ABSTRACT

Mark Ahlstrom. RECURRENCE AND PLASTICITY IN EVOLVED ADAPTIVE NEURAL CONTROLLERS. (Under the direction of Dr. M. H. N. Tabrizi). Department of Computer Science, December 10 2009.

Among the more important applications of evolutionary neurocontrollers is the development of systems that are able to dynamically adapt to a changing environment. While traditional approaches to control system design demand that the developer attempt to foresee all possible situations within which the controller may operate, neuroevolutionary approaches can facilitate the design of systems that are capable of operating in unforeseen circumstances. This paper examines two methods that have been used to provide for this adaptivity. The first method is the use of recurrent neural networks that have fixed connection weights. The second develops neurocontrollers with plastic synapses, thus allowing for the adaptation of the connection weights. Previous experimental results have shown that while both approaches can facilitate adaptive behavior, neural plasticity does not necessarily confer the expected benefits. In experiments using the NeuroEvolution of Augmenting Topologies (NEAT) method, Stanley et al. (2003) discovered that in simple cases, recurrence was sufficient in solving at least some control problems. I examined whether or not these initial results continue to scale upwards into more complex problem spaces. This was done through a series of experiments ranging from controlling a simplified cannon shot to attempting to evolve neural flight controllers capable of flying different airplanes through a series of waypoints. The results of these experiments indicate that the NEAT algorithm itself is unable to scale efficiently to some larger problem spaces.

RECURRENCE AND PLASTICITY IN EVOLVED ADAPTIVE NEURAL CONTROLLERS

A Thesis

Presented to

The Faculty of the Department of Computer Science

East Carolina University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Mark Ahlstrom

December 10 2009

RECURRENCE AND PLASTICITY IN EVOLVED ADAPTIVE NEURAL CONTROLLERS

by

Mark Ahlstrom

APPROVED BY:

DIRECTOR OF THESIS:

Dr. M. H. N. Tabrizi

COMMITTEE MEMBER:

Dr. Ronnie Smith

COMMITTEE MEMBER:

Dr. Karl Abrahamson

COMMITTEE MEMBER:

Dr. Mike Spurr

CHAIR OF THE DEPARTMENT
OF COMPUTER SCIENCE:

Dr. John Placer

DEAN OF THE
GRADUATE SCHOOL:

Dr. Paul J. Gemperline

TABLE OF CONTENTS

LIST OF FIGURES

1 Introduction

Among the primary goals of research into artificial intelligence is to develop systems able to operate effectively within the ever changing environs of the real world. While the ability to solve simple puzzles and win games with strictly defined rules is laudable, the well defined constraints that such problems present to the programmer make those sorts of problems well suited for computational solutions. However, real world control problems can provide computational systems with a particularly difficult challenge. The limitations of the system, as well as the appropriate solution may be poorly defined in a real world control problem. Furthermore, the correct solution to the control problem that is faced by the system may fluctuate along with changes to the environment that the system is in. The classic example of this sort of problem is the control of a robot, though there are other similar control problems that face this challenge as well.

The hallmark of these problems is the need for the control system to learn about its environment and adapt to changing circumstances. In order for a solution to a control problem to remain robust in the real world, it is absolutely imperative that the system itself is able to learn and adapt. If the onus is put on the developer of the system rather than the system itself, then the programmer must be able to foresee all possible environmental permutations during the design of the system. However, with the appropriate approach, the controller should be able to be designed in such a way that it is able to adapt to most circumstances as needed.

Because of the ease with which humans and other animals are able to navigate an ever changing environment, it is no surprise that among the most promising approaches to adaptive control systems are those based on biological observation. Evolutionary neurocomputing is a field within soft-computing that takes as its model the human brain, and attempts to approximately duplicate its method of development to create computational control systems. The general approach is, through a survival of the fittest process, to develop a computational system that works in a fashion similar to the way our brain works, rather than to develop a system that operates within the strict logical confines that traditional computational solutions provide.

An open question within the field of evolutionary neurocomputing is to how best to provide for adaptivity in neural network controllers. Traditionally, adaptivity in neurocontrollers has been provided by recurrent connections. However, recent experiments such as Urzelai and Floreano (2001) and Stanley et al. (2003) have explored the use of plastic or adaptive synapses in evolved neurocontrollers. While intuitively it would seem that adaptive synapses would better promote adaptive behavior than simple recurrence, experimental evidence seems to suggest that the desirability of either approach is dependent on the specifics of the problem domain. It is an examination of this specific question with which this thesis is concerned.

Evolutionary neurocomputing is a field that results from a combination of research into several other disciplines in artificial intelligence. This approach applies evolutionary computation techniques to the development of artificial neural networks. The thesis begins with an examination of evolutionary computing and artificial neural networks before examining the specifics of whether recurrence or plasticity better promotes adaptive control systems.

1.1 Outline of the Paper

This thesis is divided into three primary sections: Background (Chapter 2 and 3), and a discussion of the experiments and results (Chapters 4, 5, 6) and an examination of the conclusions that can be drawn from those results.

Chapter 2 provides the theoretical background for the approach taken in this thesis. It begins by detailing how evolutionary methods are commonly used. It then moves on to a detailed discussion of both traditional neural networks as well as current directions in neurocomputing. Finally, it discusses the general approach taken when combining the two fields in neuroevolutionary computing before addressing the specific approach taken with the NEAT software used for this thesis.

Chapter 3 examines the previous experiments that have compared recurrent and plastic networks. It begins with an exploration of the initial work by Floreano and Urzelai into plastic neural networks. From there, it examines the work done by Stanley et al. using the NEAT software.

Chapter 4 describes the experimental environments of the experiments used to confirm the functionality of the neural evolution libraries. This will also provide some analysis of the results from those confirmation experiments.

Chapter 5 explains the flight control experimental environment itself. It first gives an overview of the general environment for the flight simulation as well as the sorts of tests performed on the aircraft. It then discusses the specifics of the way that the aircraft is tested and evaluated within the simulated environment.

Chapter 6 presents the results of the flight control experiment and compares those results to the results of the previous work in the area of plasticity and recurrence.

Chapter 7 then examines those results and draws its conclusions based on them. Finally, it proposes areas where further exploration may prove fruitful.

The remainder of this chapter will give a high level overview of evolutionary neural-computing and the experiments implemented to test the scalability of this approach.

1.2 Evolutionary Computing

Evolutionary computing refers to a whole range of specific algorithmic techniques that are founded in the theory of biological evolution. It includes research into fields such as genetic algorithms, evolutionary programming, and evolution strategies. Rather than being a specific algorithmic solution to a problem, it can be understood as a general concept to be used as an approach towards solving many problems. It is particularly well suited to solving difficult optimization problems.

In general, evolutionary approaches to optimization can be understood as an efficient randomized search. The process is intended to mimic that of natural selection. A set of possible solutions is either provided or randomly generated. This set is known as the population. The members of the population are then evaluated for their fitness as a solution. The better the fitness of a member of the population, the more likely it is that it will survive through the next generation. The population is then pruned of members with poor fitness, and new members are added that are permutations of the original population members. This process is repeated until a member is found with the appropriate fitness or a specified number of generations is completed. Different approaches are explained more completely in **Chapter 2**.

1.3 Artificial Neural Networks

Artificial Neural Networks (hereafter known as ANN's) similarly take their cue from the natural world in modeling an algorithmic approach. ANN's provide a model of computation based on the scientific understanding of how the brain functions.

1.3.1 Recurrence and Plasticity

The two methods most often proposed to provide adaptivity in advanced neurocontrollers are recurrency and plasticity. Recurrent neural networks contain nodes that contain a link that feeds the output of the node back into the artificial neuron. Plasticity involves the use of adaptive synapses which can adjust their connection weights while in a running network.

1.3.2 The Problem

It is hypothesized that plastic synapses aid in situations where there is a graduated change that the network must adapt to while recurrency is more than adequate when the network adapts by changing modes of search. For example, plastic synapses would better deal with adapting to fluctuating lighting circumstances, while simple recurrent networks would be better able to deal with switching a controller from being the hunter to being prey in an artificial life simulation. While these sorts of results have been born out in simple experiments, it is untested on more complicated control problems. Thus, the aim of this thesis is to generate and test a number of recurrent, and plastic neurocontrollers in several experimental domains. This will help assess the scalability of the results seen in Stanley et al. (2003).

1.4 Organization of the Experiments

To test scalability, several experiments were devised and implemented. Details of each of these experiments will be developed later in the paper. The first three experiments were designed and run to confirm the functionality of the NEAT implementation and to establish a baseline of functionality. The flight controller experiment was designed to test the upwards scalability of recurrent and plastic networks.

1.4.1 Cannon control

The simplest experiment domain was a cannon controller. The goal of this experiment was to evolve a controller capable of controlling the angle and firing power of a cannon with simplified physics. A successful controller would be able to hit a target at a given range with a limited number of iterations modifying the angle of fire and power of the shot. This experiment was created primarily as a verification that the neural system was functioning properly.

1.4.2 Food Foraging

The food foraging problem is a simplified version of the experiment detailed in Stanley et al. (2003). The goal in this version of the experiment is to develop a controller of a simplified robot with sensors that is capable of finding food items scattered in a two dimensional world. While in theory plastic neurons should not provide an advantage in this domain, it is still useful to test with both to establish a baseline.

1.4.3 Dangerous Food Foraging

This is identical to the food foraging experiment except that in half of the experimental instances, the robot will be in a world with poisonous food. This is a re-implementation of the experiment that inspired this thesis. This should benefit from adaptive neurons, but has experimentally been shown to be solvable simply with recurrent networks.

1.4.4 Flight Control

As the goal of this thesis is to examine the scalability of NEAT, it is also necessary to develop a more complicated control problem. For this, a flight control system was used. While a fully functional flight controller is not necessary, it is necessary to use an adequately complex simulation. For that reason, an open source simulator was used that is often used in the aerospace engineering domain. This presents a controller with a sufficiently complex environment. At the same time, only a limited set of the controls was exposed to the controller to avoid requiring it to learn behavior for some edge cases. Furthermore, overall success or failure of the controller is not as important as a comparative analysis of the relative utility between recurrent and plastic controller models.

2 Foundations

This chapter will provide an overview of the key concepts that this thesis is built upon. Firstly, an understanding of what neural networks and evolutionary algorithms are and how they work is essential to an understanding of the flight control experiment. After discussing those two approaches in general, the next section will examine specifically how the two are implemented in the NEAT approach.

2.1 Artificial Neural Networks

Artificial Neural Networks are connectionist systems that model themselves after the way that a brain functions. Rather than an emphasis on logically based symbol manipulation as embodied in Good Old Fashioned Ai (GOFAI), neural networks instead focus on providing representative inputs and outputs to a system and allowing computational approaches to determine the appropriate correlation between the input and output space. While traditional AI systems require the insight of the developer as to how the input and output of a system are related,[1] neural network places that burden on the training of the network, The only insight required of the designer is some conception of a desirable end result. This means that important invariances in the problem space that escape analysis may well be captured by the ANN.(Luger, 2001)

In the case of control problems, the ability of ANN systems to discover and deal with a noisy, changing environment is essential. Carefully tailored ANN's have proven effective at producing control systems that can deal with automated takeoff for unmanned vehicles in Thomson et al. (2004), complex control of a helicopter in vertical flight in Pallett and Ahmad (1993) adaptive flight control in Ferrari and Stengel

[1]Examples here would be things like an expert system or a minimax solution to the game. While the programmer may not know the solution, they must have in depth knowledge of how different relationships within the solution space function, and implement a solution based on that insight.

