo finishes by creating two eyes, but cannot mount an attack on the roving eye. Thus, the eye finishes with more territory. Starting evolution from a pretrained 5×5 champion helped the roving eye achieve this level of play on the 7×7 board.

in order to form an effective border (figure 7.6). While evolving from scratch required NEAT to discover how to connect contiguous intersections, networks evolved from the pretrained 5×5 champion could make such connections from the start. In fact, on a larger board, discovering the same early principles takes longer because the game is significantly more complex. Therefore, the raw-starting roving eye was at a disadvantage throughout evolution. In contrast, the pretrained roving eye quickly rises within 25 generations to a level of play that takes raw-starting evolution ten times longer to achieve!

7.6 Discussion

The roving eye allows scaling the skills learned on smaller boards to larger boards. This is an important result because current techniques do not succeed in learning to play on larger boards. Therefore, roving eye neuroevolution could turn out to be an important component of a competent learning system for Go.

The play demonstrated in this chapter is not at championship level. Since black has the first move, it is not surprising that it ultimately controls more area. In addition, it has only learned to play against Gnugo, and would be likely to fail against a more skilled or significantly different opponent. An important question for the future is how one might evolve a world-class Go player using such a scaling technique. That is, can we apply the roving eye methodology in a way that more general, robust strategies would emerge?

First, the roving eye needs to be evolved on significantly larger boards. There are substantial differences between 7×7 and 19×19 Go. The larger space allows larger patterns to be formed, and evolution will take longer to make use of them. However, rudimentary skills such as the ability to surround enemy pieces or place a contiguous line should still constitute a useful starting point for future learning.

Second, to encourage better generalization, roving eyes could be coevolved instead of evolving them only against Gnugo (Bayer et al. 2002). Gnugo was used as an opponent first because it is a well-known benchmark, but in the future roving eyes should play against each other to force them to handle a variety of opponent strategies. Well-founded coevolution methodologies such as Host-Parasite (Chapter 5), Hall of Fame, or Pareto coevolution should be used to develop the most well-rounded players possible (Rosin and Belew 1997; De Jong 2004; Ficici and Pollack 2001). Chapter 5 showed that combining complexification in NEAT and competitive coevolution leads levels of sophistication unreachable through evolving neural networks of fixed-topology. This process should work well in the Go domain as well.

Third, the roving eye can be combined with game tree search techniques such as α-β. While the state space of Go is too large to be searched directly, a roving eye may help prune the search by acting as a filter that approves or disapproves of searching different positions (as was done by Moriarty and Miikkulainen 1994). In this manner, the search can be made significantly more efficient, and the network does not have to attempt to look ahead to the end of the game. Such

hybrid techniques constitute a most promising direction of future research in the long run.

7.7 Conclusion

The roving eye architecture is an appealing approach to Go because it is the same for any board size. It is also powerful because it can turn to face different directions, allowing it to process symmetrical configurations with the same connections. Thus, the roving eye is a potentially important component of learning systems that aim to perform well on larger boards even when learning directly on such large boards is prohibitively complex. An important conclusion is that NEAT was able to evolve networks that can scan only part of the board and use their memory to make up for the missing inputs. This approach to problems with many inputs can be applied to other domains as well, such as processing large visual fields. It is possible in NEAT because of its ability to build recurrent networks.

Chapter 8

Application 2: Automobile Warning System

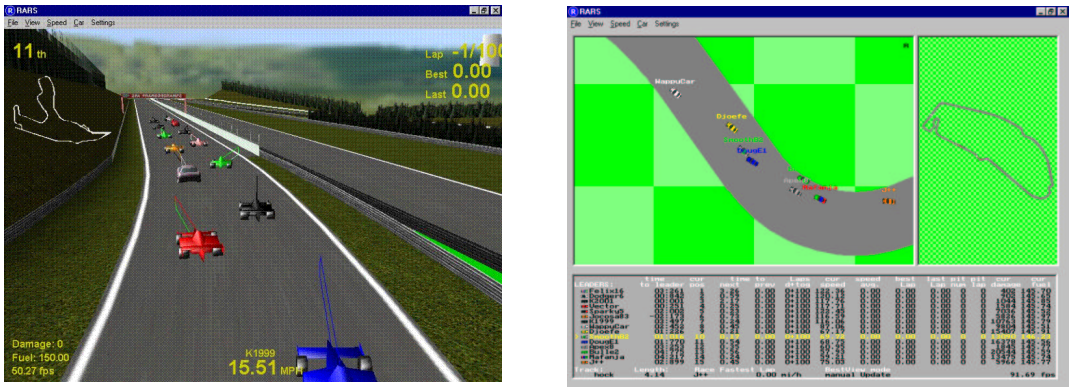
While evolving strategies and behaviors is a natural application of neuroevolution, it can also be used to augment *human behavior* in potentially dangerous situations. This chapter focuses on the commonplace yet frequently dangerous activity of driving an automobile. *Warning networks* are evolved to let drivers know when they are at risk; in this role, neural networks have the potential to save lives. This chapter describes results from the first year of an ongoing project on warning networks, and demonstrates that NEAT makes this new kind of application possible.

8.1 Motivation

If cars could warn their drivers that a crash is imminent, it is possible many accidents could be avoided. One approach for building such a warning system is to ask an expert to describe as many dangerous situations as possible and formalize that information in an automated reasoner that reacts to sensors on the car. However, the circumstances leading to a crash are frequently subtle and may vary for different drivers. Moreover, it may not be possible to predict a crash from a static snapshot of the road: the recent history of the car and other objects on the road may have to be taken into account as well. It is difficult to know how long such a history should be or what it should be tracking.

Yet if the car could learn *on its own* what to track and how long to keep salient events in memory, these challenges could be overcome. In addition, cars could be trained with different drivers under different circumstances, creating more flexible warning systems.

Teaching a car to predict crashes is the goal of the automobile warning system project at the University of Texas at Austin, started in November 2003 and funded by Toyota. NEAT is a natural choice for the learning method because NEAT can develop arbitrary recurrent neural networks that keep a variable length of prior history in memory. In other words, an expert does not need to decide



(a) 3-D Overhead View (b) 2-D Overhead Display

Figure 8.1: **RARS screenshots.** The screenshots show two views of a race in RARS. (a) In the 3-D view, lines projecting in front of the cars show their current trajectories. (b) The 2-D view shows more of the track at the same time, and also a full map and current rankings. Both views can represent the same race, which can include any number of cars operated by independent controllers. Because RARS is a popular platform that accurately simulates vehicle physics and supports multiple simultaneous drivers, it makes a good simulation testbed for evolving warning systems.

how long the warning window should be or what it should take into account, because the recurrent topology can evolve to make this determination on its own.

Yet if NEAT is to evolve crash predicting networks, it must be trained by observing driving behavior. What kind of driver would be willing to provide the hundreds or thousands of examples necessary to train a warning network? Conveniently, NEAT can evolve the drivers in simulation before it evolves warning networks. In fact, NEAT's past successes with evolving controllers suggests that it should be able to evolve robust and varied driving behavior. Furthermore, evolved driving networks can be impaired in order to simulate dangerous driving conditions.

This chapter describes the first experiments in evolving drivers and predictors with NEAT, beginning with a discussion of the RARS simulator used to train both the drivers and predictors.

8.2 The Robot Auto Racing Simulator (RARS)

Since learning requires experience, it is necessary for NEAT to gain experience through driving and predicting crashes. Even though the learned system should eventually be deployed in real cars, crashing cars in the real world while the system is learning is not possible; therefore, a reasonable alternative is to evaluate NEAT in simulation. RARS (`http://rars.sourceforge.net/` ; figure 8.1), a public domain racing simulator designed for AI testing and real-time control, was chosen for this purpose.

RARS has several features well-suited to this project. RARS is supported by an active

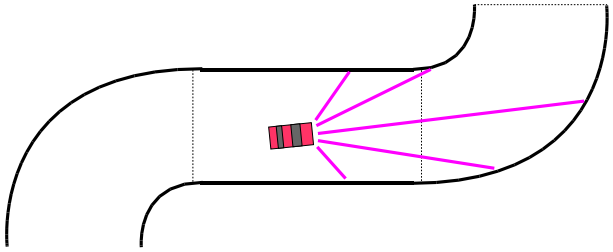


Figure 8.2: **Rangefinder sensors detect the border.** Simulated rangefinders project lines to the edges of the road and measure the distance to the intersections, giving the car a sense of its position and the road's curvature. RARS' native data structures were converted into rangefinder data so that NEAT could train neural networks with a realistic egocentric input.

community that provides documentation and support. The software was written with AI in mind, so it is easy to modify and plug in new drivers. Vehicle dynamics are accurately simulated, including skidding and traction. Multiple automobiles controlled by different automated drivers can race at the same time. Finally, the software automatically provides information like the distance between the driver and other vehicles and the direction of the road that can be used as the basis for simulated sensors.

Using raw data provided by RARS, two kinds of sensor systems were developed for this project and provided as input to NEAT neural networks. First, rangefinder sensors project rays at several angles relative to the car's heading to the edge of the road (figure 8.2). The rangefinders give the car an indication of its position and heading relative to the sides of the road, and also of the curvature of the road.

Second, several simulated radar sensors detect other cars or obstacles (figure 8.3). The radar sensors are convenient for detecting discrete objects by locating them inside of one of several slices that represent relative angles and positions. The radars return a value equivalent to the distance of the nearest car in each slice, or a maximum value if there is no car.

It is sensible to ask whether this sensor configuration is a reasonable approximation of the real world, i.e. can we expect results to transfer? If similar information can be extracted from real-world sensors, then the simulation is likely a good starting point. In fact, significant research has gone into detecting other cars (Thomanek et al. 1994) and lanes (Dickmanns and Mysliwetz 1992) in the real world, and data from such systems could be processed and fed into NEAT neural networks in a similar form to the sensors used in these experiments. Of course, in the future it will also be desirable to simulate more sophisticated processed sensors such as vehicle trackers (Haag and Nagel 1999).

RARS provides a virtual gas pedal, break, and steering wheel that can receive their values from the outputs of a neural network. The gas pedal and break are interpreted as a requested tire speed relative to the bottom of the car. There is no limit to how high the request can be, and RARS

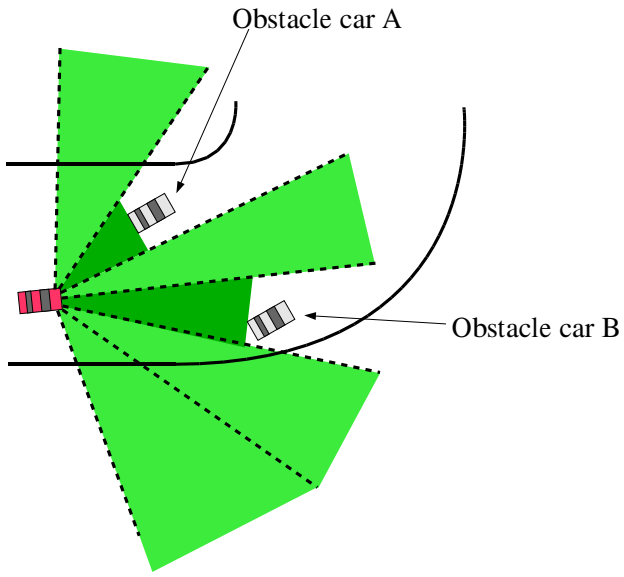


Figure 8.3: **Radar sensors detect other cars.** Radar sensors return the distance to the nearest car in the sensor slice. For example the second slice clockwise from the top detects obstacle A and the fourth detects obstacle B. All others return a maximum distance indicating there is nothing in range.

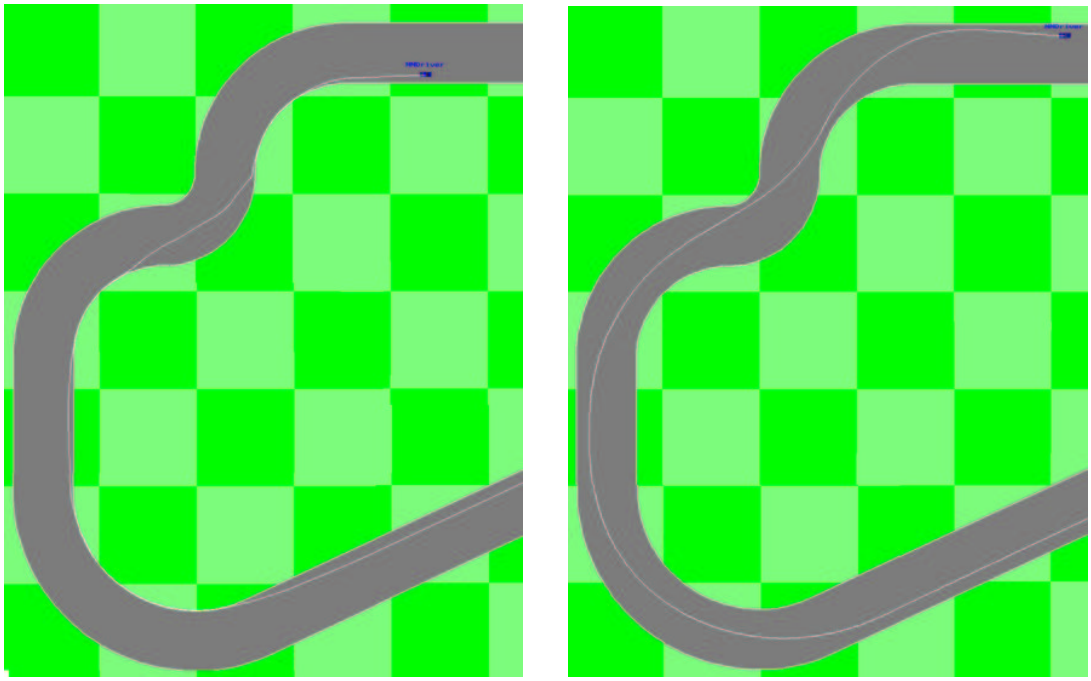
tries to match the request within the physical constraints of the car. In addition, if the request is lower than the current speed, RARS attempts to slow the car down by breaking. The steering request is treated similarly; the lateral force generated by the same turn angle request increases the higher the current speed. Thus, the driving controls in RARS work like a real automobile.

Races can be set up in RARS with one or more drivers. A natural way to begin training drivers is on an open road without other cars, as described in the next section.

8.3 Training Drivers on an Open Road

Drivers were evolved on an open road with NEAT, and these driving networks were later used to train crash predictors. Starting with an open road makes sense because it is the simplest proof of concept that crash prediction can work, and serves as a basis for further evolution in more complex scenarios. Evolution naturally produces drivers with a range of skills from wobbly, crash-prone drivers in early generations to fast, reliable drivers later in evolution. For training crash prediction, it makes sense to use drivers that are not perfect so NEAT can get experience with real crashes. Thus, the range of drivers makes it possible to pick good training cases without the need for handcoding.

Extensive testing revealed that skilled open-road drivers can be best evolved with seven rangefinder sensors. During evolution, each neural network in the population was evaluated over



(a) Early Evolution (b) Late Evolution

Figure 8.4: **NEAT discovers intelligent turning.** NEAT was able to improve steering significantly. (a) Early in evolution, the car hugs the inside of the turn, a naive strategy that minimizes driving distance but results in overall slow speed. (b) After approximately 400 generations, the driver learns to anticipate the turn by steering to the outside so it can attain maximum acceleration coming out of the turn. Notice the car takes an almost straight line through the most curvy section of the road. NEAT discovers this counter-intuitive behavior, which is hard to learn even for humans.

three trials. Each trial lasted 1,000 simulated timesteps. which is long enough to go around the track once. The network's fitness was the average score over the three trials. The score for a single trial was

$$S = 2d - b, \tag{8.1}$$

where b is the damage incurred over time spent out of bounds, i.e. off the track, and d is the distance traveled. Damage is computed by RARS internally, and is proportional to time off the track. This fitness function penalizes crashing and rewards speed. Additional evolution parameters are described in appendix A.

NEAT was able to evolve perfect drivers that never crash. Although the purpose of this project is to evolve warning networks and not the fastest drivers possible, driving times for evolved drivers were still comparable to the best hand-coded drivers provided with RARS (Table 8.1). Figure 8.4 shows the trajectory of a champion vehicle on the track. The driving network learned to take turns in an intelligent manner instead of the naive wall-following method of hugging the inside

Driver	Lap Time
Apex8	**1:39**
SmoothB2	**1:33**
NEAT	**1:31**
Felix16	**1:22**
Bulle2	**1:18**

Table 8.1: **Driving times for NEAT and handcoded drivers.** Handcoded drivers and a driver evolved by NEAT were timed on the "clkwis" track provided with RARS. The code for driver Apex8 was written by Maido Remm, Bulle2 by Marc Gueury, Felix16 by Doug Eleveld, and SmoothB2 by Dennis Lew., Each driver was timed on a single lap around the empty track. NEAT's time is on par with the best handcoded drivers.

of the turn. This surprising result shows that NEAT optimizes nontrivial behavior and discovers sophisticated techniques on its own: Such clever driving is hard even for human to learn. It allows quickly completing the course even though avoiding the inside of the turn extends the total trajectory.

The next section explains how crash predictors were evolved for an unstable driver chosen from early in evolution.

8.4 Evolving Open Road Crash Predictors

Instead of outputting driving control requests, the crash predictor outputs a *prediction* about whether and when a crash is going to happen. This prediction must be based on what the driver has been doing over some time leading up to the present. If the predictor has a good model of the driver's behavior, it can make realistic predictions about what the driver is likely to do in potentially dangerous situations.

Importantly, the networks evolve to determine on their own how many timesteps in the past to observe in order to make a prediction. Thus, they are not only learning based on the car's situation in the present. NEAT makes this task possible by automatically making this determination through evolving recurrent networks (other possible approaches will be discussed in Section 8.7).

The simplest kind of prediction is a binary output that decides whether or not a crash will happen in some fixed number of timesteps. While such a system is useful, a more sophisticated prediction can be made if the network also determines *when* it expects the crash. By predicting a time, the network is in effect constantly outputting a danger level, i.e. the sooner the predicted crash, the more dangerous the situation. Such a graded warning system would be more useful to human drivers, so NEAT was trained to make temporal predictions of this type.