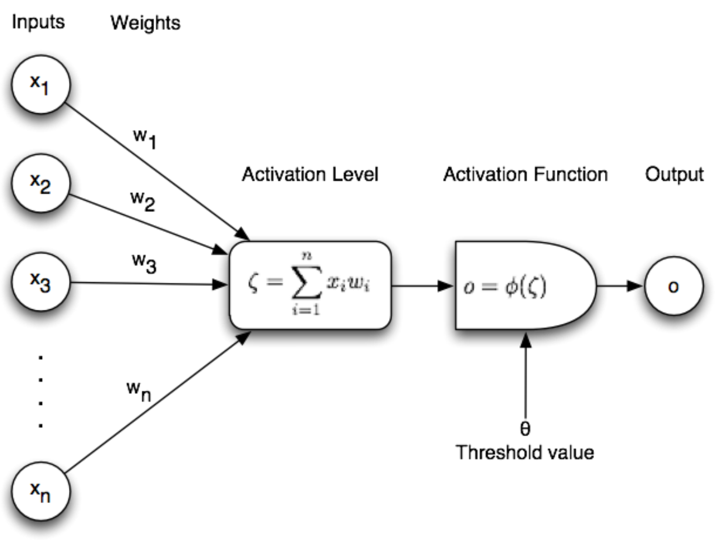


Figure 2.1: Diagram of the five parts of an artificial neuron.

(2004), and visual flight control in Beyeler et al. (2009). What each of these has in common is that they use neural networks to provide a robust system capable of dealing with unforeseen environmental changes. This has been born out in experimental work on neural networks. Noriega and Wang (1995) demonstrated that a neural network could be developed capable of efficiently adaptively controlling a flow rate control system, while Ge et al. (1997) offers a proof that an adaptive neural network can control at least a generalized set of nonlinear systems.

2.1.1 Artificial Neurons

The artificial neuron is the building block of all neural networks. Each neuron functions as a single node in the ANN system. The neuron consists of five basic parts as seen in Figure 2.1: the input layer, weights associated with each input, the activation level, the activation function, and the output. Some neurons additionally make use of a threshold value to modify the activation function. The details of each of these components is discussed below.

Input Layer

The input values for an artificial neuron are values that come either from the environment or from the output values of other artificial neurons. Depending on the activation function being used, these values may have their ranges constrained to discrete sets of values or to limited ranges of real numbers. In the case of the inputs from the experimental world, the design of the experiment defines the meaning of what each input value is. Thus, the neuron in Figure 2.1, used in a pattern recognition domain, x_1 may represent the greyscale value of the first pixel in an image,[2] while in a control problem x_1 may be the output value of the first sensor.

Each of these input values also has an associated weight. This in effect describes the importance of any input to the neuron. The value of the weight is part of what describes the neuron itself. How the weights affect the final output varies with what activation function is being used. It is important to note here the importance of the associated weights in the functioning of an ANN. Modifying the weights of inputs to neurons is fundamentally how most neural networks are tailored to function to solve a particular problem.

While the weights of a neural network are usually fixed by the training algorithm, it is also possible to have weights that can be changed in a running network. The general ability for a running neural network to change its operation is referred to as its plasticity. A neuron that can change its weight while running is referred to as a plastic neuron. Typically, this is implemented through the use of a Hebb rule (Floreano and Urzelai, 1999). The idea of Hebbian updates is that if one input node fires often, its weight is increased. Figure 2.2 demonstrates this. On the first iteration of the neuron, everything functions as normal. However, on the second iteration, each of the

[2]In this case, the assumption would be that the image is a vector of greyscale values that, if reworked to be arranged as an array, would produce the image when properly rendered

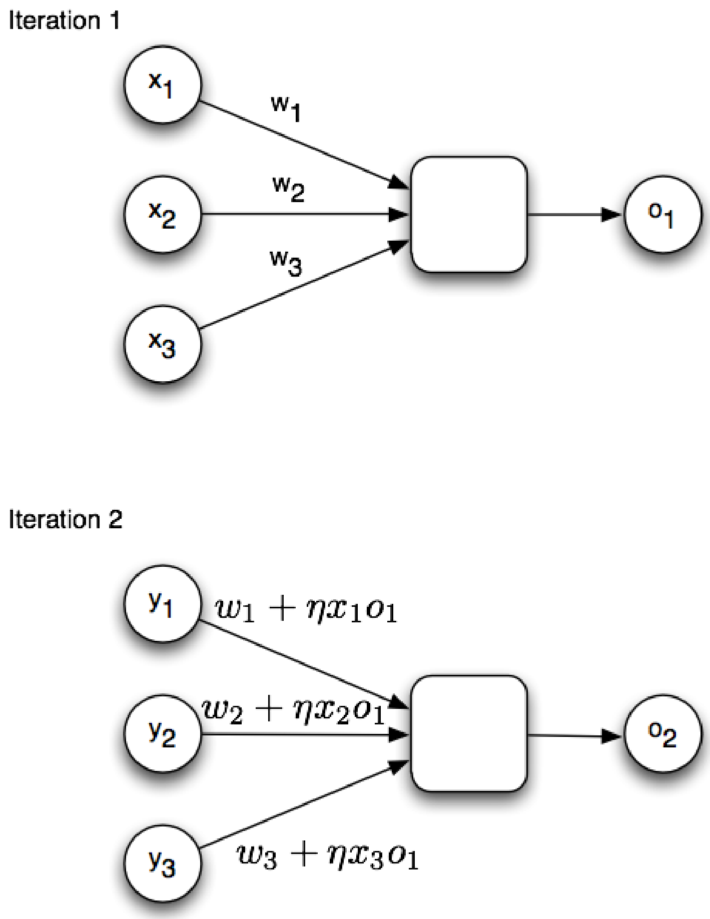


Figure 2.2: Hebbian weight update of a neuron.

weights associated with an input is modified by the values of the last iteration. The Hebb rule itself in this case is that $\delta w_n = \eta x_n o$ where η is a step size that determines by what percentage the values of x_i o modify the weight of w_i (Principe et al., 1999). Thus, if each weight starts off equal in iteration 1 and x_2 has the largest value in that iteration, then in iteration 2, x_2 will have the greatest weight.

Activation Level

The activation level ζ is determined by summing the weighted input values.

$$\zeta = \sum_{i=1}^{n} x_i w_i \tag{2.1}$$

This gives the value that is then fed to the activation function ϕ. Almost any function can in theory be used as an activation function, though there is a small set of functions that tend to be used. The next section will cover some of the more commonly used activation functions. Some activation functions take as a parameter a threshold value θ. This is particularly true when an activation function will produce only a discrete pair of values. Values over θ will be mapped to one output value while those under would be mapped to the other output value.

Activation Functions

Activation functions are the decision making core of a neural system. The activation function determines what value is passed from the input nodes to the output. As such, how it functions defines much of the behavior of an ANN system.

The most basic activation function is a simple step function. This sort of activation function was used in some of the earliest neural networks such as the McCulloch-Pitts neurons and simple single layer perceptrons (Luger, 2001). For example:

$$o = \begin{cases} 1 & \text{if } \zeta \geq \theta, \\ -1 & \text{if } \zeta < \theta \end{cases} \tag{2.2}$$

These sorts of step functions can be made more complicated by adding more cases and possible output values, but most often is implemented in a fairly straightforward

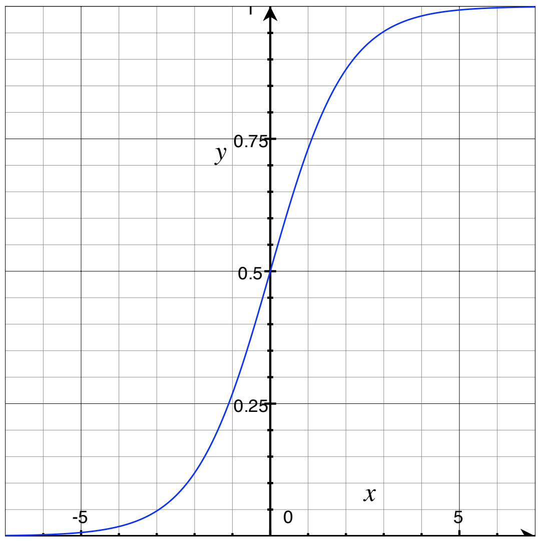


Figure 2.3: Default Sigmoid activation function curve

manner. This does limit all output nodes to a discrete, defined set of values.

The more common activation function is the nonlinear sigmoid function $\frac{1}{1+e^{-\zeta}}$ (Stanley, 2004). This produces the activation curve seen in Figure 2.3. The shape of this curve can be changed by either multiplying or adding the threshold value θ to the function. Thus, if one wanted a larger range of output value from 0 to θ, the activation function could be $\frac{\theta}{1+e^{-\zeta}}$. Other mathematical manipulations are possible to vary the shape of the curve as well. The steepness of the sigmoidal can also be changed by multiplying the activation value by the threshold as in $\frac{1}{1+e^{-\theta\zeta}}$. Figure 2.4 shows that with θ values as small as 10, this begins to resemble the step function above, allowing the sigmoid to effectively handle all cases that would otherwise be handled by a step activation function.

Linear functions, the sine function, and Gaussian functions[3] are other commonly used activation functions, though they are not specifically addressed in this thesis.

[3]A Gaussian function can also be referred to as a radial basis function

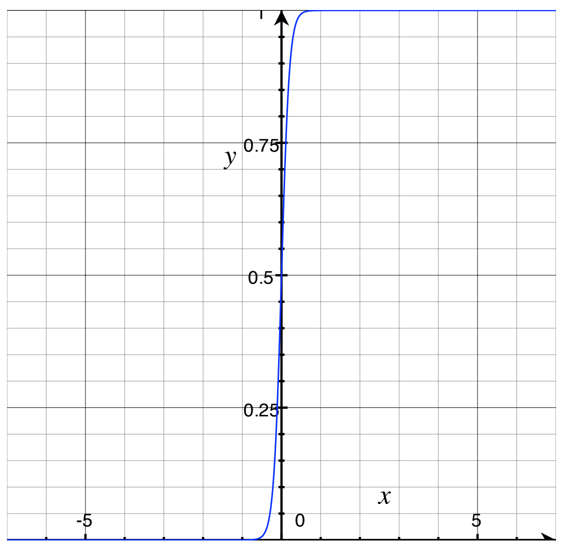


Figure 2.4: Sigmoid activation function with large threshold

Output Layer

The output layer uses the value computed by the activation function as the output of the neuron. This output can either be passed on to another layer of neurons, or be provided as part of the output of the entire network, depending on the location of the neuron in the overall network topology. In the case that the node is an output node for the network, the meaning of the output value is determined by the design of the experiment, similar to how the meaning of an input node is determined.

2.1.2 Topologies

In addition to how individual nodes function, the interrelationships of those nodes are important in the function of the network. The organization of a neural network is its topology. The topology has effects on both the performance and the function of a neural network. While neural networks are parallelizable, different topologies may limit that advantage by introducing timing dependencies. Furthermore, the topology chosen can effect how the network is able to learn the desired function (Emmert-

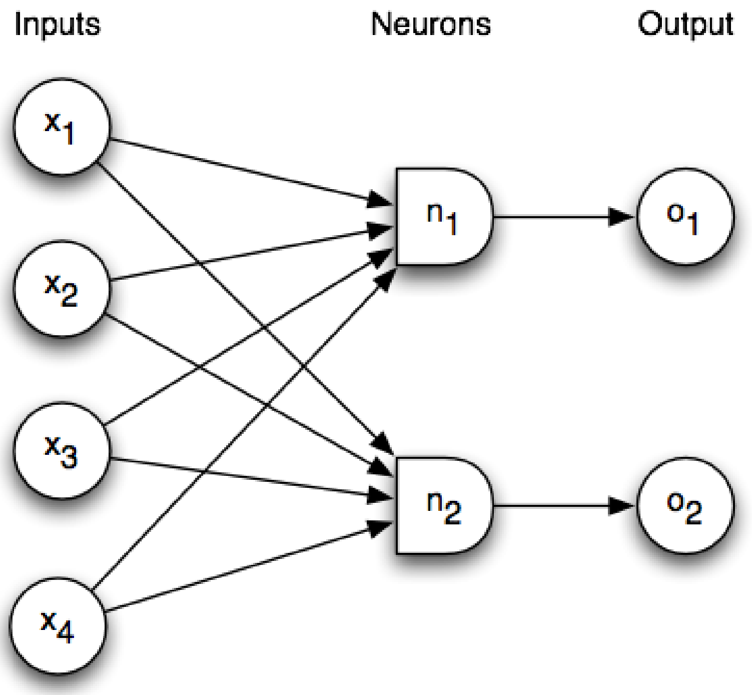


Figure 2.5: Single Layer Neural Network

Streib, 2006).

Until recently, most research in neural networks for control has used one of three kinds of standardized network topologies: A single layer feedforward network, a multi-layer feedforward network, or a multi-layer recurrent network. This has allowed some detailed work on the advantages and limitations of those architectures. There are also many other potential topologies that have been developed such as Kohonen self-organizing maps, or Hopfield networks that fall outside of the scope of this thesis. (for more, see Luger, 2001; Principe et al., 1999)

Single-Layer Network

The most basic topology for a neural network is the single layer architecture. As Figure 2.5 shows, the single layer network has an input layer of neurons that feed inputs to a set of neurons to produce the desired output. Each input is connected once to each output neuron, and has a weight associated with that connection.

While this is a general description of a single layer network, it is classically implemented as the single layer perceptron. In reality, Figure 2.5 is two single layer perceptrons that happen to take the same set of four inputs. Rojas (1996) (explaining Minsky and Papert (1988)), demonstrates the limitation of these single layer perceptrons, each is only capable of computing a linearly separable function.

Some of these limitations can be addressed by using a more complex activation function. Auer et al. (2007) demonstrate that if the computation of the activation function is appropriately modified, a single layer can function as a universal approximator.

Multi-Layer Network

The general limitation of the standard single layer network led to the development of the multilayer neural network topology. While Minsky and Papert (1988) demonstrated that single layer networks were limited to linearly separable functions, simply adding more layers of neurons increases the computational power enough to be able to overcome this limitation. Cybenko (1989) offers a proof that a multilayer neural network is powerful enough to approximate any continuous function.

As seen in Figure 2.6 , at least one layer of hidden network nodes is added between the input and output neurons to add additional computational complexity. This layer takes as input an input vector, and passes its output values to the next set of nodes. Multilayer networks can have any number of hidden layers and nodes in those layers. Principe et al. (1999) and Cybenko (1989) leave open as a question how many hidden nodes are necessary for a multilayer network to approximate a given function. It can be demonstrated that an inadequate number of hidden nodes will prevent a

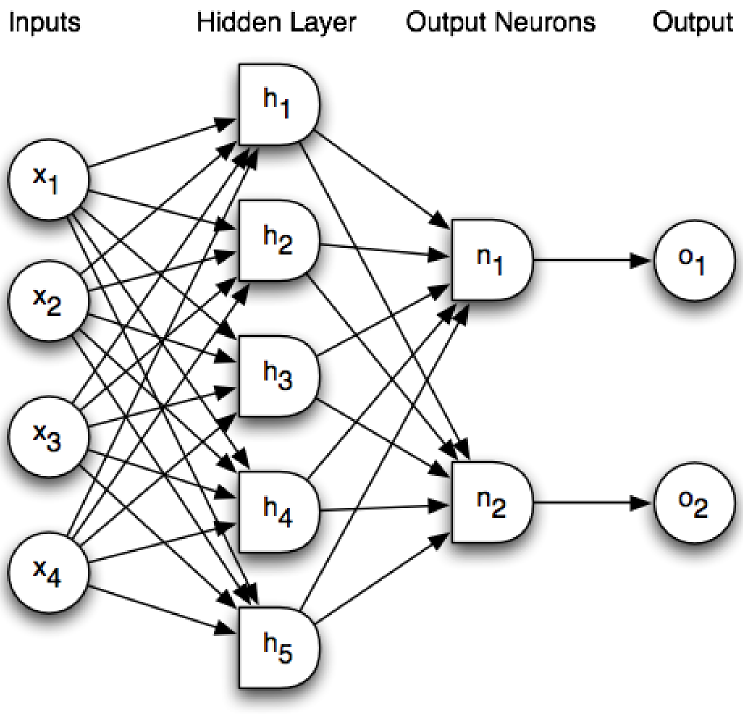


Figure 2.6: Multi Layer Neural Network

solution from being found (see Principe et al., 1999). Daqi and Shouyi (1998) shows that too many hidden nodes will also hamper finding a solution.[4] This means that experimentation is needed to find the necessary number of hidden nodes for any given problem.

Recurrent Neural Network

While a multilayer neural network is capable of approximating any continuous function, it still faces several limitations. In the domain of controller design, a significant limitation is that the network is only aware of its current state. This results because the network activity is uni-directional. It flows from the inputs, through the network,

[4]While too many hidden nodes will make it more difficult to find a solution, in theory this should just add to the compute time. The solution should still eventually be found. However, too many hidden nodes may pragmatically make a solution unfindable.

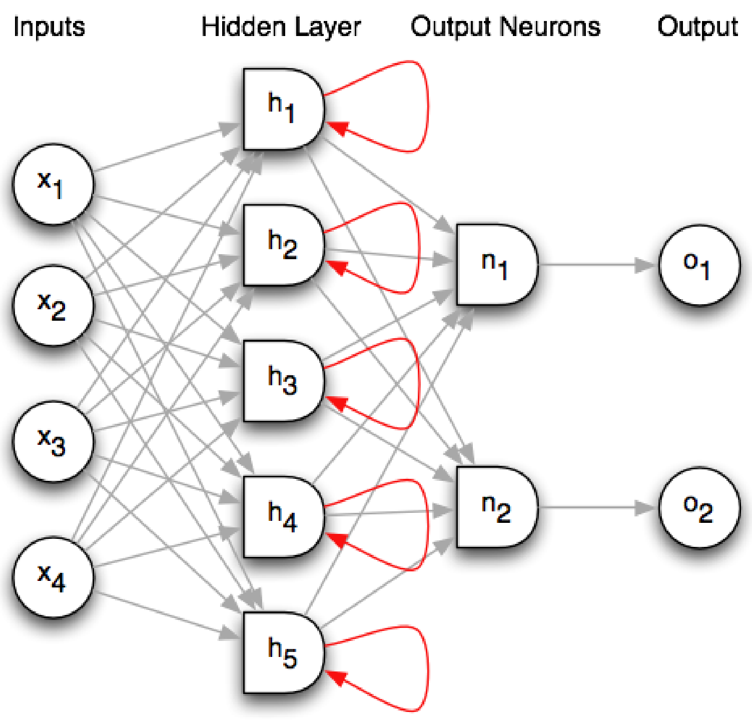


Figure 2.7: Recurrent Neural Network

to the output nodes. Principe et al. (1999) describes how the addition of connections that move data backwards from the normal flow of information (see Figure 2.7) functionally adds memory and the ability to learn temporal dependencies. Networks that have added backwards flow of information are called recurrent networks. Of note, while the addition of recurrent connections does make a network capable of dealing with new sorts of dependencies, it also makes it vastly more complicated. In a fully recurrent network with a single hidden layer with n nodes[5], this adds n^2 new connections to the network. Additional hidden layers and nodes lead to further compounding of complexity.

[5]Each node in the hidden layer feeds back into all of the nodes in the hidden layer

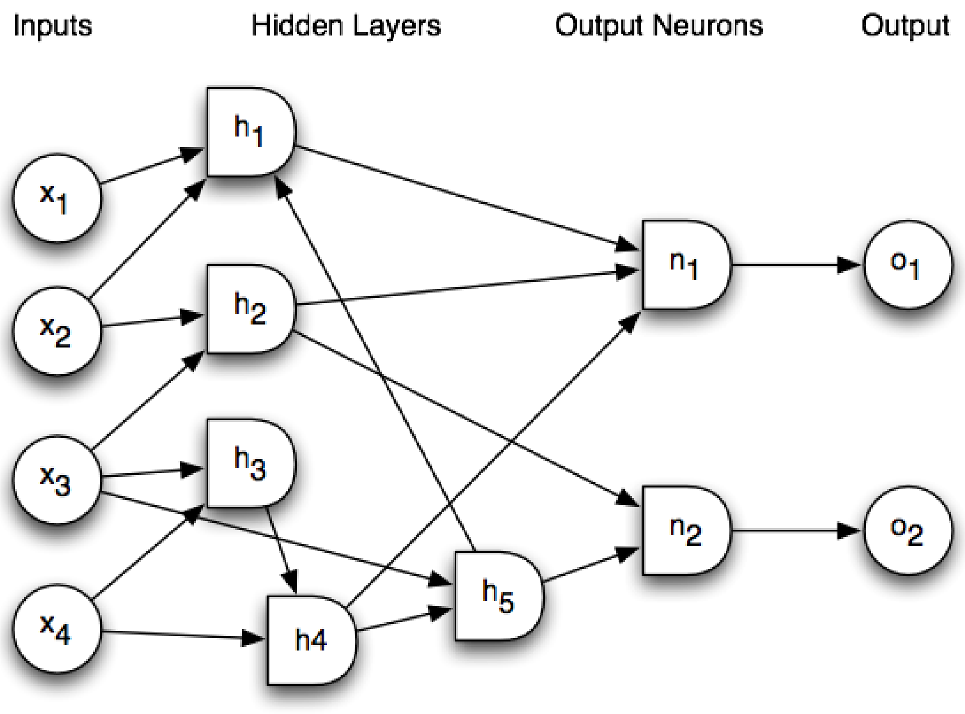


Figure 2.8: Nonstandard Neural Network

Nonstandard architecture

Each of the network architectures we have examined thus far is a fully connected network. Fully connected networks however present a variety of problems. Attik et al. (2005) enumerates these as including performance issues in both training and running networks, obfuscation of understanding the functioning of the network, and overtraining of the found solution. Each of these is a significant problem for an ANN. The first two problems are the result of extra nodes in the network that may be unused. Because of this superfluous structure, the performance is hampered and it becomes to discern how the network is functioning. In the case that overtraining is a problem, it is because all nodes potentially have some use, and end up impacting the solution when they should not.

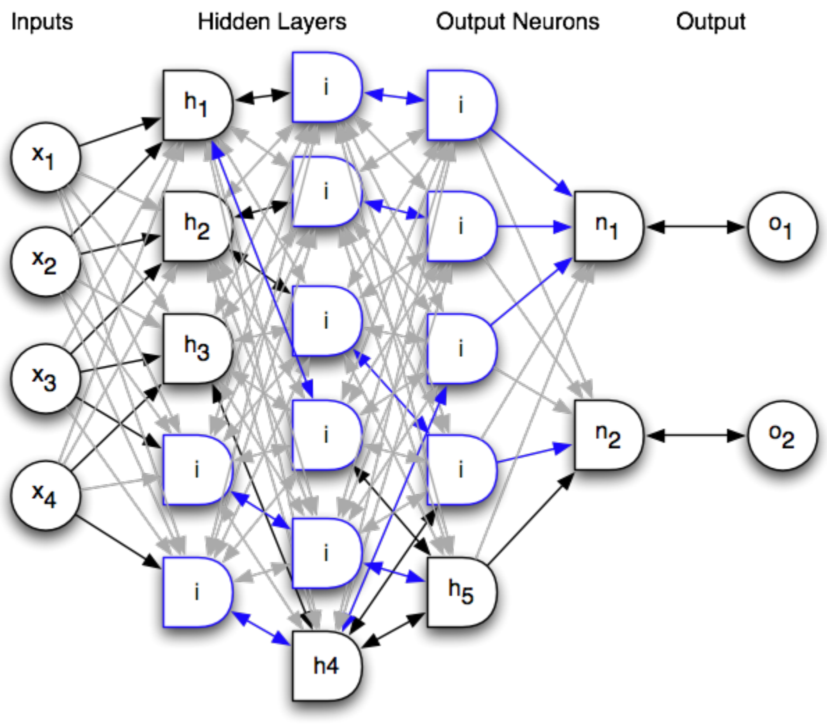


Figure 2.9: Identical functionality to nonstandard network in a full network. The dark nodes and connections are the originals, while the blue connections and nodes are those that are added and used. The grey connections are those that are added, but must be set to a 0 weight.