The temporal prediction network was evolved as follows. The crash predictor network is given the same inputs as drivers. The network has three outputs that are combined to produce a

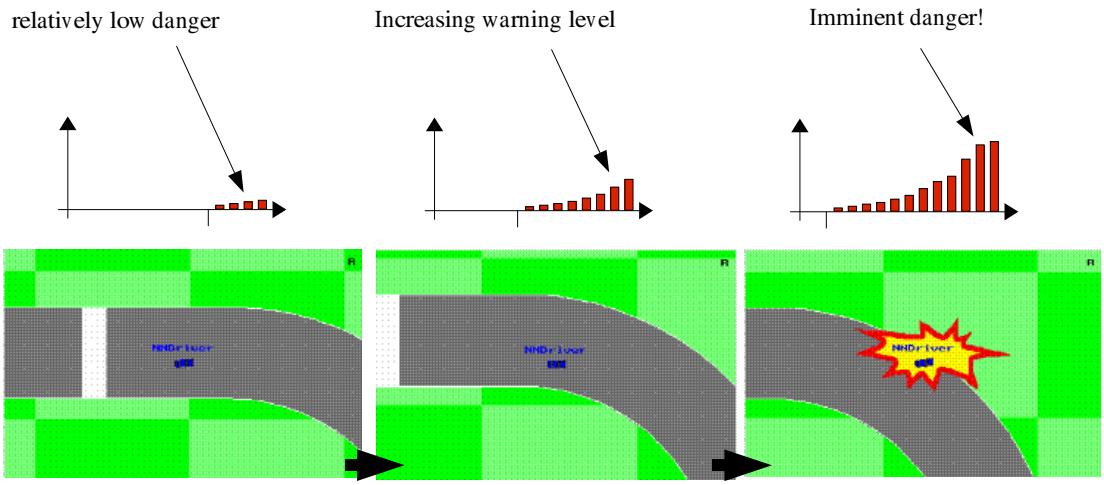


Figure 8.5: **The prediction queue foresees a crash.** The most recent warning appears on the right of the queue and the higher bars represent increasing urgency. As the car moves closer to crashing into the side of the road, the warning bars increase in size. NEAT was able to train networks that predict crashes in this way.

time, which is interpreted as predicted time to crash. The three outputs o_1, o_2, and o_3, are combined to produce a single time-to-crash T according to:

$$T = 3\frac{o_1}{o_{\min}} + 10\frac{o_2}{o_{\min}} + 17\frac{o_3}{o_{\min}}, \tag{8.2}$$

where $o_{\min}$ is the minimum output value. This method allows the network to interpolate predictions between 3 and 17 RARS timesteps in the future. Only outputs above 0.5 are included in the sum; any output below 0.5 is interpreted as 0 and discarded. The network can indicate that no crash is pending by outputting all three values below 0.5. At every timestep, the network output is pushed onto a prediction queue, which becomes a moving list of past predictions. Ideally, if a crash happens, the prediction queue has low-level warnings farther back in time and high-level warnings more recently (figure 8.5).

Fitness is computed by accumulating a total reward R_{tot} during evaluation. R_{tot} is the fitness of the entire evaluation including all crashes and predictions. During an evaluation, R_{tot} is modified at each time step in one of two cases: (1) If the car crashes, R_{tot} increases according to how close the actual prediction queue is to the ideal prediction queue. (2) If the car does not crash, a small bonus is added to R_{tot} if the oldest prediction in the queue was that no crash would occur.

Specifically, when a crash occurs, the prediction queue is compared with the the ideal queue to produce a reward. The queue holds 25 predictions q_1 to q_{25}. Upon crashing, each of the 25 predictions is compared to the ideal prediction I_l, where $1 \leq l \leq 25$, to produce a reward (figure 8.6):

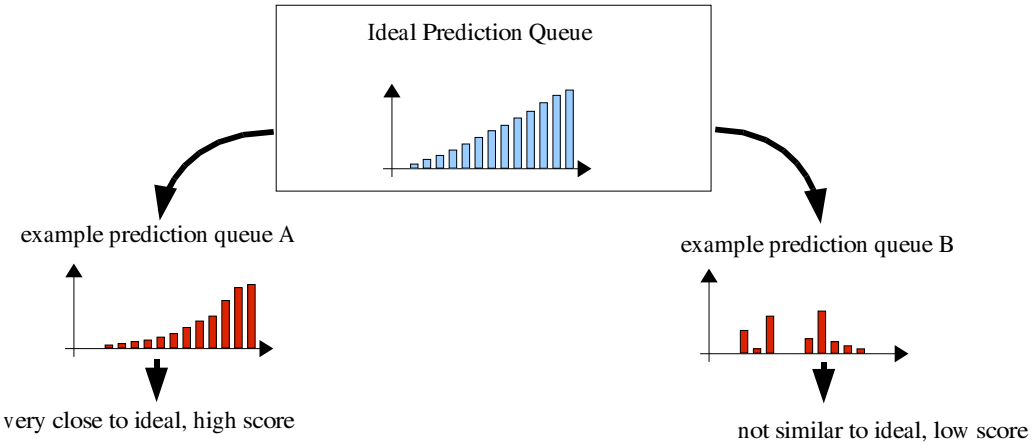


Figure 8.6: **Example prediction queues.** Since queue A is closer to the ideal than B, A would receive a higher reward. Comparing prediction queues in this way makes it possible to reward temporal predictions about when a crash is expected to occur.

$$R = \frac{\sum_{l=1}^{25} 25 - |q_l - I_l|}{25}. \tag{8.3}$$

R is then added to the accumulated reward R_{tot}. In addition, if the queue is over 25 timesteps long, the oldest prediction is dequeued. If the oldest prediction was that the car would not crash, a reward of 0.05 is added to R_{tot}. Otherwise, if it wrongly predicted a crash, the network does not receive a reward. At the end of the evaluation, R_{tot} is assigned as the network's fitness.

By using a prediction queue, NEAT was able to evolve networks that could vary their warning level, and evolution could determine on its own how far back in the past relevant information should be saved in recurrent connections. Figure 8.7 shows a successful warning network.

The main result is that NEAT evolved accurate predictors. In some cases, the warning network predicted crashes that could not be predicted only from the current state of the car. For example, when the car skids into the side of the road its heading is in a direction that could be interpreted as safe. Yet the evolved network still predicts a crash, showing that it is using memory to integrate a sequence of states into its prediction. The warning system can also be evaluated subjectively by having a human drive the car with the warning system on. Human drivers generally find the warnings accurate and helpful. Figure 8.8 shows actual prediction queues generated in real-time during human-controlled driving tests.

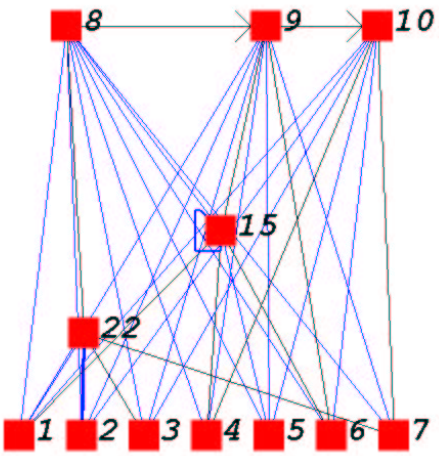
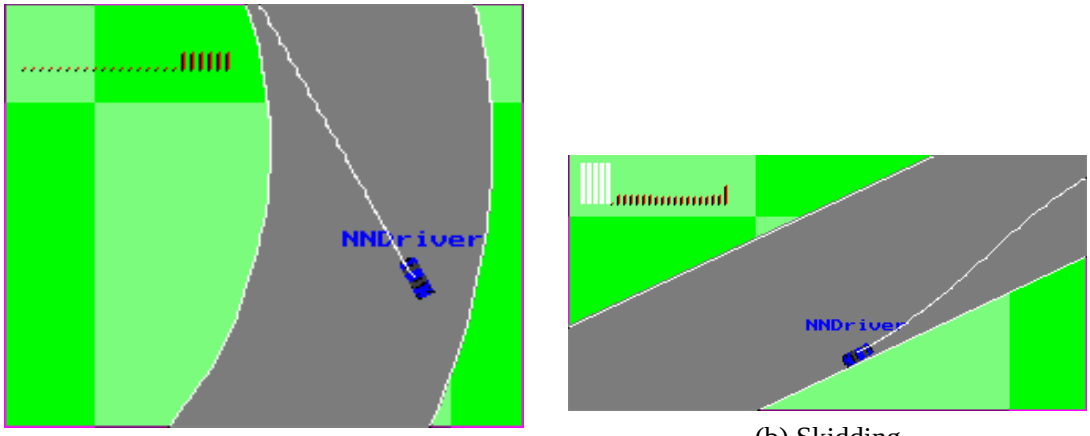


Figure 8.7: **Evolved open-road warning network.** This small network was able to make accurate predictions about crashing with minimal topology. The recurrent loop on node 15 and the connections between outputs give the network a rudimentary memory for events in the past. This memory allows it to predict crashes after events like skidding that can only be detected by considering the recent past positions of the car.



(a) Direct Crash

(b) Skidding

Figure 8.8: **Warning examples.** These screenshots were taken while a human was controlling the car. The real-time prediction queues are in the upper left corner of each box, with the most recent prediction on the right. White bars signify no warning, and the height of dark bars represents the seriousness of the warning. (a) As the car drives directly into the side of a turn, the warning switches from mild to severe. (b) Judging by the car's heading alone, there would be no reason to predict a crash in this scenario. However, as can be seen by the white trajectory line preceding the car, it has been skidding sideways for some time. The predictor network was only able to make the right warning by observing the trajectory of the car over time.

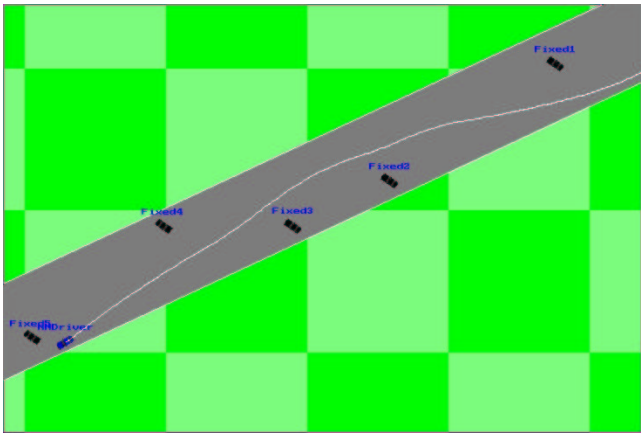


Figure 8.9: **Learning to avoid other cars.** The trajectory of the evolved driver is depicted as a white line on the road. The driver weaves skillfully around parked cars that are blocking the road. This driver can avoid randomly-placed fixed obstacles 100% of the time.

8.5 Driving with Other Cars

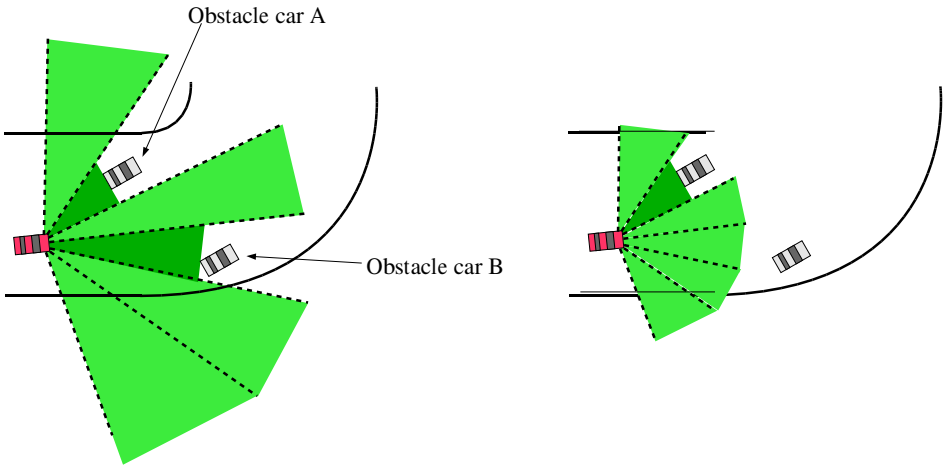
Adding other cars makes the task more realistic and complex. Before warning networks can be evolved, cars must be evolved that react to other vehicles on the road using vehicle radar sensors (figure 8.3). This task is more difficult than evolution on the open road because with other cars on the road it is difficult to ensure the driver will encounter other cars on every evaluation. If some networks encounter cars and some do not, evaluation would be noisy, which might interfere with evolution. One solution is to place stationary cars at several different positions so that the only way to avoid them is to drive around them. That way, evolution is forced to evolve controllers that can react to other vehicles. Avoiding stationary vehicles, or obstacles, is a special case of the general problem of avoiding other drivers.

NEAT was able to evolve drivers that avoid other cars in this manner (8.9). The driving network weaves around several obstacles like real cars avoiding cones in an obstacle course. Champion networks could avoid crashing 100% of the time over 100 test trials with cars placed at different random positions. The next section discusses how warning networks were evolved with stationary cars on the road.

8.6 Warning with Other Cars

Warning networks were evolved using the same prediction queue method as in Section 8.4, except now hitting another car was considered a crash causing a damage penalty.

A realistic prediction scenario can be set up by impairing the sensors of evolved driving networks (figure 8.10). That way, otherwise intelligent drivers occasionally crash because they



(a) Normal Radar Sensors (b) Shortsighted Radar Sensors

Figure 8.10: **Impairing radar sensors.** Impaired drivers were created by taking networks that drive well using normal radar sensors (a) and impairing them by cutting the range of the sensors by 70% (b). In (b), the car cannot see another car that it could easily see in (a) without the impairment. Shortsighted drivers are more likely to crash, making them useful for training crash predictors.

cannot see the obstacle in time to react. These impaired drivers are good test cases for prediction networks not only because they crash but also because they are similar to impaired human drivers. Such drivers may display patterns of behavior that can be learned by warning networks, allowing the warning system to partially compensate for the impairment.

The champion warning network evolved with such an impaired driver was able to predict 100% of crashes and was able to warn early in dangerous situations. Figure 8.11 shows how the warning network behaves when an impaired driver approaches a stationary car blocking the road. In some cases the warning network warns the driver before it begins to swerve. If the warning had been taken into account by the driver, it could have prevented the accident. If a similar situation occurred in the real world where a human driver became visually impaired, such advance warning could potentially save lives.

8.7 Discussion

The success of this experiment suggests that evolving warning networks for real cars may eventually be feasible. However, several challenges must be overcome to achieve this goal: (1) More realistic sensors, such as cameras or processed visual input, will be necessary to simulate real-world driving, (2) more complex scenarios such as traffic lights and cross traffic must be simulated, and (3) data from real-world cars must be used as a basis for how simulated sensors should work.

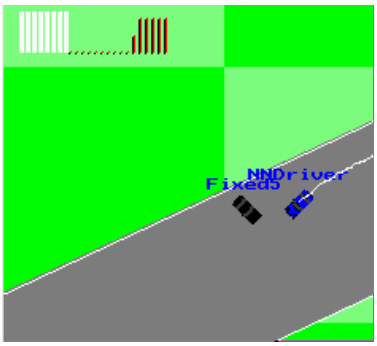


Figure 8.11: **Predicting a collision.** An impaired driver approaches a car obstructing its path. The prediction is displayed as in figure 8.8. As the driver moves dangerously closer to the obstacle, the warning network increases the urgency of its prediction. The driver begins to swerve at the last second, but the warning network is correct to warn at this point because it is dangerous to rely on swerving at the last possible moment, which does not always work. The extra time to react provided by the warning could potentially allow a driver to avoid crashing.

One of the difficulties in evaluating the performance of a warning network is that its success is somewhat subjective. Unlike in many classification tasks, a driver *does* want to be warned in situations that do not actually lead to crashes, because the point is to err on the side of caution. The question is how much erring is too much, i.e. when does warning become annoying. Answering this question may require psychological data, or perhaps drivers can answer the question for themselves and tune the strictness of the predictor to their liking. It was my subjective experience that the prediction networks evolved in these first experiments produced useful and sensible predictions, suggesting that striking the right balance is a feasible goal for the future.

Interestingly, NEAT can potentially strike such a balance by adjusting the relative cost of different kinds of errors in the fitness function. For example, the penalty for failing to warn before a crash can be made higher than for warning when there is no crash (since losing a life is more costly than a false alarm). This ability to weigh errors differently is a primary reason NEAT was chosen for this task. While recurrent supervised training techniques such as backpropagation on simple recurrent networks (SRNs; Elman 1991), recurrent backpropagation (Almeida 1987), or real-time recurrent learning (Williams and Zipser 1989) can potentially also learn temporal dependencies over variable periods of time, they cannot easily incorporate the relative cost of different kinds of errors into training. Supervised training methods also can be trapped local optima, and training recurrent networks is slow and unreliable compared to training feedforward networks (Bengio et al. 1994). Future work will compare NEAT to such methods in more detail.

8.8 Conclusion

Experiments with RARS show that NEAT can indeed evolve recurrent networks that predict crashes accurately. NEAT was also able to evolve highly effective drivers that could take the optimal trajectory through a turn. Evolved warning networks were able to predict crashes based on the previous history of the car, such as in skidding. Furthermore, they could compensate for sensory impairments.

This approach is a promising first step towards the ultimate goal of real-world warning systems. Warning about accidents before they happen has the potential to save lives, and is therefore a significant goal for neuroevolution.

Chapter 9

Application 3: The NERO Real-time Video Game

NERO is a pioneering neuroevolution-based video game. The game is built around a special real-time version of NEAT that allows players to interact with characters in the game *while they are evolving in real-time*. Real-time NEAT makes it possible for the player to take the role of a *trainer*, creating a new game genre. By demonstrating that such an application is possible, NERO opens up new opportunities for interactive machine learning in entertainment, education, and simulation. This chapter describes real-time NEAT and NERO, and reviews results from the first year of this ongoing project.