To solve these problems, a nonstandard architecture may be used. Figure 2.8 shows generally how the nonstardard architecture may work. Rather than the network being a regular graph, connections between the nodes are only included if they are advantageous to the end solution. Furthermore, as can be seen, it is difficult to even think of such a structure in layers. The longest path through the network passes through 5 nodes, while the shortest only passes through 2 nodes. There is one recurrent link going from h_5 to h_1, though h_5 also directly feeds into output neuron n_2. While it is possible to design a fully connected network that operates in the same way by using 0 weighted connections, this would add at least 11 nodes and over 50 more connections to the network. Figure 2.9 shows how complicated a traditional neural network must be to duplicate the functionality seen in the small network of Figure 2.8. Of note is that this is only the minimal representation of the functionality of that network. It is quite possible that the designer of the network may have included additional, unused nodes or hidden layers.

Network Topology Optimization

There are a number of ways to potentially optimize the architecture of a network. Daqi and Shouyi (1998) propose an empirically verified equation for determining the minimal number of needed nodes in a standard, two layer feedforward network. This equation still leaves the problem of connections whose weight is set to 0. Furthermore, it seems incapable of similarly determining the minimum number of nodes for a recurrent network.

There are other, more efficient approaches to network optimization. Attik et al. (2005) separates these into three classes of optimization. First are network growing techniques. This sort of network optimization starts only with the input and output nodes with no hidden layer. It then gradually adds nodes and connections between nodes until some set performance parameter is achieved. The second class is network pruning. In this approach, a solution is sought in a fully connected network, and then redundant and unused nodes and weights are removed from the network. This improves only the run time efficiency of a network, not the training efficiency. Finally, there are regularization techniques. These add a penalty in the training phase of the network to extra topological complexity. This means that the solution that is converged upon should be both capable of solving the problem and of having a low complexity. Attik et al. (2005) notes that none of these methods guarantees an optimal solution, and argues that to approach an optimal architecture, it is necessary to use a hybrid approach using all three techniques.

2.1.3 Learning Algorithms

Once the kinds of neurons used and the architecture of the network have been selected, it is still necessary to teach the network the function that it is intended to approximate. This can be done in a variety of ways. Each method of training the network to perform a function is a different learning algorithm.

Supervised learning and the Backpropagation Algorithm

Early work with neural networks used a class of learning algorithms called supervised learning. Supervised learning uses a set of training data to modify the neural network until it produces an acceptable output. Important to this process is the availability of a training set of data. The training set is a set of input output pairs. Each input value has an associated expected output value for the network.

The paradigmatic example of a supervised learning algorithm for a neural network is the backpropagation algorithm. Backpropagation is generally designed to be used with feedforward networks.[6] The network may be single or multilayer, though in the interests of computational power is usually a fully connected multi layer network. Backpropagation is best used in classification problems or a problem that has a known output function (such as training a controller to perform an xor operation). As with all supervised learning algorithms, it requires a representative set of training data.

Backpropagation functions by taking a given input and computing the output value for that input value. Then, comparing this with the expected output value, we generate set of error values for the weights in the ANN. Finally, we adjust the weights of the ANN according to the error values. This process is then repeated until a solution is converged upon.[7]

[6]Backpropogation can be adapted to recurrent networks. This takes some additional work however. See Pearlmutter (1996); Principe et al. (1999)

[7]For fuller explanations, including the equations for computing errors and weight updates, see

Despite the pervasiveness of backpropagation learning, it is often unsuited to application in the domain of control problems because of the requirement of a fairly comprehensive set of training data [8]. This training data must encompass both the input and output values, and thus nullifies the supposed advantage an ANN has for control problems: discovering the appropriate output values of a network for a given input. Furthermore, backpropagation tends to run a high risk of learning only to deal with the training data, hence the need for a diverse set of pairs which represent nearly every class of input output pair. Even with a comprehensive data set, it is possible for the network only to learn how to deal with the specific data in the training set and not be capable of dealing with a generalized case. This problem is called over-learning. Finally, when applied to recurrent networks, adapted backpropagation training can be prohibitively slow. (Stanley, 2004)

While there are other supervised learning algorithms in use, they share with backpropagation the need for highly representative data sets, and carry with them all of the drawbacks mentioned in the case of backpropagation.

Unsupervised Learning

An alternative to supervised learning that avoids some of its associated drawbacks is unsupervised learning. As indicated by the name, unsupervised learning lacks data fitting in the supervisory role. The only data available to the training algorithm is the input data set. The learning in an unsupervised algorithm functions by discovering statistical regularities and invariances present in the input data and modifying the

Luger (2001); Principe et al. (1999)

[8]Backpropagation can however be effectively used in control problems as in Hagan and Demuth (1999). This sort of usage often requires a good understanding of how the controller should function in the first place. It also requires a large set of training data. It is difficult to make a controller that is capable of generalizing its solutions outside of whatever the limits of the training data were, as this training data still must have both the input and expected output

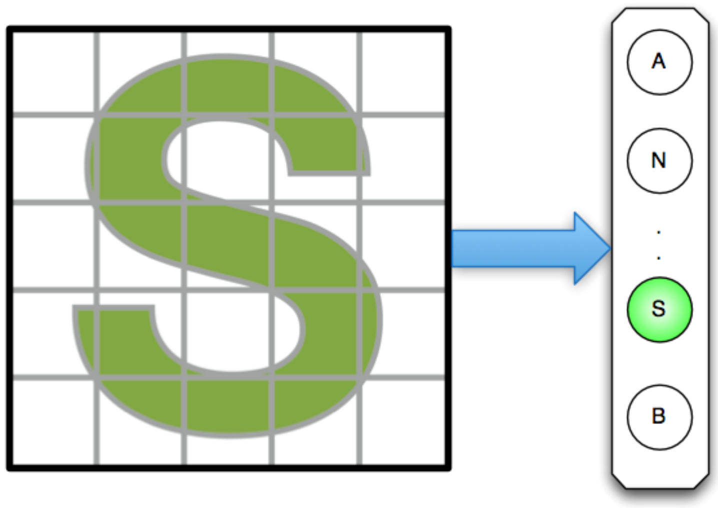


Figure 2.10: Kohonen SOM for character recognition.

network in accordance with those regularities.

The most common unsupervised learning algorithm is the Kohonen self organizing feature map (known as a SOM) (de Castro, 2007). The Kohonen network functions by mapping the input space to an n-dimensional array of nodes. For example, if attempting to perform character recognition, each letter could be placed into a 5x5 grid as seen in Figure 2.10. The percentage of each grid square that is filled can be used as the value of each input value. If the network is intended to recognize capital letters, the SOM could then have 26 nodes, one for each letter. The network is then trained by passing a variety of letterforms in to the SOM. The training operates by essentially clumping similar inputs to nearby output nodes. Thus, the output node for the letter A may be near the output node of letter N, while S may be near B, since in each case, the values in the input space may be statistically related to each other. The network then classifies the input as belonging to the output class of whatever output node has the highest activation level. More details of the training algorithm can be found in Principe et al. (1999).

An important step after the development of a classifier system is that the researcher must provide some additional analysis of the output space. In the example of the alphabet SOM, all that the neural network approach would do in this case is classify the input space into 26 distinct classes. It is not however aware of the meaning of those 26 classes. The researcher would have to test to match output 1 with A, output 2 with N and so on. This may be done programmatically, but the matching step is not an intrinsic part of the Kohonen algorithm.

It should be apparent that this learning approach is difficult to apply to control domains. It is designed only to recognize invariances and relationships in the input data itself. While there are likely statistically significant relationships between input values[9], it is unlikely that these relationships determine the correct behavior of the controlled entity. There is little room in an SOM for making relevant decisions on the correct output value for different controls. Furthermore, since it functions as a winner take all network (meaning whatever node has the highest classification is the correct value) it makes mapping the output to controls very difficult. This makes a SOM unsuitable for our purposes of controlling an airplane.

It is also possible to develop an unsupervised network using only Hebbian learning. Recall that in Hebbian learning, the weight of a node is increased if its input is of larger value. Rather than modifying the value of the weight in each iteration of a running network, Hebbian learning can be used in each iteration of the training phase of a neural network. This is commonly done with only a single layer network (Principe et al., 1999) As with Kohonen SOM's, this approach to training a neural network can be very effective in certain problem domains like classification problems, but is significantly less effective when used in control domains. The reason for the failing is similar to that of the Kohonen map. Unsupervised learning focuses on picking up on

[9]This is especially true when the inputs are related sensors

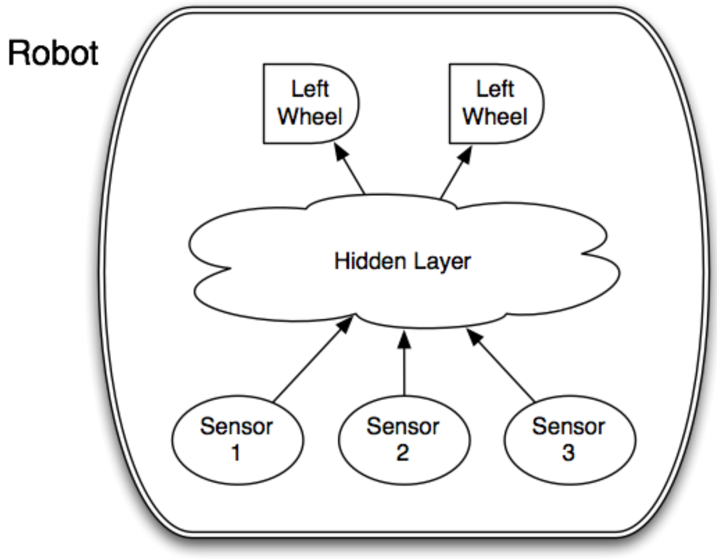


Figure 2.11: A simple robot controlled by a multilayer network. Inputs are mapped to sensors while outputs are mapped to the controlled parts of the robot

.

patterns present in the input space, but not on controlling an output space.

Reinforcement Learning

The final major class of learning algorithms, and likely the most useful in control problems, is reinforcement learning. Reinforcement learning is a broader category of algorithms than just those applied to neural networks. Reinforcement learning is based on actual interactions with the world. For this reason, it is ideally suited for control problems.

In the case of neural networks, reinforcement learning traditionally works by first mapping a set of inputs from the world to neural network inputs. For example, if controlling a simple robot with three sensors, each of those sensors would be mapped to a neural network input. It is possible that if the sensor outputs an array of data, it would be connected to an array of neural network inputs. The output nodes would then be mapped to the components controlled by the network. Thus, if the robot has two wheels, each output node could be mapped to the wheel control. An example of this sort of architecture can be seen in Figure 2.11.

The hallmark of reinforcement learning is that the network is rewarded when it takes actions that have desirable effects in the environment. The attempt then is to maximize this value. Any algorithm could potentially be used to maximize the reward value. In the neural networking domain, the most common reinforcement approaches are temporal difference learning and evolutionary algorithms (Konen and Bartz-Beielstein, 2009). Since neuro-evolution is the focus of this thesis, that is the reinforcement learning approach this paper will describe.