9.1 Motivation

Laird and van Lent (2000) suggested that interactive video games are an appropriate killer application for human-level AI. Video games carry perhaps the least risk to human life of any real-world application. The world video game market was between $15 billion and $20 billion in 2002, larger than even that of Hollywood (Thurrott 2002). Video games have become a facet of many people's lives and the market continues to expand. Yet machine learning is rarely attempted in commercial games. Thus, there is an unexplored opportunity to make video games more interesting and realistic, and to build entirely new genres. Such enhancements may have applications in education and training as well, changing the way people interact with their computers.

This chapter describes a novel video game concept built around a real-time version of NEAT that continually evolves improved behavior as the game is being played. The aim is to show that machine learning is indispensable for some kinds of video games to work, and how NEAT makes such an application possible.

In the video game industry, the term *Non-player-characters* (NPCs) refers to autonomous computer-controlled agents in the game. This chapter focuses on training NPCs as intelligent agents,

and the standard AI term *agents* is therefore used to refer to them. The behavior of such agents in current games is often repetitive and predictable. In most video games, simple scripts cannot learn or adapt to control the agents: Opponents will always make the same moves and the game quickly becomes boring. Machine learning could potentially keep video games interesting by allowing agents to change and adapt. However, a major problem with learning in video games is that if behavior is allowed to change, the game content becomes unpredictable. Agents might learn idiosyncratic behaviors or even not learn at all, making the gaming experience unsatisfying. One way to avoid this problem is to train agents offline, and then freeze the results into the final game. However, if behaviors are frozen before the game is released, agents cannot adapt and change in response to the tactics of particular players, which is the real goal of machine learning techniques.

If agents are to adapt and change in real-time, a powerful and reliable machine learning method is needed. It turns out that NEAT can be enhanced to work in real-time while a game is being played. In order to test real-time NEAT (rtNEAT) in a video game, the Digital Media Collaboratory (DMC) at the University of Texas at Austin initiated the NeuroEvolving Robotic Operatives (NERO) project in October of 2003 (`http://dev.eltlabs.org/nero_public`). This project is based on a proposal for a game based on rtNEAT developed at the *2nd Annual Game Development Workshop on Artificial Intelligence, Interactivity, and Immersive Environments* in Austin, TX (presentation by Kenneth Stanley, 2003). The idea was to create a game in which learning is *indispensable*, in other words, without learning NERO could not exist as a game. Thus, NERO is a powerful demonstration of how machine learning can open up new possibilities in gaming and allow agents to adapt.

The chapter begins with an explanation of rtNEAT, followed by a description of NERO and an overview of the current status and performance of the game.

9.2 Real-time NEAT (rtNEAT)

Real-time neuroevolution is based on the observation that in a video game, the entire population of agents plays *at the same time*. Therefore, unlike in offline genetic algorithms, agent fitness statistics are constantly collected as the game is played, and the agents are evolved continuously. The question is when the agents can be replaced with new ones so offspring can be evaluated.

Replacing the entire population together on each generation would look incongruous since everyone's behavior would change at once. In addition, behaviors would remain static during the large gaps of time between generations. Instead, in rtNEAT, a single individual is replaced every few game ticks (as in e.g. (m,1)-ES; Beyer and Paul Schwefel 2002). One of the worst individuals is removed and replaced with a child of parents chosen from among the best. This cycle of removal and replacement happens continually throughout the game (figure 9.1).

Real-time evolution was first implemented using conventional neuroevolution (Agogino et al. 2000) before NEAT was developed. However, conventional neuroevolution is not sufficiently

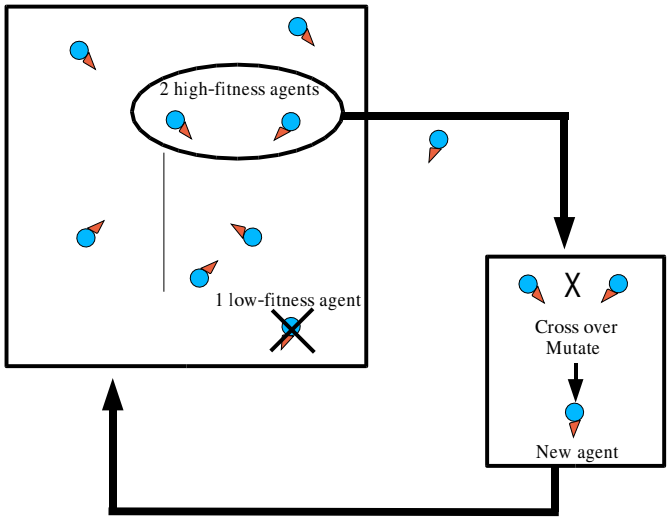


Figure 9.1: **The main replacement cycle in rtNEAT.** Robot game agents (represented as small circles) are depicted playing a game in the large box. Every few ticks, two high-fitness robots are selected to produce an offspring that replace another of lower fitness. This cycle of replacement operates continually throughout the game, creating a constant turnover of new behaviors.

powerful to meet the demands of modern video games. In contrast, a real-time version of NEAT offers the advantages of NEAT: Agent neural networks can become increasingly sophisticated and complex during gameplay. The challenge is to preserve the usual dynamics of NEAT, namely protection of innovation through speciation and complexification, even as evolution runs in real-time. While generational NEAT normally assigns offspring to species *en masse* for each new generation, rtNEAT cannot allocate space for an entire species at once since it only produces one new offspring at a time. Therefore, a new reproduction cycle must be introduced to allow rtNEAT to speciate in real-time with the same results.

The main loop in rtNEAT works as follows. Let f_i be the fitness of organism i. Recall that fitness sharing adjusts it to $\frac{f_i}{|S|}$, where $|S|$ is the number of individuals in the species (Section 3.3). In other words, fitness is reduced proportionally to the size of the species. This adjustment is important because selection in rtNEAT must be based on adjusted fitness rather than original fitness in order to maintain the same dynamics as NEAT. In addition, because the number of offspring assigned to a species in NEAT is based on its average fitness $\overline{F}$, this average must always be kept up-to-date. Thus, after every n ticks of the game clock, rtNEAT performs the following operations:

1. Remove the agent with the worst *adjusted* fitness from the population assuming one has been alive sufficiently long so that it has been properly evaluated.

2. Re-estimate $\overline{F}$ for all species
3. Choose a parent species to create the new offspring
4. Adjust C_t dynamically and *reassign* all agents to species
5. Place the new agent in the world

Each of these steps is discussed in more detail below.

9.2.1 Step 1: Removing the worst agent

The goal of this step is to remove a poorly performing agent from the game, hopefully to be replaced by something better. The agent must be chosen carefully to preserve speciation dynamics. If the agent with the worst *unadjusted* fitness were chosen, fitness sharing could no longer protect innovation because new topologies would be removed as soon as they appear. Thus, the agent with the worst *adjusted* fitness should be removed, since adjusted fitness takes into account species size, so that new smaller species are not removed as soon as they appear.

It is also important not to remove agents that are too young. In generational NEAT, *age* is not considered since networks are generally all evaluated for the same amount of time. However, in rtNEAT, new agents are constantly being born, meaning different agents have been around for different lengths of time. It would be dangerous to remove agents that are too young because they have not played for long enough to accurately assess their fitness. Therefore, rtNEAT only removes agents who have played for more than the minimum amount of time m.

9.2.2 Step 2: Re-estimating $\overline{F}$

Assuming there was an agent old enough to be removed, its species now has one less member and therefore its average fitness $\overline{F}$ has likely changed. It is important to keep $\overline{F}$ up-to-date because $\overline{F}$ is used in choosing the parent species in the next step. Therefore, rtNEAT needs to re-estimate $\overline{F}$.

9.2.3 Step 3: Choosing the parent species

In generational NEAT the number of offspring n_k assigned to species k is $\frac{\overline{F_k}}{\overline{F}_{\text{tot}}}|P|$, where $\overline{F_k}$ is the average fitness of species k, $\overline{F}_{\text{tot}}$ is the sum of all the average species' fitnesses, and $|P|$ is the population size (equation 3.3).

This behavior needs to be approximated in rtNEAT even though n_k cannot be assigned explicitly (since only one offspring is created at a time). Given that n_k is proportional to $\overline{F}$, the parent species can be chosen probabilistically using the same relationship:

$$Pr(S_k) = \frac{\overline{F_k}}{\overline{F}_{\text{tot}}}. \tag{9.1}$$

The probability of choosing a given parent species is proportional to its average fitness compared to the total of all species' average fitnesses. Thus, over the long run, the expected number of offspring for each species is proportional to n_k, preserving the speciation dynamics of generational NEAT.

9.2.4 Step 4: Dynamic Compatibility Thresholding in Real time

Recall from section 3.3 that networks are placed into a species in generational NEAT if their compatibility distance from the species' representative is less than the threshold C_t. Section 3.3 suggested that one way to avoid the burden of choosing the appropriate C_t is to instead choose a target number of species and let NEAT adjust C_t dynamically to reach the target. If there are too many species, C_t can be raised to be more inclusive; if there are too few, C_t can be lowered to be stricter.

An advantage of this kind of *dynamic compatibility thresholding* is that it keeps the number of species relatively stable. Such stability is particularly important in a real-time video game since the population may need to be small to accommodate CPU resources dedicated to graphical processing, and therefore a sudden explosion in the number of species would be undesirable.

In generational NEAT, C_t can be adjusted before the next generation is created, but in rtNEAT changing C_t alone is not sufficient because most of the population simply remains where they are. Just changing a variable does not cause anything to move to a different species. Therefore, after changing C_t in rtNEAT, the entire population must be reassigned to the existing species based on the new C_t. As in generational NEAT, if a network does not belong in any species a new species is created with that network as its representative.[1]

9.2.5 Step 5: Replacing the old agent with the new one

Since an individual was removed in step 1, the new offspring needs to replace it. How agents are replaced depends on the game. In some games, the neural network can be removed from a body and replaced without doing anything to the body. In others, the body may have died and need to be replaced as well. rtNEAT can work with any of these schemes as long as an old neural network gets replaced with a new one.

Step 5 concludes the steps necessary to approximate generational NEAT in real-time. However, there is one remaining issue: The entire loop should be performed at regular intervals, every n ticks, but how should n be chosen?

9.2.6 Determining the Number of Ticks Between Replacements

If agents are replaced too frequently, they do not live long enough to reach the minimum time m to be evaluated. For example, imagine that it takes 100 ticks to obtain an accurate performance

[1]Depending on the specific game, C_t does not necessarily need to be adjusted and species reorganized as often as every replacement. The number of ticks between adjustments is chosen by the game designer.

evaluation, but that an individual is replaced in a population of 50 on every tick. No one ever lives long enough to be evaluated and the population always consists of only new agents. On the other hand, if agents are replaced too infrequently, evolution slows down to a pace that the player no longer enjoys.

Interestingly, the appropriate frequency can be determined through a principled approach. Let I be the fraction of the population that is too young and therefore cannot be replaced. As before, n is the ticks between replacements, m is the minimum time alive, and $|P|$ is the population size. A *law of eligibility* can be formulated that specifies what fraction of the population can be expected to be ineligible once evolution reaches a steady state (i.e. after the first few time steps when no one is eligible):

$$I = \frac{m}{|P|n}. \tag{9.2}$$

According to equation 9.2, the larger the population and the more time between replacements, the lower the fraction of ineligible agents. This principle makes sense since in a larger population it takes more time to replace the entire population. Also, the more time passes between replacements, the more time the population has to age, and hence fewer are ineligible. On the other hand, the larger the minimum age, the more agents are ineligible because more time is necessary to become eligible.

It is also helpful to think of $\frac{m}{n}$ as the *number* of individuals that must be ineligible at any time; over the course of m ticks, an agent is replaced every n ticks, and all the new agents that appear over m ticks will remain ineligible for that duration since they cannot have been around for over m ticks. For example, if $|P|$ is 50, M is 500, and n is 20, 50% of the population would be ineligible.

Based on the law of eligibility, rtNEAT can decide on its own how many ticks n should lapse between replacements for a preferred level of ineligibility, specific population size, and minimum time between replacements:

$$n = \frac{m}{|P|I}. \tag{9.3}$$

It is best to let the user choose I because in general it is most critical to performance; if too much of the population is ineligible at one time, the mating pool is not sufficiently large. Equation 9.3 allows rtNEAT to determine the correct number of ticks between replacements n to maintain a desired eligibility level. In NERO, 50% of the population remains eligible using this technique.

By performing the right operations every n ticks, choosing the right individual to replace and replacing it with an offspring of a carefully chosen species, rtNEAT is able to replicate the dynamics of NEAT in real-time. Thus, it is now possible to deploy NEAT in a real video game and interact with complexifying agents as they evolve. The next section describes such a game.

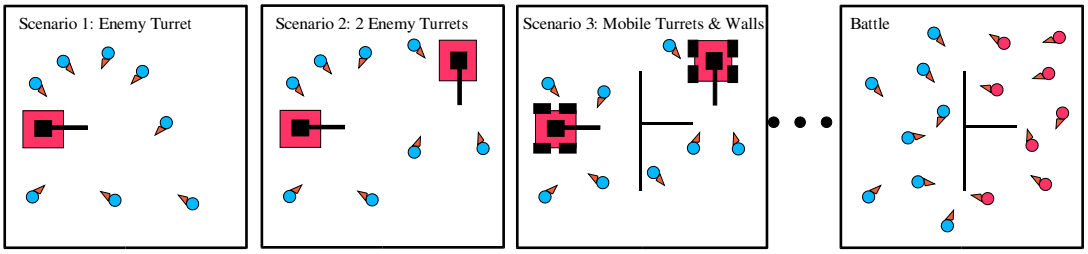


Figure 9.2: **A turret training sequence (*color figure*).** The figure depicts a sequence of increasingly difficult and complicated training exercises in which the agents attempt to attack turrets without getting hit. In the first exercise there is only a single turret but more turrets are added by the player as the team improves. Eventually walls are added and the turrets are given wheels so they can move. Finally, after the team has mastered the hardest exercise, it is deployed in a real battle against another team.

9.3 NeuroEvolving Robotic Operatives (NERO)

NERO is representative of a new video game genre that is only possible through machine learning. The idea is to put the player in the role of a *trainer* or a *drill instructor* who teaches a team of agents by designing a curriculum. Of course, for the player to be able to teach agents, the agents must be able to *learn*; rtNEAT is the learning algorithm that makes NERO possible.

In NERO, the learning agents are simulated robots, and the goal is to train a team of robots for military combat. The robots begin the game with no skills and only the ability to learn. In order to prepare for combat, the player must design a sequence of training exercises and goals. Ideally, the exercises are increasingly difficult so that the team can begin by learning a foundation of basic skills and then gradually building on them (figure 9.2). When the player is satisfied that the team is prepared, the team is deployed in a battle against another team trained by another player (possibly on the internet), making for a captivating and exciting culmination of training. The challenge is to anticipate the kinds of skills that might be necessary for battle and build training exercises to hone those skills. The next two sections explain how the agents are trained in NERO and how they fight an opposing team in battle.

9.3.1 Training Mode

The player sets up training exercises by placing objects on the field and specifying goals through several sliders (figure 9.3). The objects include static enemies, enemy turrets, rovers (i.e. turrets that move), and walls. To the player, the sliders serve as an interface for describing ideal behavior. To rtNEAT, they represent coefficients for fitness components. For example, the sliders specify how much to reward or punish approaching enemies, hitting targets, getting hit, following friends, dispersing, etc. Fitness is computed as the sum of all these components multiplied by their slider

Figure 9.3: **Setting up training scenarios.** This screenshot shows items the player can place on the field and sliders used to control behavior. The red robot is a stationary enemy turret that turns back and forth as it shoots repetitively. Behind the turret is a wall. The player can place turrets, other kinds of enemies, and walls anywhere on the training field. On the right is the box containing slider controls. These sliders specify the player's preference for the behavior the team should try to optimize. For example the "E" icon means "approach enemy," and the red bar specifies that the player wants to punish robots that approach the enemy. The crosshair icon represents "hit target," which is being rewarded. The sliders represent fitness components that are used by rtNEAT. The value of the slider is used by rtNEAT as the coefficient of the corresponding fitness component. Through placing items on the field and setting sliders, the player creates training scenarios where learning takes place.

levels, which can be positive or negative. Thus, the player has a natural interface for setting up a training exercise and specifying desired behavior.

Robots have several types of sensors. Although NERO programmers frequently experiment with new sensor configurations, the standard sensors include enemy radars, an "on target" sensor, object rangefinders, and line-of-fire sensors. Figure 9.4 shows a neural network with the standard set of sensors and outputs, and figure 9.5 describes how the sensors function.

Training mode is designed to allow the player to set up a training scenario on the field where the robots can continually be evaluated while the worst robot's neural network is replaced every few ticks. Thus, training must provide a standard way for robots to appear on the field in such a way that every robot has an equal chance to prove its worth. To meet this goal, the robots spawn from a designated area of the field called the *factory*. Each robot is allowed a limited time on the field during which its fitness is assessed. When their time on the field expires, robots are transported back to the factory, where they begin another evaluation. Neural networks are only replaced in robots that have been put back in the factory. The factory ensures that a new neural network cannot get lucky by appearing in a robot that happens to be standing in an advantageous position: All evaluations begin consistently in the factory. In addition, the fitness of robots that survive more than one deployment on the field is updated through a diminishing average that gradually forgets deployments from the distant past. Thus, older robots have more reliable fitness measures since they are averaged over more deployments than younger robots, but their fitness does not become out of date.

The diminishing average fitness is obtained by first computing an average over the first few trials and then maintaining a continuous leaky average. The fitness update rule is,

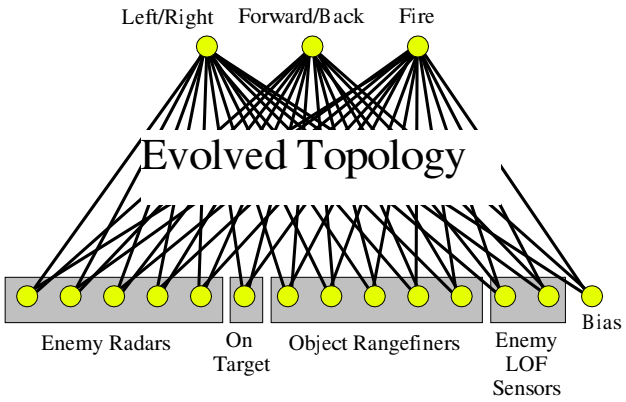


Figure 9.4: **NERO input sensors and action outputs.** Each NERO robot can see enemies, determine whether an enemy is currently in its line of fire, detect objects and walls, and see the direction the enemy is firing. Its outputs specify the direction of movement and whether or not to fire. This configuration has been used to evolve varied and complex behaviors; other variations work as well and the standard set of sensors can easily be changed.

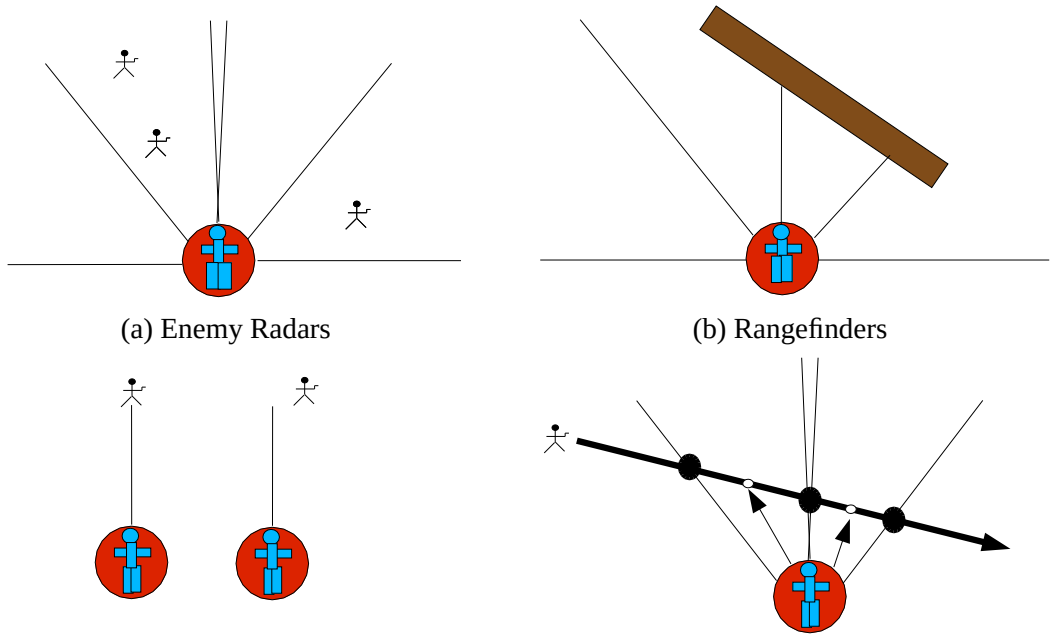
$$f_{t+1} = f_t + \frac{s_t - f_t}{r} \tag{9.4}$$

where f_t is the current fitness, s_t is the score from the current evaluation, and r controls the rate of forgetting. The lower r is set, the sooner recent evaluations are forgotten. This method ensures that fitness statistics do not become out of date even for older networks.

Training begins by deploying 50 robots on the field. Each robot is controlled by a neural network with random connection weights and no hidden nodes, as is the usual starting configuration for NEAT (see appendix A for a complete description of the rtNEAT parameters used in NERO). As the neural networks are replaced in real-time, behavior improves dramatically, and robots eventually learn to perform the task the player sets up. When the player decides that performance has reached a satisfactory level, he or she can save the team in a file. Saved teams can be reloaded for further training in different scenarios, or they can be loaded into battle mode. In battle, they face off against teams trained by an opponent player, as will be described next.

9.3.2 Battle Mode

In battle mode, the player discovers how training paid off. A battle team of 20 robots is assembled from as many different training teams as desired. For example, perhaps some robots were trained for close combat while others were trained to stay far away and avoid fire. A player may choose to compose a heterogeneous team from both training sessions.



(a) Enemy Radars (b) Rangefinders

(c) On-Target Sensor (d) Line-of-fire sensors

Figure 9.5: **NERO sensor design.** All NERO sensors are egocentric, i.e. they tell where the objects are from the robot's perspective. (a) Several enemy radar sensors divide the 360 degrees around the robot into slices. Each slice activates a sensor in proportion to how close an enemy is within that slice. If there is more than one enemy in a single slice, their activations are summed. (b) Rangefinders project rays at several angles from the robot. The distance the ray travels before it hits an object is returned as the value of the sensor. Rangefinders are useful for detecting long contiguous objects whereas radars are appropriate for relatively small, discrete objects. (c) The on-target sensor returns full activation only if a ray projected along the front heading of the robot hits an enemy. This sensor tells the robot whether it should attempt to shoot. (d) The line of fire sensors detect where a bullet stream from the closest enemy is heading. Thus, these sensors can be used to avoid fire. They work by computing where the line of fire intersects rays projecting from the robot, giving a sense of the bullet's path. These sensors provide sufficient information for robots to learn successful behaviors for battle.

Battle mode is designed to run over a server so that two players can watch the battle from separate terminals on the internet. The battle begins with the two teams arrayed on opposite sides of the field. When one player presses a "go" button, the neural networks obtain control of their robots and perform according to their training. Unlike in training, where being shot does not lead to a robot body being damaged, the robots are actually destroyed after being shot several times in battle. The battle ends when one team is completely eliminated. In some cases, the only surviving robots may insist on avoiding each other, in which case action ceases before one side is completely destroyed. In that case, the winner is the team with the most robots left standing.

The basic battlefield configuration is an empty pen surrounded by four bounding walls, although it is possible to compete on a more complex field, with walls or other obstacles (figure

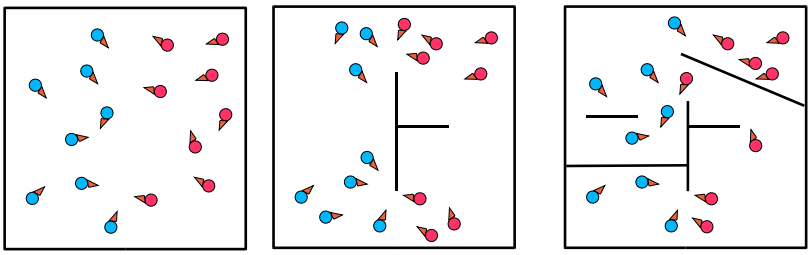


Figure 9.6: **Battlefield configurations (*color figure*).** The figure shows a range of possible configurations from an open pen to a maze-like environment. Players can construct their own battlefield configurations and train for them. The basic configuration, which is used in section 9.4, is the empty pen surrounded by four bounding walls.

9.6). Players train their robots and assemble teams for the particular battlefield configuration on which they intend to play. In the experiments described in this chapter, the battlefield was the basic pen.

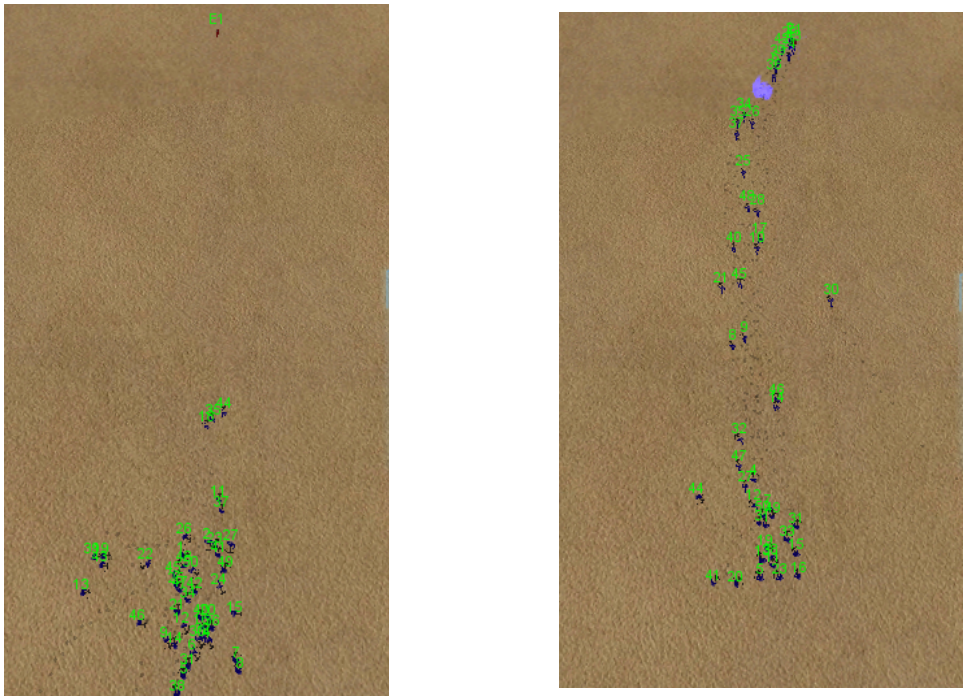
The next section gives examples of actual NERO training and battle sessions.

9.4 Playing NERO

Behavior can be evolved very quickly in NERO, fast enough so that the player can be watching and interacting with the system in real time. The game engine Torque, licensed from GarageGames, drives NERO's simulated physics and graphics. An important property of the Torque engine is that its physics simulation is slightly nondeterministic, so that the same game is never played twice. In addition, Torque makes it possible for the player to take control of enemy robots using a joystick, an option that can be useful in training.

The first playable version of NERO was completed in May of 2004. At that time, several NERO programmers trained their own teams and held a tournament. As examples of what is possible in NERO, this section outlines the behaviors evolved for the tournament, the resulting battles, and the real-time performance of NERO and rtNEAT.

NERO is capable of evolving behaviors very quickly in real-time. The most basic battle tactic is to aggressively seek the enemy and fire. To train for this tactic, a single static enemy was placed on the training field, and robots were rewarded for approaching the enemy. This training required robots to learn to run towards a target, which is difficult since robots start out in the factory facing in random directions. Starting from random neural networks, it takes on average 99.7 seconds for 90% of the robots on the field learn to approach the enemy successfully (10 runs, $sd = 44.5$s) It is important to note that the success criterion, i.e. that the team sufficiently learns to approach the enemy, is in part subjective since the player decides when training is complete by visually assessing



(a) Five seconds: Mass confusion

(b) 100 seconds: Success

Figure 9.7: **Learning to approach the enemy (*color figure*).** These screenshots show the training field before and after the robots evolved seeking behavior. The factory is at the bottom of each panel and the enemy being sought is at the top. The numbers above the robots' heads are used to identify individual robots. (a) Five seconds after the training begins, the robots scatter haphazardly around the factory, unable to effectively seek the enemy. (b) After ninety seconds, the robots consistently seek the enemy. Some robots prefer swinging left, while others swing right. These pictures demonstrate that behavior improves dramatically in real-time over only 100 seconds.

the team's performance. Nevertheless, success in seeking is generally unambiguous as shown in figure 9.7.

NERO differs from most applications of GAs in that the quality of evolution is judged from the player's perspective based on the performance of the *entire* population. On the other hand GA practitioners generally only look at the champions of a run. However, even though the entire population must solve the task, it does not converge to the same solution. In seek training, some robots evolve a tendency to run slightly to the left of the target, while others run to the right. The population diverges because the 50 agents interact as they move simultaneously on the field at the same time. If all the robots chose exactly the same path, they would often crash into each other and slow each other down, so naturally some robots take slightly different paths to the goal. In other words, NERO is actually a massively parallel coevolving ecology in which the entire population is evaluated together.

Figure 9.8: **Running away backwards.** This training screenshot shows several robots backed up against the wall after running backwards and shooting at the enemy, which is being controlled from a first-person perspective by a human trainer using a joystick. Robots learned to run away from the enemy backwards during avoidance training because that way they can shoot as they flee. Running away backwards is an example of evolution's ability to find novel and effective behaviors.

After the robots learned to seek the enemy, they were further trained to fire at the enemy. It is possible to train robots to aim by rewarding them for hitting a target, but it is also aesthetically unpleasing to players to have to wait while robots fire haphazardly in all directions and slowly figure out how to aim. Therefore, the fire output of neural networks was connected to an aiming script that points the gun properly at the enemy closest to the robot's current heading within some fixed distance. Thus, robots quickly learn to seek and attack the enemy.

Robots were also trained to avoid the enemy. In fact, rtNEAT was flexible enough to *devolve* a population that had converged on seeking behavior into a completely opposite, avoidance, behavior. For avoidance training, players controlled an enemy robot with a joystick and ran it towards robots on the field. The robots learned to back away in order to avoid being penalized for being too near the enemy. Interestingly, robots preferred to run away from the enemy backwards because that way they could still shoot the enemy. Also, most of their enemy radars are on their front half, giving them better resolution if they remain facing their target (figure 9.8).

By placing a turret on the field and asking robots to approach the turret without getting hit, robots were able to learn to avoid enemy fire (figure 9.9). The turret is programmed to periodically rotate back and forth spraying bullets. Robots evolved to run to the opposite side of the turret from the spray and approach it from behind, a tactic that is promising for battle.

Other interesting behaviors were evolved to test the limits of rtNEAT rather than specifically prepare the troops for battle. For example, robots were trained to run around walls in order to approach the enemy. As performance improved, players incrementally added more walls until the robots could navigate an entire maze without any path-planning (figure 9.10)! Interestingly, different

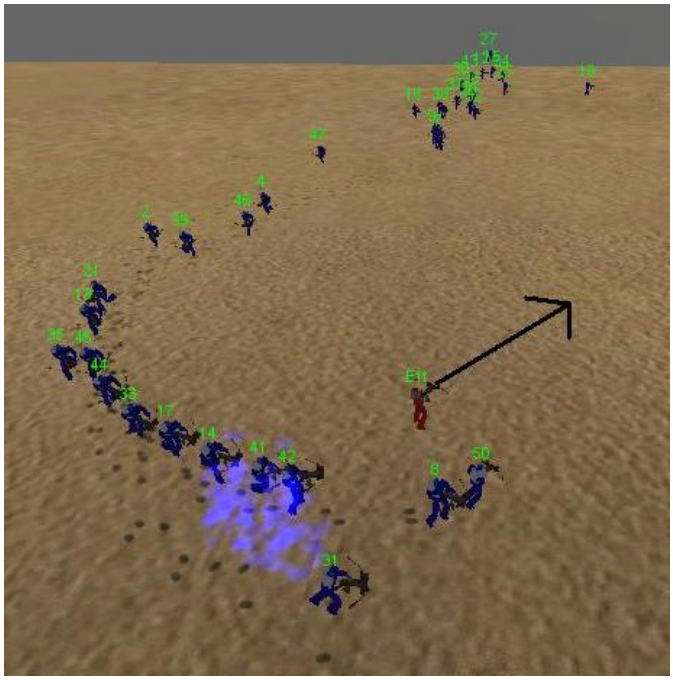


Figure 9.9: **Avoiding turret fire (*color figure*).** The black arrow points in the current direction of the turret fire (the arrow is not part of the NERO display and is only added for illustration). The turret is depicted as a red robot. The blue robots in training learn to run safely around the enemy's line of fire in order to attack. Notice how they loop around the back of the turret and attack from behind. When the turret moves, the robots change their attack trajectory accordingly. Learning to avoid fire is an important battle skill. The conclusion is that rtNEAT was able to evolve sophisticated, nontrivial behavior in real time.

species evolved to take different paths through the maze, showing that topology and function are correlated in rtNEAT, and confirming the success of real-time speciation.

In battle, some teams that were trained differently were nevertheless evenly matched, while some training types consistently prevailed against others For example, an aggressive seeking team from the tournament had only a slight advantage over an avoidant team, winning six out of ten battles, losing three, and tying one (Table 9.1). The avoidant team runs in a pack to a corner of the field's enclosing wall (figure 9.11). Sometimes, if they make it to the corner and assemble fast enough, the aggressive team runs into an ambush and is obliterated. However, slightly more often the aggressive team gets a few shots in before the avoidant team can gather in the corner. In that case, the aggressive team traps the avoidant team with greater surviving numbers. The conclusion is that seeking and running away are fairly well-balanced tactics, neither providing a significant advantage over the other. The interesting challenge of NERO is to conceive strategies that are clearly dominant over others.

One of the best teams was trained by observing a phenomenon that happened consistently in battle. Chases among robots from opposing teams frequently caused robots to eventually reach the

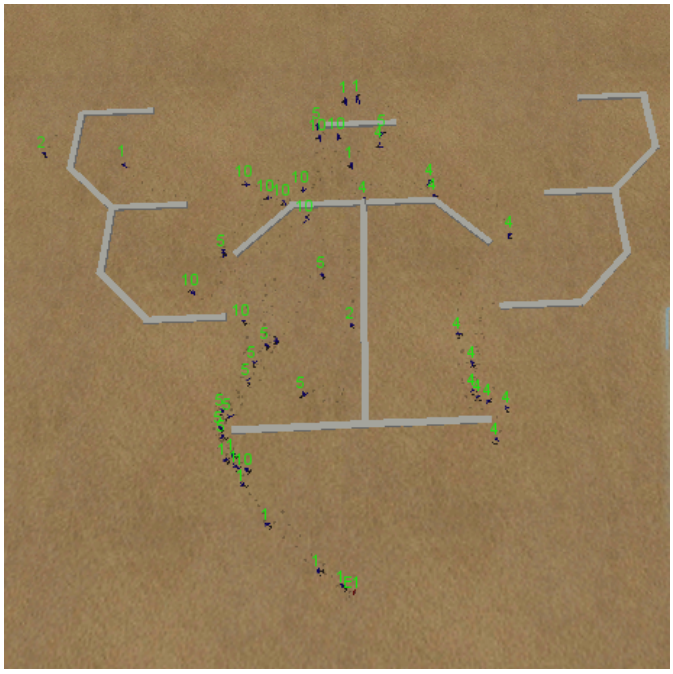


Figure 9.10: **Navigating a maze (*color figure*).** Incremental training on increasingly complex wall configurations produced robots that could navigate this maze to find the enemy. The robots spawn from the factory at the top of the maze and proceed down to the enemy at the bottom. In this picture, the green numbers above the robots specify their species. Notice that species "4" evolved to take the path through the right side of the maze while other species take the the left path. This result suggests that protecting innovation in rtNEAT indeed supports a range of diverse behaviors, each with its own network topology.