2.2 Evolutionary Computing

Before examining how evolutionary algorithms can be applied to neural network training, it is important to understand the general case of how evolutionary computing works. While neural networks are a computational model theoretically based on our understanding of how biological systems function computationally, evolutionary computing is an optimization technique theoretically based on our understanding of how biological systems optimize their design over time. In many ways, it can be thought of as an optimized randomized search.

Evolutionary algorithms are similar to simulated annealing (Kirkpatrick et al., 1983) in that it is a goal driven, randomized search. However, instead of starting with a random solution and then randomly tweaking its parameters to find better solutions, evolutionary approaches work on a set of potential solutions called the population.

2.2.1 Genetic Algorithm

Central to evolutionary computing approaches is the genetic algorithm. The general approach of a genetic algorithm is to create a random population of potential solutions, and then iteratively permute the possible solutions, improving only those that perform better in the environment. Poor solutions are often allowed to die off.

It is important here to make note of the capacity of a genetic algorithm to theoretically generate optimal solutions. Rowe (2001) offers proofs that as the size of the population tends towards infinity, the genetic algorithm will converge on an optimal fixed point in the population space. Furthermore, moderately sized populations will converge to a set of points near the optimal point found with an infinite population. Actual performance is dependent on the implementation of the algorithm.

The behavior of a genetic algorithm is defined by several components. These are the encoding structure, the initialization method, evaluation method or fitness function, selection criterion, and then mutation and recombination rules. Each of these is described below

Encoding

The encoding of solutions in a genetic algorithm is simply how a solution is represented. The representation chosen effects how the genetic algorithm operates as well as establishes the limits of the possible solution space.

The simplest encoding scheme to understand, and one of the most efficient for evolutionary algorithms, is to represent potential solutions as a binary string. A demonstration of this for solving the knapsack problem with 6 items can be seen in Figure 2.12. As can be seen, every possible solution to the knapsack problem can be represented by a six bit string. The encoding of a single solution is often also referred

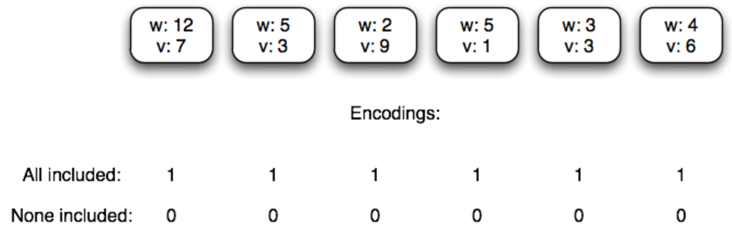


Figure 2.12: Simple encoding scheme for genetic algorithm knapsack problem optimization. Items represented by a 1 are included, and those with a 0 are not.

to as its genome.

While the knapsack problem is simple enough that all possible solutions can be efficiently represented, many more difficult problems elude such elegant representations of the solution space. In that case, there is often a trade off between the compactness and the completeness of the encoding method (Keane, 2001). Compact encodings are those that are small representations of the solutions[10]. Because genetic algorithms operate on the encodings, it is desirable to represent solutions in as efficient means as possible, as this limits the search scope of the algorithm. On the other hand, it is also important that there is sufficient completeness to the representation selected. By completeness, we mean that all of the possibly desired solutions can be represented by the encoding selected.

The example that Keane (2001) offers is attempting to optimize a jet aircraft wing. On one extreme, we could reduce the encoding to a single bit, specifying one of two possible wings. While this is a maximally compact representation, it clearly sacrifices too much completeness to be useful. On the other hand, we could represent the wing of a jet aircraft with a three dimensional array of points, with one point

[10]A compact representation in this case can mean several things. The important point about it is that a compact encoding presents only the information needed to solve the problem, and does so in the most efficient way available. In the knapsack problem, if we have 6 items, we could represent the solution as a list of those items that we want in the knapsack. This list would require 18 bits of information (3 bits for each of 6 items). Alternately, we could use a 6 bit genome with each bit indicating the presence of an item. The second representation would be more compact.

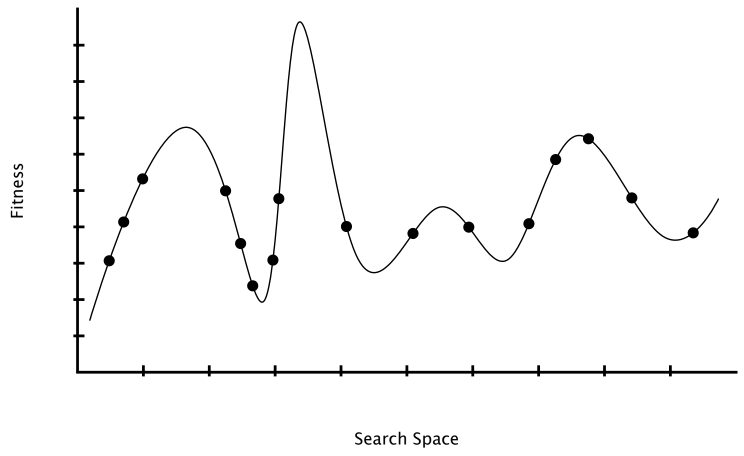


Figure 2.13: The distribution of the initial population across the fitness space should be well distributed to help ensure that the full spectrum of possible solutions can be evolved towards.

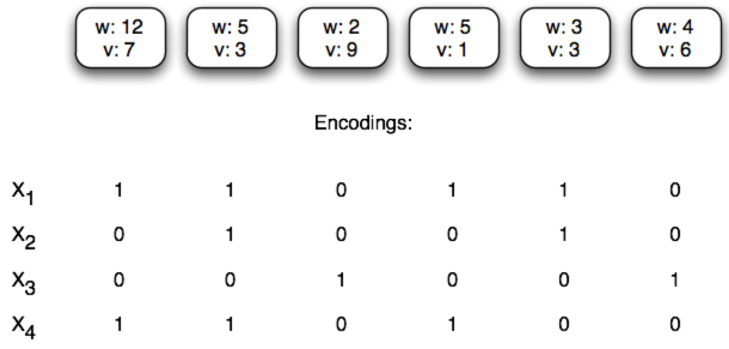
every square millimeter. This sort of representation may be able to capture virtually every possible wing shape, but creates such a difficult to work with solution object that the problem could well become nearly unsolvable.

While often the encoding is a string of numbers[11], other encodings can be used if appropriate to the problem. Alternate representations of solutions will require modifying how other elements of the genetic algorithm operate.

Initialization

There are a number of ways that an initial population may be created. One possibility is to start with all of the genes set to an identical initial condition. Thus, in the knapsack problem example, an initial population could consist of a set of only empty knapsacks. While this may be appropriate for some problems, more often it is desirable to have the population distributed relatively evenly throughout the potential

[11]In this case, it is most common to use a string of bits for simplicity and efficiency. This representation can both make operations on the artificial genome more efficient as well as making it easier to understand.



x_1	1	1	0	1	1	0
x_2	0	1	0	0	1	0
x_3	0	0	1	0	0	1
x_4	1	1	0	1	0	0

Figure 2.14: Initial population for knapsack problem.

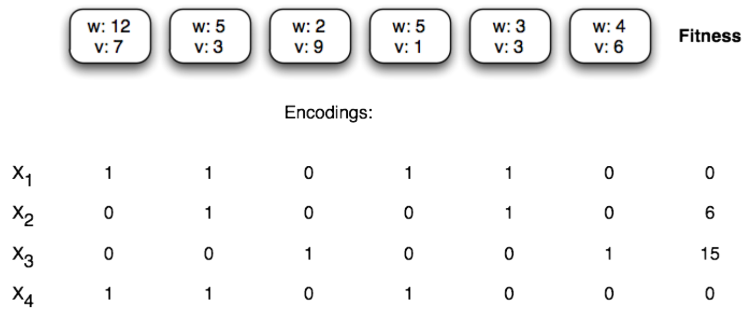
solution space as seen in Figure 2.13

An alternative that is often used is to start with a population of random solutions. This can be particularly effective when the size of the genome is known. This gives a population like that seen in 2.14. There are four potential solutions represented in that initial population, each having random settings for each portion of the genome. This has the advantage that provided the random genomes are well distributed throughout the potential solution space. This can potentially avoid solutions getting trapped in a local maximum.

Population initialization methods may be dependent on the representational scheme used for the genes of potential solutions. Clearly, if the generic representation of a potential solution is not a string of bits, initializing the population using random bit strings is inappropriate. However, the same principle can be applied by initializing the genes to random values appropriate to the representational scheme chosen.

Evaluation

Once the population has been created, the iterative process that drives the genetic algorithm begins. Each individual in the population is evaluated to determine the quality of the solution. This evaluation is done using what is known as the fitness



	w: 12 v: 7	w: 5 v: 3	w: 2 v: 9	w: 5 v: 1	w: 3 v: 3	w: 4 v: 6	Fitness
Encodings:							
x_1	1	1	0	1	1	0	0
x_2	0	1	0	0	1	0	6
x_3	0	0	1	0	0	1	15
x_4	1	1	0	1	0	0	0

Figure 2.15: Fitness of solutions in the initial population for the knapsack problem. This assumes that the desired maximum weight value is 15 See equation 2.3 for the fitness function.

function[12]. The fitness function is a problem specific function that measures the quality of the solution. Thus, in the case of the knapsack problem, the fitness function may be:

$$f = \begin{cases} 0 & \text{if } w_l < \displaystyle\sum_{i=0}^{n} w_i, \\ \displaystyle\sum_{i=0}^{n} v_i & \text{if } w_l \geq \displaystyle\sum_{i=0}^{n} w_i \end{cases} \tag{2.3}$$

In this case, f is the fitness of the proposed solution. $\sum_{i=0}^{n} w_i$ is the total weight of the items proposed to be included in the proposed solution. w_l is the maximum weight for the knapsack. Finally, $\sum_{i=0}^{n} v_i$ gives the total value of the items included in the knapsack for the proposed solution. Thus, in this case, the general way to think of the fitness function is that if the weight of included items is greater than the maximum weight for a solution the fitness is 0. However, if the weight of the solution is less than the maximum weight for a solution, the fitness is the value of the items included.

To put this in to practice, assume that the knapsack problem has a maximum weight set to 15. Working from the previous population of four elements and evalu-

[12]This is also sometimes called the objective function

ating each genome using the fitness function described in the previous paragraph, a population is arrived at as in Figure 2.15. Two solutions exceed the maximum weight allowed, and therefore have 0 set as their fitness. The other two genomes have an allowed weight, and thus have an associated non-zero fitness.

The fitness function is the crucial, problem dependent part of the evolutionary algorithm. It allows comparisons of genomes to be made, and is what the optimization is based on. While a simple problem like the knapsack problem has a fairly directly suggested fitness function, other problems may present a difficulty in crafting the fitness function. Furthermore, as the fitness function is central to the working of the evolutionary algorithm, how it functions may help or hinder efficient discovery of a solution. Jansen (2001) provides a means of some basic classification of fitness functions, and describes the limitations that some of these types of functions have.

Selection

Once individuals have been evaluated for the fitness of solutions, the population must be modified with the intention that the next generation will improve over the fitness of this generation. To do that, some of the genomes must be selected to be taken forward into the next generation while others are eliminated. This process is called selection. There are a number of different selection algorithms that may be used by evolutionary algorithms.

Initialization

Perhaps the simplest selection algorithm is to choose only the genome with the highest fitness. Similar to this is taking the highest n solutions. Thus, if the population has 100 individuals in it, one may take the 40 best individuals, culling the remaining 60. This has the advantage of being the simplest to implement. This simplicity comes

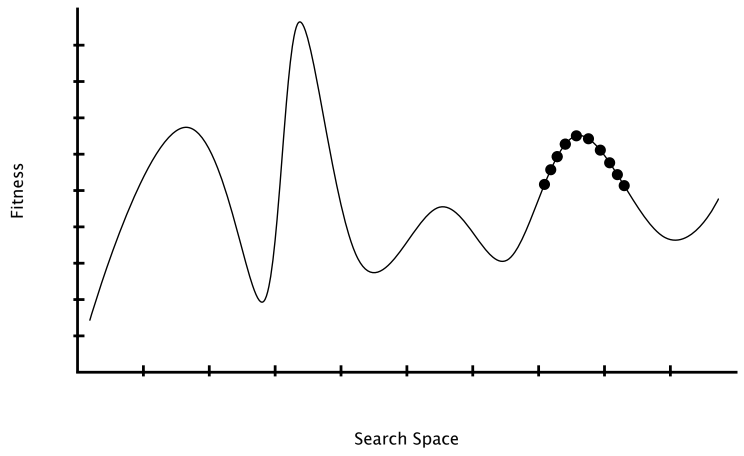


Figure 2.16: If selection chooses only the best elements to continue on to the next generation, it is likely that the solution space will end up being constrained to a local maximum rather than an ideal potential solution..

with a price. Oftentimes the ideal solution may need some portion of a less than ideal genome. If only the best genomes are kept, it would eliminate some of these other needed properties. Furthermore, it increases the likelihood of getting trapped in a local maximum, as it constrains the search space.

An alternative is to use a simple random selection algorithm. Random selection works exactly how it sounds it should. From the initial population, a random subset of that population is selected to continue into the next generation. The drawback with random selection is that it is quite possible to routinely regress in quality of solutions. In the worst case, random selection could drive the evolutionary algorithm in the direction of an ever worsening solution if the worst solutions were consistently randomly chosen.

To avoid these problems, a tournament selection mechanism may be used. Rather than simply selecting a random population, a set of random subpopulations is selected. In each random subpopulation, the individual with the best fitness is selected to continue to the next generation. Thus, with a population of 100 elements with 1/4 of the population expected to continue to the next generation, 25 sets of 4 elements would be created. Each of those sets would consist of random elements. Once the sets had been constructed, the individual in each set with the highest fitness would be selected to continue to the next generation.

Another possible approach is a roulette selection algorithm. The roulette algorithm is a weighted random selection. The details of how exactly this is performed can vary depending on the implementation. The general approach is first to assign a probability of selection to each element in the population. For a desired population size of n, then a random selection from the population would be made n times. Some individuals would be chosen more than once, and some potentially not at all. The general characteristic is that elements with a high fitness will be chosen multiple times, thus making up a greater proportion of the population in the next generation. Elements with an average fitness will remain at about the same amount of the population. Elements with a poor fitness will generally die off, though some may stay through the next population, thus preserving genetic diversity (Padhy, 2007).

As the elements for the next generation have been selected, modifications to those population elements may be made. While some organisms will survive intact to the next generation, without modifications to the genomes no improvement in the solutions would be found. However, by making changes to the genes of individual members of the population, it is possible to find better solutions that were not available in the previous generation. The probability of transitioning without change from one generation to another, or undergoing modification are parameters that may be set on the genetic evolution engine.

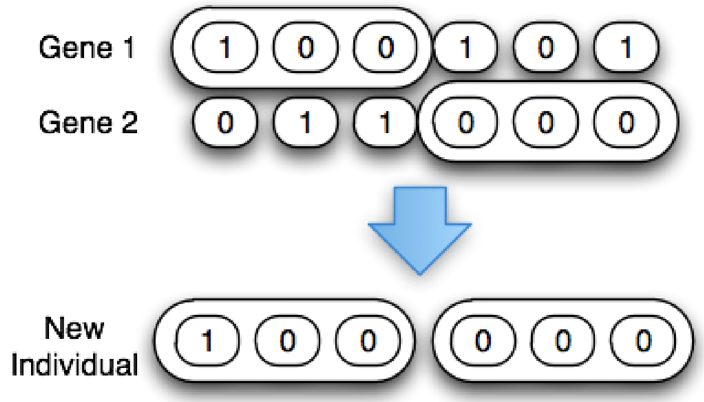


Figure 2.17: Diagram of single point crossover. The individual chromosome is split in the same place for each individual. Then the portion of the gene before the crossover point is selected from one chromosome is selected from one individual while the portion after the point is selected from the other. These are then combined to form a new individual.

Recombination

The primary mechanism for improving genomes in the next population is recombination. Recombination takes a pair of elements and recombines them to create a new organism. Its biological analog is sexual reproduction. The basic process is to select two individuals from the population and perform a genetic operation on them to produce a new individual. There are a number of different genetic operations that can be used to create a new population element. The most common is crossover.

The most basic form of crossover is single point crossover (Padhy, 2007). This is illustrated in Figure 2.17. The basic approach is to select a point common to each genome. The gene segment prior to the split point in the first segment is then paired with the gene segment after the split point in the second gene. This creates a new, unique individual. This approach is simple to implement if the representation of the genome is a bit string, but can also be used with other representational schemes. The only important point is that a common point for the two genomes to be split at must

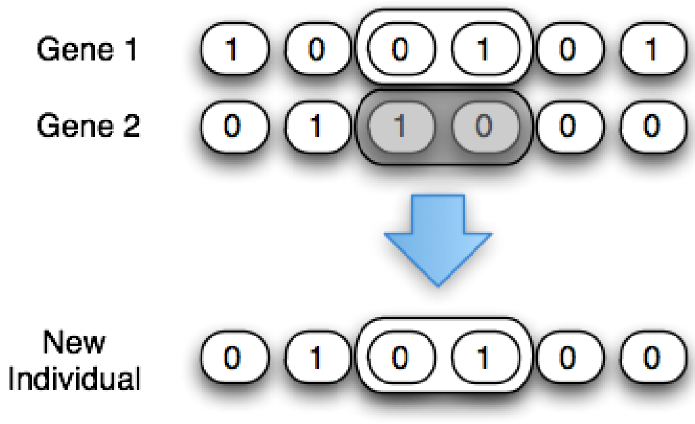


Figure 2.18: Diagram of double point crossover. The individual chromosome is split in the same two places for each individual. The portion between the split in the first gene is then copied into the portion between the split in the second gene.

be identified.

A similar approach is double point crossover (Padhy, 2007). This is illustrated in Figure 2.18. Similar to single point crossover, points must be identified in each genome that are common to each-other. In the case of double point crossover, two points in each gene must be identified. The portion of the gene between the two cut-points in the first genome is used to replace the portion of the second genome that lies between the two cut-points. Single point and double point operate identically if one of the points is at the beginning or the end of the genome.

If the representation of the genome is appropriate, an alteration of these two forms of crossover may also be used that does not preserve the length of the genome. In our knapsack example, this approach to crossover would be inappropriate, as the length of the genome is fixed because of what it represents. It can only ever use 6 elements, no more and no less. However, if rather than using a simple bit-string as the genome one were to use a list of elements included in the knapsack one were to use a list of the included items, the reproduction can be done using cut and splice crossover. As

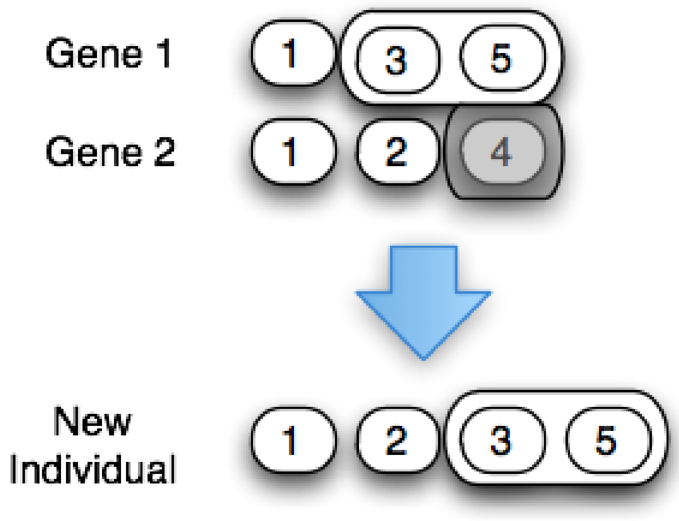


Figure 2.19: Diagram of cut and splice. Part of the first gene is cut out and then spliced into a location on the second gene.

can be seen in Figure 2.19, this can lead to creating a larger gene than was originally in existence. This allows for solutions with certain genetic representations to become more complex though each successive iteration.

There are many other possible types of recombination that can be used depending on the specific genetic algorithm being used. Padhy (2007) details a number of different fundamental versions of crossover, while the current literature is rife with additions to the possible algorithms to be used in ensuring genetic diversity through reproduction.

Mutation

In addition to mating two genomes to produce new offspring going in to the next generation, the genetic algorithm also makes use of mutation to ensure genetic diversity. This is an essential part of the algorithm in making sure that every portion of the solution space can be reached though evolution. Without the mutation operator, any permutation of the solution not represented by a part of the genome in the initial population would be unreachable. Mutation operates on a single gene and uses a genetic operator to modify the contents of that gene alone. A mutation step can be run after creating the next generation via crossover.

The most basic form of mutation is to invert a random bit in the genome. If the gene is represented as a string of bits, flipping a single random one introduces enough change to allow any point in the search space to be reached. Other bitwise operators can also be used. For example, a portion of the genome could be selected and inverted. Alternately, a portion of the genome could be selected and xor'd with a known string.

Mutation is also possible with other representative structures. Additionally, it is not limited to simply changing the values of portions of the genome, but can also be used to add or delete structure to the genome if the representation is not of a fixed size. For example, if the genome were a list of the items to be included in the knapsack in the knapsack problem, a mutation could add a new item to the genome. Similarly, it could remove an item already existing in the genome from the solution. This approach will be seen in more detail in how NEAT handles mutation. The relevant fact of mutation is that a single individual is selected from a portion of the population and randomly altered to create a new individual.

Speciation

Some algorithms, including the NEAT approach used in this paper use speciation. Different individuals may become too dissimilar to be allowed to breed any more. With certain representational schemes, speciation may be a near necessity, as some individuals will be distinct enough that a common crossover point cannot be discovered.

Speciation is often used in multimodal function approximation (Stanley, 2004). In that case, there are multiple optimal solutions that are trying to be found. Speciation is used to assure that each of the optimal solutions can be found by clustering species around each of the different optimal solutions.

Another use of speciation is to maintain different sets of functionality in the population. While not actually applicable to the knapsack problem, one could envision maintaining two species as well. One species could be maintained with the intent of maximizing the value of the items in the knapsack while another population would be maintained to maximize the fitness[13]. In this case, speciation would allow sub-populations to improve within themselves, maximizing different traits while possibly developing other traits that allow it to better solve the general problem.

Convergence

While the genetic algorithm can be run for a fixed number of generations, it is also possible for the algorithm to run until the solutions converge in a single[14] optimal solution. Convergence occurs when the population comes to be focused in one area of the solution space. An example of this can be seen in Figure 2.16 where there is convergence in a sub-optimal state. Convergence can be detected mathematically by comparing the genomes in the population. For example, once a certain percentage of the genes in each genome are identical, the solution can be said to have converged (Padhy, 2007).

[13]It is important to note here that this is not actually how species would be maintained, but is useful for illustrative purposes. In actuality, speciation is used by NEAT to maintain different topologies.

[14]In some problem domains it may be desirable to converge to multiple different solutions, but that is not addressed in this paper.

2.3 Neuroevolution

Neuroevolution takes the biological model for computation, and the biological model for optimization and attempts to use the two together. The goal is to use the genetic algorithm as the training mechanism for optimizing neural networks. There are a number of different specific approaches for accomplishing this. Unlike the differing neural network training algorithms, an evolutionary approach is not tied to any specific ANN architecture.[15]

2.3.1 Fixed-Topology EANN

The most basic approach to integrating an evolutionary process in ANN optimization is to set the topology used and then evolve the weights of the connections between neurons. Stanley (2004) terms this a Fixed-Topology Evolved Artificial Neural Network. The traditional approach to neuroevolution is to simply replace the backpropagation algorithm for setting neural network weights with a genetic algorithm. This operates on a simple, feed forward network. The genome is simply a representation of the different connection weights to be supplied to the neural network when it is run. The quality of that genome is then evaluated according to a fitness function. This approach is described by Padhy (2007). There are a number of problems for which this is an effective way of proceeding, and it can train networks more quickly than back propagation. However, more efficient alternatives have been developed.[16]

[15]Most of this section references Stanley (2004), as he gives a thorough account of the history of work in neuroevolution.