Battle Number	Seekers	Avoiders
1	**6**	0
2	4	**7**
3	**8**	0
4	7	7
5	**8**	3
6	6	**10**
7	**5**	4
8	**5**	2
9	3	**7**
10	**8**	0

Table 9.1: **Seekers vs. Avoiders.** Scores from 10 battles are shown between a team trained to aggressively seek and attack the enemy and another team taught to run away backwards and shoot at the same time. The seeking team wins six out of the 10 games, but its advantage is not significant, showing that when strategies contrast they can still be evenly matched. Results like this one can be unexpected, teaching players about the relative strengths and weakness of different tactics.

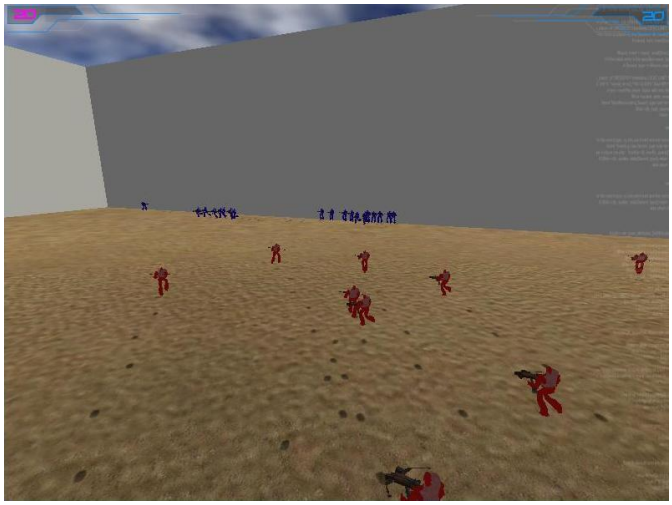


Figure 9.11: **Seekers chasing avoiders in battle (*color figure*).** In this battle screenshot, red robots trained to seek and attack the enemy pursue blue robots that have backed up against the wall. Teams trained for different tactics are clearly discernable in battle, demonstrating the ability of the training to evolve diverse tactics.

Battle Number	Wall-fighters	Seekers
1	**7**	2
2	**9**	0
3	**4**	3
4	**7**	2
5	**10**	0
6	**8**	2
7	**12**	2
8	**7**	2
9	**4**	2
10	**9**	1

Table 9.2: **Wall-fighters vs. Seekers.** The table shows final scores from 10 battles between a team trained to fight near walls and another trained to aggressively seek and attack the enemy. The wall-fighters win every battle because they know how to avoid fire near a wall, while the aggressive team runs directly into fire when fighting near a wall. The total superiority of the wall-fighters shows that the right tactical training indeed matters in battle, and that rtNEAT was able to evolve sophisticated fighting tactics.

field's bounding walls. Particularly for robots trained to avoid turret fire by attacking from behind (figure 9.9), enemies standing against the wall present a serious problem since it is not possible to go around them. Thus, training a team against a turret with its back against the wall, it was possible to familiarize robots with attacking enemies against a wall. This team learned to hover near the turret and fire when it turned away, but back off quickly when it turned towards them. This tactic works effectively when several friendly robots from the same team are nearby since an enemy can only be facing one direction at a time. In fact, the wall-based team won the first NERO tournament by using this strategy. Table 9.2 shows that the wall-trained team wins 100% of the time against the aggressive seeking team. Thus, it is possible to learn sophisticated tactics that dominate over simpler ones like seek or avoid.

9.5 Discussion

Participants in the first NERO tournament agreed that the game was engrossing and entertaining. Battles were exciting events for all the participants, evoking plentiful clapping and cheering. Players spent many hours honing behaviors and assembling teams with just the right combination of tactics.

An important point of this project is that NERO would not be possible without rtNEAT. rtNEAT was able to evolve interesting tactics quickly in real-time while players interacted with NERO, showing that neuroevolution can be deployed in a real game and work fast enough to provide entertaining results.

The success of the first NERO prototype suggests that the rtNEAT technology has immediate potential commercial applications in modern games. Any game in which agent behavior is repetitive and boring can be improved by allowing rtNEAT to at least partially modify tactics in real-time. Especially in persistent video games such as Massive Multiplayer Online Games (MMOGs) that last for months or years, the potential for rtNEAT to continually adapt and optimize agent behavior may permanently alter the gaming experience for millions of players around the world.

Since the first tournament took place, new features have been added to NERO, increasing its appeal and complexity. For example, robots can now duck behind walls and learn to run to a flag placed by the player to designate important areas of the field. The game continues to be developed and new features and sensors are constantly being added. The goal is to have a full network-playable version with an easy and intuitive user interface in the near future.

An important issue for the future is how to assess results in a game in which behavior is largely subjective. One possible approach is to train benchmark teams and measure the success of future training against those benchmarks. This idea and others will be employed in testing as the project matures and standard strategies are identified. At present, the project's main contribution is to show that an entirely new genre of game is possible because of rtNEAT.

NERO is also being used as a common platform for quickly implementing complicated real-time neuroevolution experiments. While video games are intended mainly for entertainment,

they are an excellent catalyst for improving machine learning technology. Because of the gaming industry's financial success and low physical risk, it makes sense to explore this area as a stepping stone to other more critical applications. With this new technology, it may finally be possible to use games for training as has long been envisioned. As humans improve in such training games, so could surrounding agents, keeping the simulation realistic for longer than has been possible in the past.

9.6 Conclusion

A real-time version of NEAT (rtNEAT) was developed to allow users to interact with evolving agents. In rtNEAT, an entire population is simultaneously and asynchronously evaluated as it evolves. Using this method, it was possible to build an entirely new kind of video game, NERO, where the characters adapt in real time in response to the player's actions. In NERO, the player takes the role of a trainer and constructs training scenarios for a team of simulated robots. The rtNEAT technique can form the basis for other similar interactive learning applications in the future, and eventually even make it possible to use gaming as a method for training people in sophisticated tasks.

Chapter 10

Discussion and Future Work

NEAT is a principled approach to evolving neural network topologies and weights. This chapter begins with a general discussion of NEAT's approach to evolving topologies and then examines how network behavior becomes more sophisticated through complexification. The remaining sections overview three primary areas of future work: (1) Using NEAT to evolve solutions other than neural networks, (2) expanding NEAT's neural model, and (3) using NEAT with indirect encoding.

10.1 Evolving Neural Network Topologies

NEAT presents several advances in the evolution of neural networks. Through historical markings, NEAT is able to combine different topologies without topological analysis. NEAT also demonstrates that a meaningful metric for comparing and clustering similar networks can be derived from historical information in the population.

A parallel can be drawn between structure evolution in NEAT and incremental evolution (Gomez and Miikkulainen 1997; Wieland 1991). Incremental evolution is a method used to train a system to solve harder tasks than it normally could by training it on incrementally more challenging tasks. NE is likely to get stuck on a local optimum when attempting to solve the harder task directly. However, after solving the easier version of the task first, the population is likely to be in a part of fitness space closer to the solution to the harder task, allowing it to avoid local optima. Adding structure to a solution is analogous to taking a solution to an easy task as the starting point for evolving the solution to a harder task. The network structure before the addition is optimized in a lower-dimensional space. When structure is added, the network increments into a more complex space where it is already close to the solution. The difference between the incrementality of adding structure and general incremental evolution is that adding structure is *automatic* in NEAT whereas a sequence of progressively harder tasks requires human design.

A key insight behind NEAT is that it is *not* only the ultimate structure of the solution that really matters, but rather the structure of all the intermediate solutions along the way to finding the

solution. The connectivity of every intermediate solution represents a parameter space that evolution must optimize, and the more connections there are, the more parameters need to be optimized. Therefore, if the amount of structure can be minimized throughout evolution, so can the dimensionality of the spaces being explored, leading to significant performance gains.

In order to minimize structure throughout evolution, NEAT incrementally elaborates structure in a stochastic manner from a minimal starting point. Because of speciation, useful elaborations survive even if they are initially detrimental. Thus, NEAT strengthens the analogy between genetic algorithms and natural evolution by not only performing the optimizing function of evolution, but also this complexifying function, allowing solutions to become incrementally more complex at the same time as they become more optimal. The next section discusses the implications of complexification.

10.2 Benefits of Complexification

What makes complexification such a powerful search method? Whereas in fixed-topology evolution the good structures must be optimized in the high-dimensional space of the solutions themselves, complexifying evolution only searches high-dimensional structures that are elaborations of known good lower-dimensional structures. Before adding a new dimension, the values of the existing genes have already been optimized over preceding generations. Thus, after a new gene is added, the genome is *already* in a promising part of the new, higher-dimensional space. Thus, the search in the higher-dimensional space is not starting blindly as it would if evolution began searching in that space. It is for this reason that complexification can find high-dimensional solutions that fixed-topology or simplifying evolution cannot.

Complexification is particularly well suited for coevolution problems. When a fixed genome is used to represent a strategy, that strategy can be optimized, but it is not possible to add functionality without sacrificing some of the knowledge that is already encoded in the genome. In contrast, if new genetic material can be added, sophisticated elaborations can be layered above existing structure, establishing an evolutionary arms race. This process was evident in the robot duel domain (Chapter 5), where successive dominant strategies often built new functionality on top of existing behavior by adding new nodes and connections.

The advantages of complexification do not imply that fixed-sized genomes cannot sometimes evolve increasingly complex *phenotypic behavior*. Depending on the mapping between the genotype and the phenotype, it may be possible for a fixed, finite set of genes to represent solutions (phenotypes) with behaviors that vary in complexity. For example, such behaviors have been observed in Cellular Automata (CA), a computational structure consisting of a lattice of cells that change their state as a function of their own current state and the state of other cells in their neighborhood. This neighborhood function can be represented in a genome of size 2^{n+1} (assuming n neighboring cells with binary state) and evolved to obtain desired target behavior. For example,

Mitchell et al. (1996) were able to evolve neighborhood functions to determine whether black or white cells were in the majority in the CA lattice. The evolved CAs displayed complex global behavior patterns that converged on a single classification, depending on which cell type was in the majority. Over the course of evolution, CAs evolved from simple naive rules to significantly more complex rules that produce nonlocal interactions, even as the genome remained the same size.

In the CA example, the correct neighborhood size was chosen a priori. This choice is difficult to make, and crucial for success. If the desired behavior had not existed within the chosen size, even if the behavior would become gradually more complex, the system would never had solved the task. Interestingly, such a dead-end could be avoided if the neighborhood (i.e. the genome) could be expanded during evolution. It is possible that CAs could be more effectively evolved by complexifying (i.e. expanding) the genomes, and speciating to protect innovation, as in NEAT.

Moreover, not only can the chosen neighborhood be too small to represent the solution, it can also be unnecessarily large. Searching in a space of more dimensions than necessary can impede progress, as discussed above. If the desired function existed in a smaller neighborhood it could have been found with significantly fewer evaluations. Indeed, it is even possible that the most efficient neighborhood is not symmetric, or contains cells that are not directly adjacent to the cell being processed. Moreover, even the most efficient neighborhood may be too large a space in which to begin searching. Starting search in a small space and incrementing into a promising part of higher-dimensional space is more likely to find a solution. For these reasons, complexification can be an advantage, *even if* behavioral complexity can increase to some extent within a fixed space.

The search for an optimal policy in reinforcement learning is another useful example of where complexification can benefit search. Search proceeds by optimizing the value function, and much research has gone into how optimization can work to improve control policies (Sutton and Barto 1998) within the space of the value function. However, humans have always chosen the *representation* of the policy, whether it be a neural network, a state table, or something else. Yet it is this representation that defines whether the problem is tractable in the first place! On one hand, if the policy is encoded in a million-neuron neural network, a solution may never be found. On the other, if the neural network instead contains only a single neuron, a sufficiently sophisticated policy may not even exist in the space of possible solutions. In other words, RL is just as much about finding the right representation as it is about finding the solution within the space of that representation. Complexification is unlike almost any other approach in that it finds *both* the right level of representation *and* the solution simultaneously. Just as climbing a hill leads to a more optimal solution, so does increasing complexity lead to a more expressive representation. It is the combination of both of these principles that allows the wholly autonomous synthesis of solutions. NEAT is a principled approach to complexification, embodying the intuitive idea that novel representations need to be protected from premature elimination in order to give them a chance to realize their full potential.

The CA example raises the intriguing possibility that *any* structured phenotype can be evolved through complexification from a minimal starting point, historical markings, and the pro-

tection of innovation through speciation. The next section discusses the possibility that such a methodology could be applied to the evolution of arbitrary types of structures.

10.3 Non-neural NEAT

In addition to neural networks and CA, electrical circuits (Miller et al. 2000a,b), genetic programs (Koza 1992), robot body morphologies (Lipson and Pollack 2000), Bayesian networks (Mengshoel 1999), finite automata (Brave 1996), and building and vehicle architectures (O'Reilly 2000) are all structures of varying complexity that can benefit from complexification. By starting search in a minimal space and adding new dimensions incrementally, highly complex phenotypes can be discovered that would be difficult to find if search began in the intractable space of the final solution, or if it was prematurely restricted to too small a space.

The search for optimal structures is a common problem across AI. For example, Bayesian methods have been applied to learning model structure (Attias 2000; Ueda and Ghahramani 2002). In these approaches, the posterior probabilities of different structures are computed, allowing overly complex or simplistic models to be eliminated. Note that these approaches are not aimed at generating increasingly complex functional structures, but rather at providing a model that explains existing data. In other cases, solutions involve growing gradually larger structures, but the goal of the growth is to form gradually better *approximations*. For example, methods like Incremental Grid Growing (Blackmore and Miikkulainen 1995), and Growing Neural Gas (Fritzke 1995) add neurons to a network until it approximates the topology of the input space reasonably well. On the other hand, complexifying systems do not have to be non-deterministic (like NEAT), nor do they need to be based on evolutionary algorithms. For example, Harvey (1993) introduced a deterministic algorithm where the chromosome lengths of the entire population increase all at the same time in order to expand the search space; Fahlman and Lebiere (1990) developed a supervised (non-evolutionary) neural network training method called cascade correlation, where new hidden neurons are added to the network in a predetermined manner in order to complexify the function it computes. The conclusion is that complexification is an important general principle in AI.

In addition, methods have been developed in other areas of evolutionary computation to evolve phenotypes of variable structure and size. For example, in genetic programming, homologous crossover techniques based on program structure analysis allow arbitrary program trees to be effectively recombined (**?**). An interesting direction for future research is to compare these related methods with complexification as it is implemented in NEAT. It is possible that historical markings can be used to improve homologous crossover techniques in other areas of evolutionary computation.

In the future, complexification may help with the general problem of finding the appropriate level of abstraction for difficult problems. Complexification can start out with a simple, high-level description of the solution, composed of general-purpose elements. If such an abstraction is insuf-

ficient, it can be elaborated by breaking down each high-level element into lower level and more specific components. Such a process can continue indefinitely, leading to increasingly complex substructures, and increasingly low-level solutions to subproblems. Although in NEAT the solutions are composed of only connections and nodes, it does provide an early example of how such a process could be implemented.

One of the primary and most elusive goals of AI is to create systems that *scale up*. In a sense, complexification *is* the process of scaling up. It is the general principle of taking a simple idea and elaborating it for broader application. Much of AI is concerned with search, whether over complex multi-dimensional landscapes, or through highly-branching trees of possibilities. However, intelligence is as much about deciding *what space* to search as it is about searching once the proper space has already been identified. Currently, only humans are able to decide the proper level of abstraction for solving many problems, whether it be a simple high-level combination of general-purpose parts, or an extremely complex assembly of low-level components. A program that can decide what level of abstraction is most appropriate for a given domain would be a highly compelling demonstration of Artificial Intelligence. This is, I believe, where generic complexification methods can have their largest impact in the future.

10.4 Expanding the Neural Model

Most experiments in this dissertation evolved neural networks with a variant of the standard McCullough-Pitts neuron; incoming activations were multiplied by their connection weights, summed, and run through a sigmoid function (Section 2.2). The weights of connections did not change during activation. Chapter 6 described an elaboration of this neural model in which Hebbian equations with evolved parameters could adapt the weights of connections during activation. In the future, the neural model should be further enhanced to make it both more biologically plausible and to increase its functionality.

There are two primary ways to expand the neural model: (1) More learning parameters can be added to increase the flexibility of the neural network's ability to adapt during evaluation, and (2) the neural activation function can be enhanced to work more like real neurons. The remainder of this section described these two enhancements.

10.4.1 Additional Learning Parameters

While Hebbian learning is commonly cited as a major force for adaptation in the brain, other processes also change synaptic strength. For example, in *habituation*, connections weaken after prolonged low-level activation, regardless of how active the outgoing node is (Kandel et al. 1991). Habituation is useful when one or more inputs are irrelevant to the optimal behavior. In addition, hidden nodes that become irrelevant under certain conditions could be gradually filtered out during evaluation. For example, a network could learn to ignore an air conditioner humming in the

background.

Another important type of learning is *sensitization*. Incoming connections to a neuron are strengthened due to high activation over a *different* incoming connection. Sensitization allows an organism to increase general attention due to a specific stimulus. For example, if a robot keeps getting hit it may increase its sensitivity to all its inputs on the assumption that it is in a dangerous situation.

Finally, while the adaptive parameters in Chapter 6 are stored in connections, in some cases it makes more sense to store them in the neurons themselves. For example, since sensitization involves multiple connections, the parameters for sensitization would need to be centralized at the node where the connections converge. It is possible to store parameters in both nodes and connections in NEAT, although this dissertation does not describe any experiments with node-based parameters.

10.4.2 More Realistic Neural Activation

The simplified neural model employed so far in NEAT assumes that incoming activation is integrated into the receiving neuron all at one time in a discrete time step. While this common assumption is sufficient to support a broad range of sophisticated behavioral and strategic policies, it does not allow different neurons to display different temporal behavior. For example, while it may be advantageous for some neurons to react immediately to incoming signals, the network might be able to time its reactions more precisely if other neurons integrated incoming signals at a relatively slower rate. Such a network could then potentially evolve intricate temporal properties, such as complex oscillating patterns, without even the need for external input or clocks.

A network model with this capability is the Continuous-Time Recurrent Neural Network (CTRNNs; Yamauchi and Beer 1994). This type of network is commonly used in the evolution of walking gaits and locomotion behavior for various animal morphologies (Reil and Massey 2001; Reil and Husbands 2002; Terzopoulos et al. 1994). The CTRNN model allows regular oscillating patterns to evolve with precise time-dependent activation functions on each neuron.