[16]While alternatives for fixed topology EANN's such as SANE, ESP, and CMA-ES are interesting to study, they fall outside of the scope of this paper. These other options are discussed in more detail with references to the original works in Stanley (2004)

2.3.2 TWEANNs

As was seen in section 2.1.2, the arrangement of the neural network can have a significant effect on both the efficiency and the functionality of a neural network. While the fixed topology EANN can simplify the training of neural networks, it still leaves the network architecture as an important part of the workload for the experimenter. To solve this problem, Topology and Weight Evolving Neural Networks (TWEANNs) were developed. A TWEANN is capable of developing both the topology of the network as well as the weights to be used in the network. This has the additional advantage of being able to generate minimal network architectures. As was noted in 2.1.2, being able to optimize the topology of a neural network is an important component of being able to efficiently solve large problems. While there are other systems that have been used to evolve both topology and weight, this thesis will focus only on NEAT.

2.4 NEAT

NeuroEvolution of Augmenting Topologies is a TWEANN that attempts to build up neural networks. It was introduced in Stanley and Miikkulainen (2002) and fully described in Stanley (2004). It has been shown to be effective in developing solutions to control problems (Monroy et al., 2006; Pardoe et al., 2005; Stanley, 2004). NEAT addresses three problems inherent in TWEANN's: Encoding structure, promoting innovative solutions, and minimizing the solution space so that efficient solutions can be found. The rest of this description of NEAT will focus on how it deals with each of these elements.

2.4.1 Encoding

Encoding in a TWEANN is dramatically more difficult than it is in a fixed topology network. The fixed topology network only needs an encoding structure that is capable of working with the weights of a neural network. The addition of topological structure not only adds more data to the representational scheme, but it also means that the genotype becomes unbounded in length. As we have seen, this can make some genetic operations more difficult.[17] Furthermore, representing a neural network for use in a TWEAN is more difficult than simply creating a representation of a neural network. Beyond being required to have the representative capacity to describe the neural network, the encoding scheme must also be compatible with genetic operators to allow for the optimization of the network.

There are two general schemes for encoding neural networks, direct and indirect encoding. In direct encoding, the encoding scheme is a direct representation in each genome of the nodes, connections and weights that will appear in the final neural network. Indirect encoding, on the other hand, uses a grammar tree to describe how the network is to be constructed. This grammar tree allows a more expressive genotype to be created in that a single gene in the genotype can map to multiple nodes in the phenotype. As NEAT uses direct encoding due to concerns over the expressive power of a chosen indirect encoding (Stanley, 2004)[18], only that form of encoding neural networks will be examined here.

[17]Some TWEANNs simply set an upper limit for size for representational efficiency. However, this denies the researcher one of the primary advantages of an evolved approach to neural network optimization. The entire goal of TWEANNs is supposed to be that the researcher is not required to know things like the optimal size or weights of the neural network, but that data is to be decided by the optimization algorithm. Setting an upper limit reintroduces that dependency on human understanding of the problem space.

[18]Some recent work has suggested that this may not be an ideal approach and that an indirect encoding may scale better to larger problems (Reisinger and Miikkulainen, 2007). Stanley acknowledges in his own thesis that indirect encoding may be a desirable way to expand NEAT in future work in order to better allow for scaling to larger problems.

Direct Encoding

Direct encoding represents each node, connection and weight individually in the genotype. There are a number of ways that this can be accomplished.

The simplest approach is binary encoding. The network is represented by a bit matrix. This approach however has significant drawbacks that prevent it from being used by NEAT. Since the matrix size is fixed, there is an upper limit on the size of the network represented. If it becomes necessary to grow to a larger network, the experiment must be restarted with a larger matrix. Secondly, many of the possible representations are simply non-functional, leading to wasted computation in the evaluation cycle. Finally, minimally connected networks are extremely wasteful of the representative space. Most of the matrix ends up being set to 0 to indication that connections and wights do not exist. For these reasons, binary encoding is not used by NEAT.

Most TWEANNs use graph encoding. In graph encoding, the structure of the neural network is encoded as a graph. This seems intuitively ideal, as the structure of a neural network is a graph. The difficulty here becomes effectively utilizing genetic operators on the graph. While mutation can be clearly defined in how it operates, crossover operations present a difficulty.

The problem with crossover operations in graph encoding is called competing conventions, or the permutations problem. In the TWEANN literature, this is sometimes also known as the Variable Length Genome Problem. The difficulty is that a single functional neural network or sub-network may have multiple genetic representations. This creates a problem during crossover because even though two representations may be identical, they may become genetically incompatible due to the difference in representation. This problem can be seen in Figure 2.20. In the figure, both networks

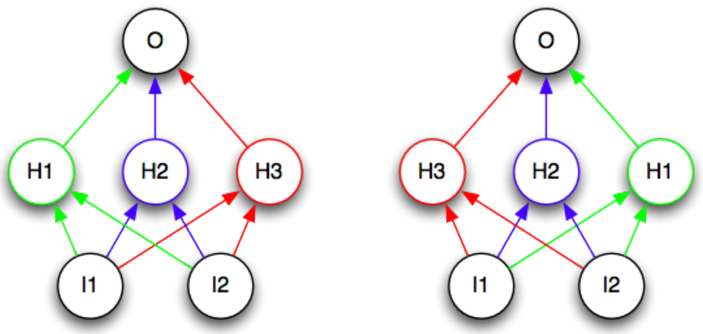


Figure 2.20: Each of these networks computes the same function, but has a different representation. This results in the networks being instantiated in different ways. This creates problems in mating, as the networks are no longer compatible with each other.

do exactly the same thing. However, in the first network, the first hidden node is node $H1$. In the second, the first hidden node is $H2$. In this case, the hidden nodes represent the semantic function undertaken by that node in the neural network. The TWEANN recombination mechanism is normally unaware of the semantic function of the node. Therefore, when it mates the two by replacing the first node in the second network with the first node in the first network, the hidden layer becomes $H1, H2, H1$. Consequently, the functionality of the first hidden node is duplicated. At the same time, the functionality of node $H3$ is lost in the child network.

The solution used by NEAT is to use historical marking to allow the identification of similar topologies. Historical marking makes a note of topological innovations introduced through additive mutation by assigning them a number. Thus, continuing to look at the example in Figure 2.20, as each hidden node is introduced via mutation, a new historical marker is assigned to it. Therefore, $H1$ may have been assigned historical marker 3, $H2$ assigned historical marker 5, and $H3$ assigned historical marker 9. When later comparing the two networks, the location of $H1$ and $H3$ is irrelevant. The recombination mechanism can discern that the first and third nodes in the hidden layer are the same because they share the same historical innovation marking. This limits the impact that the competing conventions problem has on recombination.

2.4.2 Innovation

The second key problem in TWEANNs is protecting innovation. Topological advances towards a solution that are made through mutation or mating may not survive through to the next generation because although the topology may be advantageous, the connection weights are not also at that instant also advantageous. Thus, in this case, the topology may be correct for the solution, while the connection weights prevent that topological improvement from being used. In a fixed size EANN, this is not a problem, as the structure is fixed. Only one element of the neural network is being evolved at a time.

The solution used by NEAT is speciation. While speciation is normally used to ensure that multiple solutions to the various different optimal portions of the solution space are maintained, in TWEANNs the goal of speciation is to preserve a diverse set of network structures and allow each to evolve the appropriate connection weights. This has the further advantage in limiting breeding to those individuals that are likely to be genetically compatible. Individuals that are in the same species are likely to share the same set of historical innovations, making them compatible for reproduction.

In NEAT, speciation additionally uses fitness sharing to promote the most promising species to continue to be propagated. In fitness sharing, the fitness score of the best individuals in a species are shared in some measure by the other individuals in the species population. This is dependent on having a measure of the distance between two individuals in the population. NEAT is able to provide this measure by using the historical marking. If the distance between two individuals is greater than some threshold value, the fitness is not shared. If the individuals are close enough for fitness sharing, some of the fitness of one individual is shared with the other, proportional to the distance between the two individuals.

2.4.3 Efficiency

The final issue addressed by NEAT is the efficiency of the network topography. Recall from Attik et al. (2005) that generating an optimal network topography is a difficult problem for neural networks. The approach that Attik proposes is a hybrid one, both adding new nodes and eliminating unused nodes. The approach NEAT takes is similar to the optimization approach recommended by Attik.

NEAT optimizes its ANN topology using the principle of complexification. It starts by creating a minimally connected network. Through mutation, it then adds new nodes to the network. It is limited to adding nodes that have an advantage by the use of the fitness evaluation. Only those nodes that prove themselves to be advantageous in improving a solution are kept through time. It also, as previously discussed, assures that these innovations are given the chance to develop into a useful innovation through through speciation.

NEAT also can trim unused nodes from the network if they no longer improve fitness. In the case that a species with unused nodes becomes stagnant or becomes regressive in fitness, that species may be culled from the population over the course of generations.

2.5 Summary of Foundations

This chapter has covered the basics of what ANN's are and how they function, as well as what the genetic algorithm is and how it functions. More importantly, it has applied those foundations to the current state of the art in neuroevolutionary approaches. This provides the necessary background information to understand the details of the experiments in this thesis.

3 Previous Work

Previous work on adaptive, evolved networks has focused on small sample cases to develop an understanding of how different architectures and optimization strategies perform. There are two primary works that were taken as the starting point for this experiment. Firstly is work by Floreano and Urzelai (1999) and Urzelai and Floreano (2001) in evolved robot control, and secondly is an experiment by Stanley et al. (2003).

3.1 Floreano and Urzelai

The first experiment that directly informs this one is Floreano and Urzelai (1999) and Urzelai and Floreano (2001). The goal of the experiment was to examine whether or not availing a robot controller of adaptive synapses would improve the speed of training and the scalability of that approach.

The experiment derives from the observation that many neuro-controllers develop in such a way that they merely exploit contingent invariances in the training environment, rather than developing a robust solution. While the tricks that these solutions employ can be advantageous to efficient discoveries of a solution, they often break when the environmental conditions change in such a way that makes the exploited condition no longer true.

As a solution to this sort of problem Chalmers (1990) proposed that plastic synapses may allow a controller to better deal with environmental changes. Floreano expanded on this approach with a number of experiments in (Floreano, 1998; Floreano and Nolfi, 1997).

3.1.1 Encoding

Encoding adaptivity is extended in Urzelai and Floreano (2001) from previous work to be more compact. The approach uses an artificial chromosome that is converted into a neural network controller. The initial weights are set to small random values on initialization. These synaptic weights are then modified as the network runs according rules ls that are specified on the genome. Urzelai and Floreano (2001) specifies four adaptation rules that are encoded on the genome. There are two ways that these adaptation rules are encoded. The first is to encode each synapse for specified rules allowing each of the nodes to have different properties in each of the incoming synapses. A second, more compact way of encoding adaptive features is to encode the adaptation rules at the node level. This means that all the synapses of the specified node will share the same characteristics.

In Urzelai and Floreano (2001), each gene is encoded with five bits. The first bit represents the sign (positive or negative) of the signal traveling along the synapse. The next four bits encode the actual properties of the synapse. In the case of traditionally evolved weights, these four bits simply specified the weight of the connection. In the case of adaptive synapses, these four bits are used differently. The first two bits specify which of the four adaptation rules are being encoded. These four adaptation rules are each a different Hebbian rule for how to update the weight of the neuron based on previous firings. The last two bits, then specify one of four preset learning rates (0.0, 0.3, 0.6, 0.9). This experiment also allowed for noisy synapses, but the inclusion is not relevant here.