The neurons used in CTRNNs are called *leaky integrator neurons*. The term "integrator" describes the neurons' ability to integrate incoming activation over time instead of all at once. "Leaky" means that some amount of internal activation leaks out of the neuron on each time step, so that its activity level would gradually sink to zero without input. The rate at which both incoming activation is integrated and existing internal activation is leaked is controlled by a *time constant* τ. The time constant allows different neurons to display different temporal properties. That way, combinations of neurons can create complex, precisely timed oscillations. In addition, neural activation can change smoothly over several time steps instead of jumping suddenly, allowing for smooth, natural motions.

Blynel and Floreano (2002) give a good overview of the equations underlying CTRNNs and leaky integrator neuron behavior. The general CTRNN equation describes how the internal

activation level or *state* γ of the neuron changes over time:

$$\frac{d\gamma}{dt} = \frac{1}{\tau_i}\left(\sum_{j=1}^{N} w_{ij}A_j + \sum_{k=1}^{S} w_{ik}I_k\right), \tag{10.1}$$

where i is the neuron index, N is the total number of neurons, S is the number of inputs, w_{ij} is the weight of the connection between neurons i and j, and I_k is sensory activation. The variable A_j is the incoming activation from neuron j, which is computed as $A_j = \sigma(\gamma_j - \theta_j)$, where θ_j is the bias of neuron j. While this equation describes how activation state changes over time, it cannot be used in practice in a discrete time system because it does not directly specify what should be added or subtracted from γ on each time step. To determine the discrete time update rule for γ equation 10.1 must be integrated to get $\gamma_i(n+1)$ in terms of $\gamma_i(n)$, where n is the time step number. The *Forward Euler* integration method (Hughes-Hallett et al. 1994, pp. 490-495). produces the discrete-time update rule (Blynel and Floreano 2002):

$$\gamma_i(n+1) = \gamma_i(n) + \frac{\Delta t}{\tau_i}\left(-\gamma_i(n) + \sum_{j=1}^{N} w_{ij}A_j(n) + \sum_{k=1}^{S} w_{ik}I_k\right). \tag{10.2}$$

In the future, each NEAT neuron can contain its own τ and these time constants can evolve just as connections weights. Then, using equation 10.2 as the activation function, the neural network can evolve sophisticated oscillation patterns and control e.g. robot gaits or swimming patterns.

10.5 Complexifying Artificial Embryogeny

The ultimate goal of neuroevolution, and evolutionary computation in general, is to evolve solutions to extremely difficult real-world problems. Such solutions are likely to require thousands or even millions of neurons or distinct parts. Even with the benefits of NEAT, if every gene were to map directly to a single unit of phenotypic structure, evolution would be searching for a solution in an intractable million-dimensional genotypic space.

In order to be tractable, the number of genes required to specify a phenotype must be orders of magnitude less than the number of structural units composing that phenotype. Nature has shown such representational systems to be possible on an enormous scale. Even with 100 trillion neural connections in the human brain, there are only about 30,000 active genes in the human genome (2800 million amino acids) (Deloukas et al. 1998; Zigmond et al. 1999).

Such representational efficiency is made possible through gene reuse. In an indirect genetic encoding (Section 2.3.2), a single gene may be used multiple times at different stages of development. There are two primary forms of reuse. First, phenotypic structures can occur in repeating patterns, where the same structural theme, perhaps with some variation, appears over and over again. Each time a pattern repeats, the same gene group can provide the specification. Examples of

repeating patterns in biological organisms include the numerous left/right symmetries of vertebrates (Raff 1996, pp. 302-303), and the numerous receptive fields in the visual cortex (Gilbert and Wiesel 1992; Hubel and Wiesel 1965). Repetition frequently involves variation on a general theme. For example, each vertebrae in the spine is formed similarly to the others, albeit with different incoming and outgoing connections (Zigmond et al. 1999, pp. 30-31).

The second primary form of reuse occurs when the same gene product is used to *initiate* separate developmental pathways. For example, Cohn et al. (1997) found that the same gene product, fibroblast growth factor (FGF), induces the appearance of *both* forelimbs and hindlimbs depending on the part of the body where the FGF is applied. Thus, the same gene can be used to initiate different structures at different locations.

Indirect encoding refers to all possible mappings between genotype and phenotype in which genes can be reused. For example, a gene may contain a parameter specifying how many times its corresponding structure should appear in the phenotype. Natural organisms implement gene reuse through a process of development, or embryogeny.[1] The same genes can be used at different points in development for different purposes, and the order in which activations of genes take place determines when and where a particular gene is expressed (Raff 1996). Thus, mappings between genotype and phenotype based on artificial embryogeny are a subset of all possible indirect encodings. Recently, researchers have begun to replicate this process in artificial developmental systems. The hope is that extremely compact codes can evolve to represent immensely complex phenotypes.

Several names have been used for artificial evolutionary systems that utilize a developmental phase, including Artificial Ontogeny (Bongard and Pfeifer 2001), Computational Embryogeny (Bentley and Kumar 1999), Cellular Encoding (Gruau 1994), and morphogenesis (Jakobi 1995). The term Artificial Embryogeny (AE) can be used to refer to the entire class of such systems.[2] Neural networks are a natural application of AE because current neural network models are orders of magnitude less complex than their biological counterparts. The potential benefit of combining complexification through expanding the genome with AE is a system that can discover networks of millions of neurons or more. Thus, a significant goal for future work is to combine NEAT with an AE-based indirect encoding.

It is possible to combine NEAT with an indirect encoding because any kinds of genes can be tracked through historical markings, including indirect encodings. For example, in grammatical rewrite systems, a gene is a rule that specifies how a symbol in the developing phenotype should be expanded (Belew and Kammeyer 1993; Boers and Kuiper 1992; Hornby and Pollack 2001a,b; Kitano 1990; Lindenmayer 1968). Another approach is to encode development as a tree of instruction genes that are executed in parallel by different parts of a developing phenotype (Gruau et al. 1996;

[1] Bentley and Kumar (1999) pointed out that the correct term is *embryogeny* as opposed to *embryology*. Embryogeny is the embryological process of development itself, while embryology is the *study* of the process of development. This discussion focuses on evolving developmental systems, i.e. implementing artificial embryogeny.

[2] *Embryogeny* conveys that systems in this class develop phenotypes using genetic information starting from a small initial structure.

Komosinski and Rotaru-Varga 2001; Luke and Spector 1996). Other indirect encodings attempt to simulate genetic regulatory networks (GRNs) in biology (Bongard and Pfeifer 2001; Astor and Adami 2000; Dellaert and Beer 1994; Eggenberger 1997; Jakobi 1995). In a GRN, genes produce signals that either activate or inhibit other genes in the genome. Some genes produce signals that cause e.g. cells to grow or axons to form. The interaction of all the genes forms a network that produces a phenotype. All these encodings support variable length genomes and historical markings, making complexification possible. For a complete review of prior work in AE, see Stanley and Miikkulainen (2003).

Direct encodings such as NEAT generally expand the genome by adding random genes (Angeline et al. 1993; Pujol and Poli 1998; Stanley and Miikkulainen 2002b). In contrast, nature uses duplication as the primary means of expanding the genome. The reason is that when genes are duplicated, the phenotype is not dramatically altered (Force et al. 1999). Such stability is important because generally major mutations in the genome could permanently and immediately disabled the lineage. As Force et al. (1999) explained, subsequent mutations repartition the roles of both the original genes and the duplicated genes without significantly altering the overall developmental plan. Once duplicate genes have undergone sufficient mutation to be activated at different times during development than their original counterparts, subsequent mutations can begin to alter how these new instances develop. Thus, because of the duplicate genes, evolution has the flexibility to alter the developmental process in more ways than were possible before the duplication.

Such a gradual process is difficult to achieve with direct encodings. When each gene maps to a single unit of phenotypic structure, duplicating a gene is equivalent to duplicating part of the phenotype, which can significantly alter its functionality and structure. While in some cases such duplication is not destructive (e.g. with single neurons), duplicating an entire substructure of multiple components likely is.

In contrast, with AE-based indirect encoding, the duplicate genes can have overlapping, redundant roles. Thus they are guaranteed to affect development as soon as they are incorporated. Then, over the following generations, they can be partitioned gradually into different but related roles. Therefore, through AE, duplication can complexify solutions without decreasing their performance.

The duplication process must carefully integrate the new genes into the already-existing developmental plan of the organism, and the subsequent mutations must not be too severe. If the genes become disconnected from the existing developmental plan, subsequent mutations will likely have little effect. Thus, in order to allow duplicate genes to gradually take on new roles, the conditions under which they activate should lie on a continuum. A slight mutation in one duplicate should cause it to be activated in some but not all cases where its counterpart was formerly always activated.

In addition to reducing the search space, in complexifying evolution with AE important substructures only need to be discovered once, even when they appear in the phenotype multiple

times. Reuse of new structure is *inherent* in the underlying developmental program that has already evolved. For example, appendages can evolve digits all at once since the existing genetic specification ensures that each appendage follows the same developmental process. Thus, new genes that specify new structure at the end of such a process will be encountered each time the process occurs. On the other hand, complexification with direct encoding would require digits to be discovered separately on several occasions, since each set of digits must be specified by a separate set of genes. Thus, combining complexification and AE is potentially powerful because it makes it possible to elaborate all repeating structures simultaneously.

The main idea behind combining AE-based indirect encoding with NEAT is that new genes that are added through duplication can be assigned historical markings just as new connections are now. Thus, it is possible to implement the same kind of artificial synapsis (Section 2.4.4) based on historical marking as in directly-encoded NEAT.

By combining synapsis and gradual divergence of duplicate genes, researchers can begin to study different ways to implement gene duplication.[3] For example, how should clusters of genes be chosen for duplication? Everything from copying single genes to duplicating whole genomes is possible. While biologists continue to debate this issue (Amores et al. 1998; Martin 1999), it can also be addressed through experiments in evolutionary computation. Calabretta et al. (2000) have already shown that distinct neural modules emerge when clusters of genes are duplicated in the evolution of neural networks. Thus duplicating entire groups of genes can be beneficial.

A complexifying system that starts with small, simple genomes will first evolve basic structures, such as bilateral symmetry, and then elaborate on them in future generations by adding new genes. The original simple developmental plan provides a framework that can be elaborated and enhanced in the future. By accumulating such enhancements, complex structures can evolve that would have been difficult to discover all at once. One of the most intriguing phenomena that might emerge from a successful implementation is repetition with variation. That is, instead of duplicating the same structure multiple times, a general *theme*, such as a limb, can be reused multiple times with differing manifestations. Such patterns do not follow traditional modular design in engineering, in which discrete identical parts are assembled into larger constructions. Instead, the beginnings and ends of individual parts are amorphous, and their internal structure is only vaguely constrained. The capacity to reuse parts with variation is potentially a very powerful way to create complexity, and a most intriguing direction of future research with NEAT.

[3] *Gene deletion* is also possible, although it is potentially more deleterious than duplication. Duplication creates redundancy, which does not cause any loss of functionality. In contrast, deletion may cause important steps in development to be removed.

10.6 Conclusion

NEAT has shown promise in a variety of domains because it can complexify at the same time as protecting innovation. Although NEAT is used in this dissertation to evolve neural networks, it is a general method that can eventually be used to evolve any structured phenotype. Already the neural model has been enhanced by adding learning parameters (Chapter 6). In the future, the neural model can be further expanded to include more types of synaptic adaptation, and to allow leaky integrator neuron activation. Finally, by combining NEAT with an indirect encoding, an AE version NEAT may be able to evolve networks or other structures with thousands or millions of parts. The possible future research directions with NEAT are exciting because each may lead to significant breakthroughs in many fields of science and engineering.

Chapter 11

Conclusion

In this dissertation, the NeuroEvolution of Augmenting Topologies (NEAT) method for evolving neural network topologies was presented and evaluated. This chapter summarizes the contributions of the dissertation and places NEAT in the general context of innovation for machine learning.

11.1 Contributions

NEAT is a significant advance in evolutionary computation (EC), neuroevolution (NE), and machine learning (ML). NEAT is the first EC method to track genes through historical markings, making possible the first principled methodology for evolving a population of diverse topologies. Comparisons in Chapter 4 showed that this approach is indeed powerful, outperforming traditional reinforcement learning methods in a difficult benchmark task.

Yet the ultimate goal of ML is more than finding solutions quickly. Ideally, ML should *innovate*, not only by optimizing parameters within a provided model, but by creating entire models on its own. It is here that NEAT makes its most significant contribution, since NEAT is an algorithm that embodies a philosophy of innovation. It is a common principle that the young should be given a sufficient chance to reach their potential even if they sometimes fail along the way; in the same vein, it is in the interest of society that novel ideas should be thoroughly tested and refined before they are adopted or discarded. In NEAT, separating a novel network topology from the rest of the population so that it can compete within its own niche protects innovation and holds true to this philosophy. Moreover, when the entire population is gradually *complexifying* by adding new structure and creating new species, it becomes possible to discover novel solutions. Thus, NEAT is a genuine algorithm for innovation. This design was confirmed in the robot duel experiments of Chapter 5, where complexification and speciation were shown to establish an evolutionary arms race of continual innovation.

The innovation approach lays the foundation for novel of experiments and applications that were not previously possible. First, in Chapter 6, two ways of evolving adaptive neural networks

with NEAT were introduced. Hebbian plastic networks that change their connection weights over their lifetime based on experience were compared to static recurrent networks that adapt by changing their internal activation levels. The results showed that the different types of adaptive networks are appropriate for different situations. This experiment also demonstrated how NEAT can be used to evolve and compare new kinds of neural networks, opening up new avenues of research.

Second, by enhancing NEAT to work in real-time, it became possible for humans to interact directly with evolving populations in real time, creating a new genre of video games (NERO; Chapter 9). Real-time NEAT makes further interactive educational, entertainment, and training applications possible, all with the potential to discover increasingly sophisticated solutions in real time as the population complexifies.

Third, NEAT achieved promising results in two challenging real-world domains: the game of Go and an automobile warning system. NEAT was able to scale up to defeat Gnugo on a 7×7 board by controlling a roving eye that cannot see the whole board at once. NEAT also evolved successful drivers and predictors in the RARS driving simulator, forming a foundation for warning systems that may someday save lives.

An important feature of NEAT is that all its operations are genetic; it makes no assumptions about the phenotype. Therefore, NEAT potentially can be generalized to evolve various structured phenotypes in the future, from electronic circuits to robot morphologies. In fact, even the encoding, i.e. the mapping between genotype and phenotype, is independent of the algorithm, and therefore can be improved and expanded without changing NEAT. Thus, NEAT is a general methodology for evolving innovative solutions in many domains, with great potential for further development in the future.

11.2 Conclusion

Tracking genes through historical markings, speciation to protect innovation, and complexification from a small starting point combine to produce a novel methodology for evolving a population of increasingly complex and diverse structures. Not only is this method fast, but it supports continual innovation. Thus, this dissertation is a first step in automating the discovery of complex solutions to difficult real-world problems.

Appendix A

Parameter Values

This appendix describes parameter settings for all the experiments in the dissertation. It is divided into four sections: The first section defines NEAT's system parameters. The following section lists parameters that are common to all experiments. Next, parameters that differ among the experiments are listed and explained. The last section gives specific details about each experiment, including parameters that are unique to different simulators. NEAT C++ source code is available on the web at `http://nn.cs.utexas.edu/keyword?neat`. The experiments in this dissertation can be reproduced with the source code and the parameter settings in this appendix

A.1 Definitions

There are 16 primary system parameters:

1. **Weight Mutation Range**: The maximum magnitude of a mutation that changes the weight of a connection (Section 3.1).
2. c_1: Coefficient representing how important excess genes are in measuring compatibility (equation 3.1).
3. c_2: Coefficient for disjoint genes (equation 3.1).
4. c_3: Coefficient for average weight difference (equation 3.1).
5. C_t: Compatibility threshold (equation 3.1); when dynamic thresholding is used, this variable determines the starting threshold.
6. **Survival Threshold**: Percentage of each species allowed to reproduce (Section 3.3).
7. **Mutate Only Probability**: Probability that a reproduction will only result from mutation and not crossover .

8. **Add Node Probability**: Probability a new node gene will be added to the genome (Section 3.1).

9. **Add Link Probability**: Probability a new connection will be added (Section 3.1).

10. **Interspecies Mating Rate**: Percentage of crossovers allowed to occur between parents of different species (Section 3.3).

11. **Mate By Choosing Probability**: Probability that genes will be chosen one at a time from either parent during crossover (Section 3.2).

12. **Mate By Averaging Probability**: Probability that matching genes will be averaged during crossover (Section 3.2).

13. **Mate Only Probability**: Probability an offspring will be created only through crossover without mutation.

14. **Recurrent Connection Probability**: Probability a new connection will be recurrent.

15. **Population Size**: Number of networks in the population.

16. **Maximum Stagnation**: Maximum number of generations a species is allowed to stay the same fitness before it is removed. In competitive coevolution, the worst species is removed if it has been around this many generations (Section 3.3).

17. **Target Number of Species**: Desired number of species in the population; used only in dynamic compatibility thresholding (Section 3.3).

The appropriate values for these parameters were found experimentally, as will be described in the following sections. Many of them follow a logical pattern: For example, links need to be added significantly more often than nodes, an average weight difference of 0.5 is about as significant as one disjoint or excess gene, and the appropriate compatibility settings depend on factors explained in detail in Section A.3. Through extensive experimentation, performance was found to be robust to moderate variations in the parameter values.

A.2 Common Parameters

Table A.1 lists parameters that were the same across all experiments. In addition, a modified sigmoidal transfer function, $\varphi(x) = \frac{1}{1+e^{-4.9x}}$, was used at all nodes. The steepened sigmoid allows more fine tuning at extreme activations. It is optimized to be close to linear during its steepest ascent between activations -0.5 and 0.5.

Parameter	Value
c_1	1.0
c_2	1.0
Mutate Only Prob.	0.25
Mate By Choosing Prob.	0.6
Mate By Averaging Prob.	0.4
Mate Only Prob.	0.2
Recurrent Connection Prob.	0.2

Table A.1: **Common parameter settings.** These parameter values were used in every experiment.

Parameter	DPM	DPNM	DP-SMALL	Duel	Adapt	Go	Auto	NERO
Population Size	150	150	16	256	500	400	100	50
c_3	0.4	3.0	3.0	2.0	2.0	2.0	3.0	0.4
C_t	3.0	4.0	12.0	3.0	6.0	8.0	3.0	4.0
Weight Mutation Power	2.5	1.8	3.5	2.5	2.5	1.5	0.1	0.5
Survival Threshold	0.2	0.4	0.3	0.2	0.2	0.2	0.2	1.0
Add Node Probability	0.03	0.001	0.001	0.0025	0.005	0.005	0.02	0.03
Add Link Probability	0.05	0.03	0.03	0.1	0.05	0.1	0.1	0.05
Interspecies Mating Rate	0.001	0.001	0.001	0.05	0.01	0.05	0.05	0.05
Maximum Stagnation	15	15	1,000	20	130	200	NA	NA
Target Number of Species	NA	NA	2	20	20	20	8	4

Table A.2: **Variable parameter settings.** Across each row the values for a specific NEAT parameters are listed for each experiment in the dissertation. The following abbreviations identify the experiments: DPM = Double Pole Balancing Markov (also used for XOR); DPNM= Double Pole Balancing Non Markov; DP-SMALL = Small Population Double Pole Balancing; Duel = Robot Duel; Adapt = Adaptive NEAT; Go = Roving Eye for Go; Auto = Automobile Warning System; NERO = NeuroEvolving Robotic Operatives with rtNEAT. Entries with "NA" indicate that parameter was not used in the experiment.

A.3 Variable Parameters

Several parameter values differed among the experiments (table A.2). Importantly, experiments required different population sizes; for example, in pole balancing, population sizes were chosen based on previous work. In NERO, the population was small so that the CPU could accommodate all the agents evolving simultaneously. In other experiments, the population was chosen to be as large as possible to still allow at least 500 generations of evolution in under two weeks. The population size affects several other parameters, and those parameters were also varied. For example, larger populations can accommodate higher rates of structural mutation and more species since there is more room to try new topologies. When the population is bigger, there is also more room

to separate species based on weight differences in addition to topological differences. Other factors affecting parameter values were how many generations the experiment takes to find a satisfactory behavior, how difficult the task is, and whether evolution happens in real-time. For example, structure was added more slowly in more difficult domains to give the system time to optimize existing connections. Table A.2 lists those parameters that differed among experiments.

A.4 Experiment-Specific Parameter Settings

Some experiments used additional parameters that are specific to a simulator or engine. This section explains what those parameters are and gives their values or points to where they can be found.

A.4.1 Pole Balancing

The equations of motion for N unjointed poles balanced on a single cart (Section 4.2) are

$$\ddot{x} = \frac{F - \mu_c sgn(\dot{x}) + \sum_{i=1}^{N} \tilde{F}_i}{M + \sum_{i=1}^{N} \tilde{m}_i}$$

$$\ddot{\theta}_i = -\frac{3}{4l_i}(\ddot{x}\cos\theta_i + g\sin\theta_i + \frac{\mu_{pi}\dot{\theta}_i}{m_i l_i})$$

Where $\tilde{F}_i$ is the effective force from the ith pole on the cart

$$\tilde{F}_i = m_i l_i \dot{\theta_i^2}\sin\theta_i + \frac{3}{4}m_i\cos\theta_i(\frac{\mu_{pi}\dot{\theta}_i}{m_i l_i} + g\sin\theta_i)$$

and $\tilde{m_i}$ is the effective mass of the ith pole

$$\tilde{m_i} = m_i(1 - \frac{3}{4}\cos^2\theta_i)$$

Parameters for the double pole problem are listed in table A.3.

A.4.2 Robot Duel

Unlike in the other experiments, two separate populations were coevolved in the robot duel experiments (Section 5.1).

Robot motion was simulated as follows. The turn angle θ is determined as $\theta = 0.24|l - r|$, where l is the output of the left turn neuron, and r is the output of the right turn neuron. The robot moves forward a distance of $1.33f$ on the 600 by 600 board, where f is the forward motion output. These coefficients were calibrated experimentally to achieve accurate and smooth motion with neural outputs between zero and one.

Sym.	Description	Value
x	Position of cart on track	[-2.4,2.4] m
θ	Angle of pole from vertical	[-36,36] deg.
F	Force applied to cart	[-10,10] N
l_i	Half length of ith pole	l_1 = 0.5m l_2 = 0.05m
M	Mass of cart	1.0 kg
m_i	Mass of ith pole	m_1 = 0.1 kg & m_2 = 0.01 kg
μ_c	Coefficient of friction of cart on track	0.0005
μ_p	Coefficient of friction if ith pole's hinge	0.000002

Table A.3: **Simulation parameters for the double pole domain.**

A.4.3 Adaptive NEAT

With the plastic networks, genomes contained four rules with values for η_1 and η_2 (Section 6.2). The first rule was fixed at zero to allow genes to encode fixed connection weights that do not adapt. The probability of mutating the parameters of a rule was 0.3, in which case each parameter mutated with a probability of 0.2 by adding a uniform random value between 0 and 0.5. A connection gene had a 0.05 chance of being pointed to a different rule.

The learning rules affect how compatible network genomes are. Thus they were factored into network compatibility calculations. The average difference in learning rule parameter values was added to $\overline{W}$ in calculating the compatibility of two genes. That way, speciation took into account learning rule differences in addition to topology and weight differences.

A.4.4 Roving Eye for Go

In the Go experiments (Section 7.4), except when the 5×5 champion was used to start 7×7 evolution, the starting genome had several sensors initially disconnected. That way, NEAT could start in a lower-dimensional space and decide on its own which sensors to use. Specifically, the out-of-range sensors all started out disconnected, and the front-left, front-right, back-left, back-center, and back-right sensors were disconnected from the network. However, these sensors were still present in the genome and were frequently eventually connected to the network by NEAT over the course of evolution.

A.4.5 Automobile Warning System

The complete physics model used in RARS (Section 8.2) is described and source code is available at `http://rars.sourceforge.net/`.

A.4.6 NeuroEvolving Robotic Operatives

In NERO (Section 9.3), the percentage of the population allowed to be ineligible at one time was 50%. The number of ticks between replacements is 20 and the minimum evaluation time is 500. The number of ticks between replacements can also be derived from equation 9.3. NERO physics is controlled by the Torque Engine, which is licensed from GarageGames (`http://www.garagegames.com/`).

Bibliography

Agogino, A., Stanley, K., and Miikkulainen, R. (2000). Online interactive neuro-evolution. *Neural Processing Letters*, 11:29–38.

Aharonov-Barki, R., Beker, T., and Ruppin, E. (2001). Emergence of memory-Driven command neurons in evolved artificial agents. *Neural Computation*, 13(3):691–716.

Albus, J. S. (1975). A new approach to manipulator control: The cerebellar model articulation controller. *Journal of Dynamic Systems, Measurement, and Control*, 97(3):220–227.

Almeida, L. B. (1987). A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In *Proceedings of the IEEE First International Conference on Neural Networks* (San Diego, CA), vol. II, 609–618. Piscataway, NJ: IEEE.

Amores, A., Force, A., Yan, Y.-L., Joly, L., Amemiya, C., Fritz, A., Ho, R. K., Langeland, J., Prince, V., Wang, Y.-L., Westerfield, M., Ekker, M., and Postlethwait, J. H. (1998). Zebrafish HOX clusters and vertebrate genome evolution. *Science*, 282:1711–1784.

Angeline, P. J., and Pollack, J. B. (1993). Competitive environments evolve better solutions for complex tasks. In (Forrest 1993), 264–270.

Angeline, P. J., Saunders, G. M., and Pollack, J. B. (1993). An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5:54–65.

Astor, J., and Adami, C. (2000). A developmental model for the evolution of artificial neural networks. *Artificial Life*, 6(3):189–218.

Attias, H. (2000). A variational bayesian framework for graphical models. In *Advances in Neural Information Processing Systems, 12*, 209–215. Cambridge, MA: MIT Press.

Bäck, T., editor (1997). *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA-97, East Lansing, MI)*. San Francisco, CA: Morgan Kaufmann.

Baxter, J. (1992). The evolution of learning algorithms for artificial neural networks. In Green, D., and Bossomaier, T., editors, *Complex Systems*, 313–326. Amsterdam: IOS Press.

Bayer, A., Bump, D., Denholm, D., Dumonteil, J., Farneback, G., Traber, T., Urvoy, T., and Wallin, I. (2002). *GNU Go 3.2*. Cambridge, MA: Free Software Foundation.

Beasley, D., Bull, D. R., and Martin, R. R. (1993). A sequential niche technique for multimodal function optimization. *Evolutionary Computation*, 1(2):101–125.

Belew, R. K., and Kammeyer, T. E. (1993). Evolving aesthetic sorting networks using developmental grammars. In (Forrest 1993).

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.

Bentley, P. J., and Kumar, S. (1999). The ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, 35–43. San Francisco, CA: Morgan Kaufmann.

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

Blackmore, J., and Miikkulainen, R. (1995). Visualizing high-dimensional structure with the incremental grid growing neural network. In Prieditis, A., and Russell, S., editors, *Machine Learning: Proceedings of the 12th Annual Conference*, 55–63. San Francisco, CA: Morgan Kaufmann.

Blynel, J., and Floreano, D. (2002). Levels of dynamics and adaptive behavior in evolutionary neural controllers. In *Proceedings of the Seventh International Conference on Simulation of Adaptive Behavior on From Animals to Animats*, 272–281.

Boers, E. J., and Kuiper, H. (1992). *Biological Metaphors and the Design of Modular Artificial Neural Networks*. Master's thesis, Departments of Computer Science and Experimental and Theoretical Psychology at Leiden University, The Netherlands.

Bongard, J. C., and Pfeifer, R. (2001). Repeated structure and dissociation of genotypic and phenotypic complexity in artificial ontogeny. In (Spector et al. 2001), 829–836.

Bouzy, B., and Cazenave, T. (2001). Computer Go : An AI oriented survey. *Artificial Intelligence Journal*, 132(1):39–193.

Braun, H., and Weisbrod, J. (1993). Evolving feedforward neural networks. In *Proceedings of ANNGA93, International Conference on Artificial Neural Networks and Genetic Algorithms*. Innsbruck: Springer-Verlag.

Brave, S. (1996). Evolving deterministic finite automata using cellular encoding. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, 39–44. Stanford University, CA, USA: MIT Press.

Brooks, R. A., and Maes, P., editors (1994). *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems (Artificial Life IV)*. Cambridge, MA: MIT Press.

Calabretta, R., Nolfi, S., Parisi, D., and Wagner, G. P. (2000). Duplication of modules facilitates the evolution of functional specialization. *Artificial Life*, 6(1):69–84.

Carroll, S. B. (1995). Homeotic genes and the evolution of arthropods and chordates. *Nature*, 376:479–485.

Cazenave, T. (2000). Generation of patterns with external conditions for the game of Go. *Advance in Computer Games*, 9:275–293.

Chalmers, D. J. (1990). The evolution of learning: An experiment in genetic connectionism. In (Touretzky et al. 1990), 81–90.

Cliff, D., Harvey, I., and Husbands, P. (1993). Explorations in evolutionary robotics. *Adaptive Behavior*, 2:73–110.

Cohn, M., Patel, K., Krumlauf, R., Wilkinson, D., Clarke, J., and Tickle, C. (1997). HOX9 genes and vertebrate limb specification. *Nature*, 387:97–101.

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

Darnell, J. E., and Doolittle, W. F. (1986). Speculations on the early course of evolution. *Proceedings of the National Academy of Sciences, USA*, 83:1271–1275.

Darwen, P., and Yao, X. (1996). Automatic modularization by speciation. In *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation (ICEC '96)*, 88–93. Piscataway, NJ: IEEE Computer Society Press.

Darwen, P. J. (1996). *Co-Evolutionary Learning by Automatic Modularisation with Speciation*. PhD thesis, School of Computer Science, University College, University of New South Wales.

Dasgupta, D., and McGregor, D. (1992). Designing application-specific neural networks using the structured genetic algorithm. In *Proceedings of the International Conference on Combinations of Genetic Algorithms and Neural Networks*, 87–96.

Dawkins, R., and Krebs, J. R. (1979). Arms races between and within species. *Proceedings of the Royal Society of London Series B*, 205:489–511.

De Jong, E. D. (2004). The incremental pareto-coevolution archive. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004)*. Berlin: Springer Verlag.

De Jong, K. A. (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, The University of Michigan, Ann Arbor, MI. University Microfilms No. 76-09381.

Dellaert, F., and Beer, R. (1994). Toward an evolvable model of development for autonomous agent synthesis. In (Brooks and Maes 1994).

Deloukas, P., Schuler, G., Gyapay, G., Beasley, E., Soderlund, C., Rodriguez-Tome, P., Hui, L., Matise, T., McKusick, K., Beckmann, J., Bentolila, S., Bihoreau, M., Birren, B., Browne, J., Butler, A., Castle, A., Chiannilkulchai, N., Clee, C., Day, P., Dehejia, A., Dibling, T., Drouot, N., Duprat, S., Fizames, C., and Bentley, D. (1998). A physical map of 30,000 human genes. *Science*, 282(5389):744–746.

Dickinson, S. J., Christensen, H. I., Tsotsos, J. K., and Olofsson, G. (1997). Active object recognition integrating attention. *Computer Vision and Image Understanding*, 67(3):239–260.

Dickmanns, E., and Mysliwetz, B. (1992). Recursive 3-D road and relative ego-state recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):199–213.

Eggenberger, P. (1997). Evolving morphologies of simulated 3D organisms based on differential gene expression. In Husbands, P., and Harvey, I., editors, *Proceedings of the Fourth European Conference on Artificial Life*, 205–213. Cambridge, MA: MIT Press.

Elman, J. L. (1991). Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 7:195–225.

Enzenberger, M. (2003). Evaluation in go by a neural network using soft segmentation. In *Proceedings of the 10th Advances in Computer Games conference*, 97–108.

Fahlman, S. E., and Lebiere, C. (1990). The cascade-correlation learning architecture. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 2*, 524–532. San Francisco, CA: Morgan Kaufmann.

Ficici, S. G., and Pollack, J. B. (2001). Pareto optimality in coevolutionary learning. In Kelemen, J., editor, *Sixth European Conference on Artificial Life*. Berlin; New York: Springer-Verlag.

Floreano, D., and Nolfi, S. (1997). God save the red queen! Competition in co-evolutionary robotics. *Evolutionary Computation*, 5.

Floreano, D., and Urzelai, J. (2000). Evolutionary robots with on-line self-organization and behavioral fitness. *Neural Networks*, 13:431–4434.

Floriano, D., and Mondada, F. (1994). Automatic creation of an autonomous agent: Genetic evolution of a neural-network driven robot. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*.

Force, A., Lynch, M., Pickett, F. B., Amores, A., lin Yan, Y., and Postlethwait, J. (1999). Preservation of duplicate genes by complementary, degenerative mutations. *Genetics*, 151:1531–1545.

Forrest, S., editor (1993). *Proceedings of the Fifth International Conference on Genetic Algorithms*. San Francisco, CA: Morgan Kaufmann.

Fritzke, B. (1995). A growing neural gas network learns topologies. In G.Tesauro, D.S.Touretzky, and T.K.Leen, editors, *Advances in Neural Information Processing Systems 7*, 625–632. Cambridge, MA: MIT Press.

Fullmer, B., and Miikkulainen, R. (1992). Using marker-based genetic encoding of neural networks to evolve finite-state behaviour. In Varela, F. J., and Bourgine, P., editors, *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, 255–262. Cambridge, MA: MIT Press.

Gilbert, C. D., and Wiesel, T. N. (1992). Receptive field dynamics in adult primary visual cortex. *Nature*, 356:150–152.

Gold, S., and Rangarajan, A. (1996). A graduated assignment algorithm for graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18:377–388.

Goldberg, D., Korb, B., and Deb, K. (1989). Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3(5):493–530.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley.

Goldberg, D. E., and Richardson, J. (1987). Genetic algorithms with sharing for multimodal function optimization. In Grefenstette, J. J., editor, *Proceedings of the Second International Conference on Genetic Algorithms*, 148–154. San Francisco, CA: Morgan Kaufmann.

Gomez, F., and Miikkulainen, R. (1997). Incremental evolution of complex general behavior. *Adaptive Behavior*, 5:317–342.

Gomez, F., and Miikkulainen, R. (1998). 2-D pole-balancing with recurrent evolutionary networks. In *Proceedings of the International Conference on Artificial Neural Networks*, 425–430. Berlin; New York: Springer-Verlag.

Gomez, F., and Miikkulainen, R. (1999). Solving non-Markovian control tasks with neuroevolution. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*. Denver, CO: Morgan Kaufmann.

Gomez, F. J. (2003). *Robust non-linear control through neuroevolution*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin. Technical Report AI-TR-03-303.

Gomez, F. J., and Miikkulainen, R. (2003). Active guidance for a finless rocket through neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2003)*. Berlin: Springer Verlag.

Gruau, F. (1993). Genetic synthesis of modular neural networks. In (Forrest 1993), 318–325.

Gruau, F. (1994). *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm*. PhD thesis, Ecole Normale Superieure de Lyon, France.

Gruau, F., Whitley, D., and Pyeatt, L. (1996). A comparison between cellular encoding and direct encoding for genetic neural networks. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, 81–89. Cambridge, MA: MIT Press.

Haag, M., and Nagel, H.-H. (1999). Combination of edge element and optical flow estimates for 3d-model-based vehicle tracking in traffic image sequences. *International Journal of Computer Vision*, 35(3):295–319.

Hancock, P. J. B. (1992). Genetic algorithms and permutation problems: A comparison of recombination operators for neural net structure specification. In Whitley, D., and Schaffer, J., editors, *International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92)*. IEEE Computer Society Press.

Harvey, I. (1993). *The Artificial Evolution of Adaptive Behavior*. PhD thesis, School of Cognitive and Computing Sciences, University of Sussex, Sussex.

Haykin, S. (1994). *Neural Networks: A Comprehensive Foundation*. New York: Macmillan.

Herrera, F., and Lozano, M. (1998). Tackling real-coded genetic algorithms: Operators and tools for the behavioural analysis. *Artificial Intelligence Review*, 12(4):265–319.

Hershberger, D., Burridge, R., Kortenkamp, D., and Simmons, R. (2000). Distributed visual servoing with a roving eye. In *Proceedings of the Conference on Intelligent Robots and Systems (IROS)*.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Ann Arbor, MI: University of Michigan Press.

Holland, P. W., Garcia-Fernandez, J., Williams, N. A., and Sidow, A. (1994). Gene duplications and the origin of vertebrate development. *Development Supplement*, 125–133.

Hornby, G. S., and Pollack, J. B. (2001a). The advantages of generative grammatical encodings for physical design. In *Proceedings of the 2002 Congress on Evolutionary Computation*.

Hornby, G. S., and Pollack, J. B. (2001b). Body-brain co-evolution using L-systems as a generative encoding. In (Spector et al. 2001).

Hornby, G. S., and Pollack, J. B. (2002). Creating high-level components with a generative representation for body-brain evolution. *Artificial Life*, 8(3).

Hubel, D. H., and Wiesel, T. N. (1965). Receptive fields and functional architecture in two nonstriate visual areas (18 and 19) of the cat. *Journal of Neurophysiology*, 28:229–289.

Hughes-Hallett, D., Gleason, A. M., Flath, D. E., Gordon, S. P., Lomen, D. O., Lovelock, D., McCallum, W. G., Osgood, B. G., Pasquale, A., Tecosky-Feldman, J., Thrash, J. B., Thrash, K. R., Tucker, T. W., and Bretscher, O. K. (1994). *Calculus*. New York, NY: John Wiley & Sons, Inc.

Igel, C. (2003). Neuroevolution for reinforcement learning using evolution strategies. In Sarker, R., Reynolds, R., Abbass, H., Tan, K. C., McKay, B., Essam, D., and Gedeon, T., editors, *Congress on Evolutionary Computation 2003 (CEC 2003)*, 2588–2595. Piscataway, NJ: IEEE Press.

Jakobi, N. (1995). Harnessing morphogenesis. In *Proceedings of Information Processing in Cells and Tissues*, 29–41. University of Liverpool.

Jim, K.-C., and Giles, C. L. (2000). Talking helps: Evolving communicating agents for the predator-prey pursuit problem. *Artificial Life*, 6(3):237–254.

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

Kandel, E. R., Schwartz, J. H., and Jessell, T. M., editors (1991). *Principles of Neural Science*. New York: Elsevier. Third edition.

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

Komosinski, M., and Rotaru-Varga, A. (2001). Comparison of different genotype encodings for simulated 3D agents. *Artificial Life*, 7(4):395–418.

Koza, J. (1995). Gene duplication to enable genetic programming to concurrently evolve both the architecture and work-performing steps of a computer program. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann.

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

Krishnan, R., and Ciesielski, V. B. (1994). 2DELTA-GANN: A new approach to training neural networks using genetic algorithms. In *Proceedings of the Australian Conference on Neural Networks*, 194–197.

Laird, J. E., and van Lent, M. (2000). Human-level AI's killer application: Interactive computer games. In *Proceedings of the 17th National Conference on Artificial Intelligence*. Cambridge, MA: MIT Press.

Lee, C.-H., and Kim, J.-H. (1996). Evolutionary ordered neural network with a linked-list encoding scheme. In *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, 665–669.

Lin, T., Horne, B. G., and Giles, C. L. (1996). How embedded memory in recurrent neural network architectures helps learning long-term temporal dependencies. Technical Report CS-TR-3626 and UMIACS-TR-96-28, University of Maryland, College Park, MD.

Lindenmayer, A. (1968). Mathematical models for cellular interaction in development parts I and II. *Journal of Theoretical Biology*, 18:280–299 and 300–315.

Lindgren, K., and Johansson, J. (2001). Coevolution of strategies in n-person prisoner's dilemma. In Crutchfield, J., and Schuster, P., editors, *Evolutionary Dynamics - Exploring the Interplay of Selection, Neutrality, Accident, and Function*. Reading, MA: Addison-Wesley.

Lipson, H., and Pollack, J. B. (2000). Automatic design and manufacture of robotic lifeforms. *Nature*, 406:974–978.

Lubberts, A., and Miikkulainen, R. (2001). Co-evolving a go-playing neural network. In *Coevolution: Turning Adaptive Algorithms Upon Themselves, Birds-of-a-Feather Workshop, Genetic and Evolutionary Computation Conference (GECCO-2001)*.

Luke, S., and Spector, L. (1996). Evolving graphs and networks with edge encoding: Preliminary report. In Koza, J. R., editor, *Late-breaking Papers of Genetic Programming 1996*. Stanford Bookstore.

Mahfoud, S. W. (1994). Crossover interactions among niches. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, 188–193. IEEE Computer Society Press.

Mahfoud, S. W. (1995). *Niching Methods for Genetic Algorithms*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL.

Maley, C. C. (1999). Four steps toward open-ended evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, 1336–1343. San Francisco, CA: Morgan Kaufmann.

Mandischer, M. (1993). Representation and evolution of neural networks. In Albrecht, R. F., Reeves, C. R., and Steele, U. C., editors, *Artificial Neural Nets and Genetic Algorithms*, 643–649. Berlin; New York: Springer-Verlag.

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

Martin, A. P. (1999). Increasing genomic complexity by gene duplication and the origin of vertebrates. *The American Naturalist*, 154(2):111–128.

McQuesten, P., and Miikkulainen, R. (1997). Culling and teaching in neuro-evolution. In (Bäck 1997), 760–767.

Mengshoel, O. J. (1999). *Efficient Bayesian Network Inference: Genetic Algorithms, Stochastic Local Search, and Abstraction*. PhD thesis, University of Illinois at Urbana-Champaign Computer Science Department, Urbana-Champaign, IL.

Meuleau, N., Peshkin, L., Kim, K.-E., and Kaelbling, L. P. (1999). Learning finite-state controllers for partially observable environments. In *Proceedings of the Fifteenth International Conference on Uncertainty in Artificial Intelligence*.

Miller, G., and Cliff, D. (1994). Co-evolution of pursuit and evasion i: Biological and game-theoretic foundations. Technical Report CSRP311, School of Cognitive and Computing Sciences, University of Sussex, Brighton, UK.

Miller, J. F., Job, D., and Vassilev, V. K. (2000a). Principles in the evolutionary design of digital circuits – Part I. *Journal of Genetic Programming and Evolvable Machines*, 1(1):8–35.

Miller, J. F., Job, D., and Vassilev, V. K. (2000b). Principles in the evolutionary design of digital circuits – Part II. *Journal of Genetic Programming and Evolvable Machines*, 3(2):259–288.

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

Mitchell, M., Crutchfield, J. P., and Das, R. (1996). Evolving cellular automata with genetic algorithms: A review of recent work. In *Proceedings of the First International Conference on Evolutionary Computation and Its Applications (EvCA'96)*. Russian Academy of Sciences.

Mondada, F., Franzi, E., and Ienne, P. (1993). Mobile robot miniaturization: A tool for investigation in control algorithms. In *Proceedings of the Third International Symposium on Experimental Robotics*, 501–513.

Montana, D. J., and Davis, L. (1989). Training feedforward neural networks using genetic algorithms. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, 762–767. San Francisco, CA: Morgan Kaufmann.

Moriarty, D. E. (1997). *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin. Technical Report UT-AI97-257.

Moriarty, D. E., and Miikkulainen, R. (1994). Evolving neural networks to focus minimax search. In *Proceedings of the 12th National Conference on Artificial Intelligence*, 1371–1377. San Francisco, CA: Morgan Kaufmann.

Moriarty, D. E., and Miikkulainen, R. (1996). Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22:11–32.

Moriarty, D. E., and Miikkulainen, R. (1997). Forming neural networks through efficient and adaptive co-evolution. *Evolutionary Computation*, 5:373–399.

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

Nadeau, J. H., and Sankoff, D. (1997). Comparable rates of gene loss and functional divergence after genome duplications early in vertebrate evolution. *Genetics*, 147:1259–1266.

Noble, J., and Watson, R. A. (2001). Pareto coevolution: Using performance against coevolved opponents in a game as dimensions for parerto selection. In et al, L. S., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*. San Francisco, CA: Morgan Kaufmann.

Nolfi, S. (1997). Evolving non-trivial behavior on autonomous robots: Adaptation is more powerful than decomposition and integration. In Gomi, T., editor, *Evolutionary Robotics: From Intelligent Robots to Artificial Life*. Ontario, Canada: AAI Books.

Nolfi, S., Elman, J. L., and Parisi, D. (1990). Learning and evolution in neural networks. Technical Report 9019, Center for Research in Language, University of California, San Diego.

Nolfi, S., Elman, J. L., and Parisi, D. (1994). Learning and evolution in neural networks. *Adaptive Behavior*, 2:5–28.

Nolfi, S., and Parisi, D. (1993). Auto-teaching: Networks that develop their own teaching input. In Deneubourg, J. L., Bersini, H., Goss, S., Nicolis, G., and Dagonnier, R., editors, *Proceedings of the Second European Conference on Artificial Life*, 845–862.

Nolfi, S., and Parisi, D. (1995). Learning to adapt to changing environments in evolving neural networks. Technical Report 95-15, Institute of Psychology, National Research Council, Rome, Italy.

Opitz, D. W., and Shavlik, J. W. (1997). Connectionist theory refinement: Genetically searching the space of network topologies. *Journal of Artificial Intelligence Research*, 6:177–209.

O'Reilly, U.-M. (2000). Emergent design: Artificial life for architecture design. In *7th International Conference on Artificial Life (ALIFE-00)*. Cambridge, MA: MIT Press.

Pierce, D., and Kuipers, B. (1997). Map learning with uninterpreted sensors and effectors. *Artificial Intelligence*, 92:169–227.

Postlethwait, H. H., Yan, Y. L., Gates, M. A., Horne, S., Amores, A., Brownlie, A., and Donovan, A. (1998). Vertebrate genome evolution and the zebrafish gene map. *Nature Genetics*, 18:345–349.

Potter, M. A., and De Jong, K. A. (1995). Evolving neural networks with collaborative species. In *Proceedings of the 1995 Summer Computer Simulation Conference*.

Potter, M. A., De Jong, K. A., and Grefenstette, J. J. (1995). A coevolutionary approach to learning sequential decision rules. In Eshelman, L. J., editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*. San Francisco, CA: Morgan Kaufmann.

Pujol, J. C. F., and Poli, R. (1997). Evolution of the topology and the weights of neural networks using genetic programming with a dual representation. Technical Report CSRP-97-7, School of Computer Science, The University of Birmingham, Birmingham B15 2TT, UK.

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

Radcliffe, N. J. (1993). Genetic set recombination and its application to neural network topology optimization. *Neural computing and applications*, 1(1):67–90.

Radding, C. M. (1982). Homologous pairing and strand exchange in genetic recombination. *Annual Review of Genetics*, 16:405–437.

Raff, R. A. (1996). *The Shape of Life: Genes, Development, and the Evolution of Animal Form*. Chicago: The University of Chicago Press.

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

Reil, T., and Husbands, P. (2002). Evolution of central pattern generators for bipedal walking in a real-time physics environment. *IEEE Transactions on Evolutionary Computation*, 6(2):159–168.

Reil, T., and Massey, C. (2001). Biologically inspired control of physically simulated bipeds. *Theory in Biosciences*, 120:1–13.

Richards, N., Moriarty, D., and Miikkulainen, R. (1998). Evolving neural networks to play Go. *Applied Intelligence*, 8:85–96.

Ring, M. B. (1994). *Continual Learning in Reinforcement Environments*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712.

Rosin, C. D. (1997). *Coevolutionary Search Among Adversaries*. PhD thesis, University of California, San Diego, San Diego, CA.

Rosin, C. D., and Belew, R. K. (1997). New methods for competitive evolution. *Evolutionary Computation*, 5.

Rumelhart, D. E., McClelland, J. L., and the PDP Research Group (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*. Cambridge, MA: MIT Press.

Santamaria, J. C., Sutton, R. S., and Ram, A. (1998). Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior*, 6(2):163–218.

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

Sidow, A. (1996). Gen(om)e duplications in the evolution of early vertebrates. *Current Opinion in Genetics and Development*, 6:715–722.

Sigal, N., and Alberts, B. (1972). Genetic recombination: The nature of a crossed strand-exchange between two homologous DNA molecules. *Journal of Molecular Biology*, 71(3):789–793.

Sims, K. (1994). Evolving 3D morphology and behavior by competition. In (Brooks and Maes 1994), 28–39.

Singh, S. P., and Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158.

Spears, W. (1995). Speciation using tag bits. In *Handbook of Evolutionary Computation*. IOP Publishing Ltd. and Oxford University Press.

Spector, L., Goodman, E. D., Wu, A., Langdon, W. B., Voigt, H.-M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M. H., and Burke, E., editors (2001). *Proceedings of the Genetic and Evolutionary Computation Conference*. San Francisco, CA: Morgan Kaufmann.

Stanley, K. O., and Miikkulainen, R. (2002a). The dominance tournament method of monitoring progress in coevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002) Workshop Program*. San Francisco, CA: Morgan Kaufmann.

Stanley, K. O., and Miikkulainen, R. (2002b). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2). In press.

Stanley, K. O., and Miikkulainen, R. (2003). A taxonomy for artificial embryogeny. *Artificial Life*, 9(2):93–130.

Stanley, K. O., and Miikkulainen, R. (2004). Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21:63–100.

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

Terzopoulos, D., Rabie, T., and Grzeszczuk, R. (1994). Artificial fishes: Autonomous locomotion, perception, behavior, and learning in a simulated physical world. *Artificial Life*, 1(4):327–351.

Thierens, D. (1996). Non-redundant genetic coding of neural networks. In *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, 571–575. Piscataway, NJ: IEEE Computer Society Press.

Thomanek, F., Dickmanns, E. D., and Dickmanns, D. (1994). Multiple object recognition and scene interpretation for autonomous road vehicle guidance. In *Intelligent Vehicles Symposium '94*.

Thurrott, P. (2002). Top stories of 2001, #9: Expanding video-Game market brings microsoft home for the holidays. *Windows & .NET Magazine Network*.

Touretzky, D. S., Elman, J. L., Sejnowski, T. J., and Hinton, G. E., editors (1990). *Proceedings of the 1990 Connectionist Models Summer School*. San Francisco, CA: Morgan Kaufmann.

Ueda, N., and Ghahramani, Z. (2002). Bayesian model search for mixture models based on optimizing variational bounds. *Neural Networks*, 15:1223–1241.

Van Valin, L. (1973). A new evolutionary law. *Evolution Theory*, 1:1–30.

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

Watson, J. D., Hopkins, N. H., Roberts, J. W., Steitz, J. A., and Weiner, A. M. (1987). *Molecular Biology of the Gene Fourth Edition*. Menlo Park, CA: The Benjamin Cummings Publishing Company, Inc.

Whitley, D. (1995). Genetic algorithms and neural networks. In Periaux, J., Galan, M., and Cuesta, P., editors, *Genetic Algorithms in Engineering and Computer Science*, 203–216. John Wiley and Sons.

Whitley, D., Beveridge, J. R., Guerra, C., and Graves, C. (1997). Messy genetic algorithms for subset feature selection. In (Bäck 1997), 568–575.

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

Wieland, A. (1991). Evolving neural network controllers for unstable systems. In *Proceedings of the International Joint Conference on Neural Networks* (Seattle, WA), 667–673. Piscataway, NJ: IEEE.

Williams, R. J., and Zipser, D. (1989). Experimental analysis of the real-time recurrent learning algorithm. *Connection Science*, 1:87–111.

Willshaw, D., and Dayan, P. (1990). Optimal plasticity from matrix memories: What goes up must come down. *Neural Computation*, 2:85–93.

Wright, A. H. (1991). Genetic algorithms for real parameter optimization. In Rawlins, G. J. E., editor, *Foundations of Genetic Algorithms*, 205–218. San Francisco, CA: Morgan Kaufmann.

Yamauchi, B., and Beer, R. D. (1994). Integrating reactive, sequential, and learning behavior using dynamical neural networks. In Cliff, D., Husbands, P., Meyer, J.-A., and Wilson, S. W., editors, *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, 382–391. Cambridge, MA: MIT Press.

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

Yao, X., and Liu, Y. (1996). Towards designing artificial neural networks by evolution. *Applied Mathematics and Computation*, 91(1):83–90.

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

Yin, X., and Germay, N. (1993). A fast genetic algorithm with sharing scheme using cluster analysis methods in multimodal function optimization. In Albrecht, R. F., Reeves, C., and Steele, N. C., editors, *Proceedings of ANNGA93, International Conference on Artificial Neural Networks and Genetic Algorithms*, 450–457. Berlin: Springer-Verlag.

Zhang, B.-T., and Muhlenbein, H. (1993). Evolving optimal neural networks using genetic algorithms with Occam's razor. *Complex Systems*, 7:199–220.

Zigmond, M. J., Bloom, F. E., Landis, S. C., Roberts, J. L., and Squire, L. R., editors (1999). *Fundamental Neuroscience*. London: Academic Press.

Vita

Kenneth Owen Stanley was born in Boston, MA on May 16th, 1975 to Richard and Doris Stanley. He graduated from Newton South High School in 1993 and graduated Magna Cum Laude with a B.S.E. in Computer Science Engineering and a minor in Cognitive Science from the University of Pennsylvania in 1997. He enrolled at the University of Texas at Austin in 1997, where he received an M.S. in Computer Science in 1999. He spent the summer of 1999 working on neuroevolution at Hewlett-Packard Labs. In 2002, he won the Best Paper Award in Genetic Algorithms for his work on NEAT at the Genetic and Evolutionary Computation Conference (GECCO-2002). He has published papers in JAIR, Evolutionary Computation, and Artificial Life journals.

Permanent Address: 600 West 26th St. Apt. A208
Austin, Texas 78705 USA
kstanley@cs.utexas.edu
http://www.cs.utexas.edu/users/kstanley/

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.