3.1.2 Experiment

Environment

The experiment uses a robot positioned in a rectangular environment. The robot is equipped with a vision sensor, infrared proximity sensors, and ambient light sensors. The neural controller is a fully recurrent neural network that activates two motor neurons, each of which controls one wheel on the robot.

The rectangular environment has a light bulb on one side of the box. Beneath the light is a gray area of floor. At the beginning of the experiment the light is off. The light is switched on if the robot passes over a black area of the floor on the upside of the rectangle. A black stripe is painted on the wall of the rectangle adjacent to the black area of the floor.

Fitness

The fitness function for each iteration of the neuro-controller is the total number of sensory motor cycles that the robot spends on the gray area of the rectangle while the light is on divided by the total number of sensory motor cycles available (500). This effectively gives the percentage of time the robot spends on the gray area with the light on. This will never be 100% due to the amount of time that moving to turn the light on and then back to the gray area takes.

Implementation

The experiment begins with the robot placed in a random location and orientation in the box. The robot then runs a given neural network for 500 sensory motor cycles. Each sensory motor cycle lasts 100 ms. After each cycle, the neural controller can update its synaptic weights according to the adaptation rules encoded on the genome.

The experiment was conducted both on the physical robot, as well as with simulations that included uniformed 5% noise added to sensor activation. The results on the physical robot did not differ in any significant way from the results of those obtained in simulation. In simulation 10 different populations of 100 neural controllers were each evolved for 200 generations. The individual fitness of a given neural controller in a given generation was the average of three experimental runs. At the end of each generation the 20 best individuals were reproduced.

3.1.3 Results

The conclusion of the experiment was that adaptive synapses produced both higher maximum fitness and average fitness in fewer generations than traditional genetically evolved weights. Additionally, node encoding was found to produce better results than synaptic encoding for adaptive synapses. In the case of traditionally evolved weights and node encoding, solutions could not reliably be converged upon.

The experiment was also duplicated using larger neural networks to determine whether or not the results would scale upwards. The neural network was extended by adding 20 hidden neurons. This increased the size of the genetic string from 60 to 160 bits for node encoding, and from 720 to 5120 bits for synapse encoding. The results of this expanded experiment showed that adaptive synapses with node encoding produced comparable results to the network without hidden nodes. However the increase in genetic string size for synapse encoding produced controllers that were unable to increase their fitness. The hypothesis of Floreano and Urzelai (1999) is that the search space may contain a smaller proportion of successful solutions than the node encoding contains.

3.2 Stanley, Bryant and Miikkullainen

The work that directly inspired this paper was Stanley et al. (2003). This was an extension of Floreano and Urzelai (1999) intended to determine whether or not adapted synapses were necessary in tasks that required adaptation. While Floreano and Urzelai had shown that adapted synapses could prove beneficial in solving some sets of problems, it did so in a fixed environment where adaptation was not necessary. The goal then of Stanley was to determine whether or not changing environments required plastic synapses and whether or not local learning rules enable a neural controller to adapt to its environment.

3.2.1 Encoding

The experiment used the NEAT method described in 2.4. NEAT was modified to allow the evolution of local Hebbian rules on individual synapses. This corresponds with the synaptic encoding of Floreano and Urzelai (1999), but instead of limiting the Hebbian rule to one of four types with one of four preset learning rates, NEAT allows a fuller range of parameters to be explored by the evolutionary process. This was done by evolving of a generalized learning rule that combines the properties of each of the four learning rules used by Floreano and Urzelai.

It is important to note that NEAT, by starting with a minimally connected network and evolving structure avoids some of the size problems encountered by Floreano. Floreano, by specifying fully connected recurrent neural network immediately started with a very large search space. NEAT, on the other hand, gradually grows the search space through structural mutations that grow the topology of the network. While this allows for a theoretically unbounded search space, it also limits additional topology to those extensions are beneficial in the end solution.

3.2.2 Experiment

As the goal of Stanley's experiment is to test whether or not adapted synapses are necessary for dealing with a changing environment, a problem domain had to be found where the optimal course of action could only be discovered via the exploration of the experimental world. The dangerous food foraging problem domain is one such instance. The general principle is that a controlled agent is placed in an environment with either food items or with poison items. The agent must develop both the ability to find a food source, and to change its behavior based on discovered properties of the food source. This requirement for a change in behavior based on the environment should push towards a need for adaptivity.

Environment

The simulated agent is similar to the robot used in Floreano and Urzelai (1999). It has two wheels, each controlled by a separate motor. It also has five sensors for detecting each type of potential food item. Finally, the robot has a pleasure sensor which activates that consumes food and a pain sensor which activates when it consumes poison.

As the problem domain is explained, the controlled agent is placed at the center of the field with randomly located potential food items. Food items may be of one of two types, A or B. Each experiment run will find only one type of food. Food items are either all poison or all food. The agent must attempt to consume at least one food item to determine whether or not all food items are desirable. After consuming a food item network receives either a pleasure or a pain signal, based on whether or not the food item was food or poison. If the item was food, the agent should then attempt to find and consume all the available food. If the item was poison, the agent should avoid further foraging.

Fitness

The fitness of the neural network was determined by taking the difference between the amount of food eaten by a robot and the amount of poison consumed by the robot. NEAT requires the fitness to be positive, so the maximum possible amount of poison consumed was added to this value. In the implementation of the experiment, the maximum amount of poison that could be consumed is 32 items. This gives the fitness function f

$$f = 32 + e - p \tag{3.1}$$

where e is the number of edible items and p is the number of poisonous items.

Implementation

The experiment was run in two parts, each part run five times. The first evolved fixed weight recurrent in neural networks over 350 generations. The second evolved recurrent neural networks with plastic synapses over 500 generations. Each iteration consisted of eight trials, two trials of edible type A items and two trials with edible type B items, followed by two trials each of type A and B poisonous items. Before each trial, the network is reset to its initial state.

3.2.3 Results

The experiment found that fixed weight recurrent networks were able to efficiently solve the task in under 350 generations. The typical solution of the recurrent network, behaviorally, was that upon finding poison it would turn away from all food items and drive to the edge of the environment. Other similar recurrent solutions were also found that tended to rely on what Stanley terms a “trick” using recurrent connections on the output nodes.

The adaptive networks have more difficulty discovering solutions. Only three of the five runs converged to consistently score the maximum fitness. However, the adaptive solutions, rather than relying on a trick using recurrent connections, were described as finding more holistic, functionally different solutions.

Stanley discusses in some detail how and why fixed weight recurrent networks were able to efficiently solve a problem requiring adaptation. The open question he leaves at the end of the paper is whether or not fixed weight recurrent networks can scale up to more difficult tasks or whether there is a class of problems that require adaptive synapses. A further exploration of these questions is the goal of this paper. The details of that examination can be found in the next several chapters.

4 Basic Experiments

To verify that the NEAT implementation was functioning properly, and to create a baseline set of results, an initial set of experiments were run. These tests included the self verification tests that were released with the NEAT software, as well as several experiments that were independently implemented to assure that there was an adequate understanding of how to interact with the NEAT software

4.1 Self Verification

The NEAT software package contains several experiments in the distribution both designed to test that the software operates as intended and to provide an example of how a new experiment ought to be coded. These tests develop an xor gate via neuroevolution, and controllers for the single and double pole balancing experiment. After porting the NEAT software to a more modern version of the GCC compiler, the included tests were run to assure that the software was functional. Each of the included tests is described in detail in Stanley (2004).

4.1.1 Self Verification Tests

NEAT contains three self verification tests that were run. The following sections describe each of these basic experiments in turn, and then summarizes the results of running them.

Xor

The xor test is the canonical test for neural network functionality (Luger, 2001; Principe et al., 1999). A neural network for evaluating the exclusive-or is one of the simplest examples of a non-linearly separable classifier. The simplest solution requires two input nodes, a single hidden node, and an output node. NEAT is able to solve for this in generally 32 generations.

Pole Balancing

A more difficult problem than xor that is often used as a benchmark of neural network functionality is pole balancing (Moriarty and Miikkulainen, 1998). In pole balancing, a single pole is placed on top of a rolling cart. That cart is placed on a track so that it can move either to the left or to the right. The goal of the controller is to be able to move the cart to the left or the right and maintain the pole in an upright, balanced position. The controller fails if the pole falls over. This operates with a simple two dimensional physics. Traditionally, the controller uses as its input the position on the track, the velocity of the cart, and both the angle and angular velocity of the pole. NEAT was able to solve this task efficiently.

Double Pole Balancing

Stanley (2004) notes that pole balancing was at the time of publication not a difficult task for a modern neuroevolutionary system. Single pole balancing has been extended to the problem of double pole balancing. In this case, rather than simply attempting to keep a single pole upright, a second pole is added to the cart. This again is a problem that NEAT is able to efficiently solve.

4.1.2 Self Verification Results

All of the tests in included in the NEAT distribution were able to converge upon a solution within the parameters established in Stanley (2004). This demonstrates that the changes needed to be made to NEAT for it to compile under gcc did not change the underlying functionality in any noticeable way. Any further results either are the result of the NEAT algorithm or are the result of the specific experiment and its implementation.

4.2 Cannon Controller

To verify that the implementation of experiments was correct, a simplified controller was developed. For this experiment, a neural controller for a two dimensional cannon was developed.

Experiment Setup

The goal of the cannon controller is to control the angle and the power of a cannon shot to iteratively hit a set of targets. At the beginning of each generation, a wind velocity between 0 and 50 meters per second is generated. Then a list of 100 targets with distances ranging from 5,000 to 25,000 are generated. Each potential controller is evaluated on its ability to hit all 100 targets. For each target, the controller is allowed to make 50 shots iteratively to find a solution.

Each canon controller takes as input the wind, distance to the target, and the distance the previous shot missed by. Distance to the target is normalized to a range between 0 and 1 by dividing by the distance to the target by 50,000. Wind is normalized to a range between -1 and 1 by dividing by 50. Finally, the distance missed by is normalized to a range between -1 and 1 by dividing by 50,000. If there was no previous shot, the distance missed by is set to 0. If the shot lands within 50 meters of the target, it is counted as a hit. If there is a hit, the missed input is set to 0 and the network is activated 20 times.

The output of the cannon controller is the angle of fire and the power of the shot. The output is limited to a range from 0 to 1. In the case of the angle of fire, this output variable is multiplied by 2 to give the angle of fire in radians. The power of the shot is multiplied by 500 to give the muzzle velocity from the cannon in meters per second.

The location of impact is determined first by calculating the time of flight, then calculating the distance travelled over that time at a constant velocity. The equation for time of flight is:

$$t = \frac{(2(500O_0))\sin(2O_1)}{9.8} \tag{4.1}$$

where O_0 is the velocity output and O_1 is the angle output. As mentioned before, these output values range from 0-1, therefore, O_1 is multiplied by 2 to give an angle between 0 and 2 radians. O_0 is likewise multiplied by 500 to give an appropriate value for the velocity. The distance to impact is then calculated by multiplying the velocity, time and cosine of the angle together as follows:

$$d = (500 * O_0 + w)t\cos(2O_1) \tag{4.2}$$

In this case, w is the wind speed, and the other variables are as before. While this is not accurate physics in many ways, that is inconsequential to the ability of a neurocontroller to be able to develop a solution to the physics used.

The population was initialized with a random initial neural network with up to 2 hidden nodes. The population size was set to 500 and the experiment was run for 500 generations. This was done six times, and then repeated after enabling plastic synapses. All networks were allowed to use recurrent connections. If a fitness of 100 was attained for 10 straight generations, it was determined that a solution had been converged upon and the run was ended.

Results

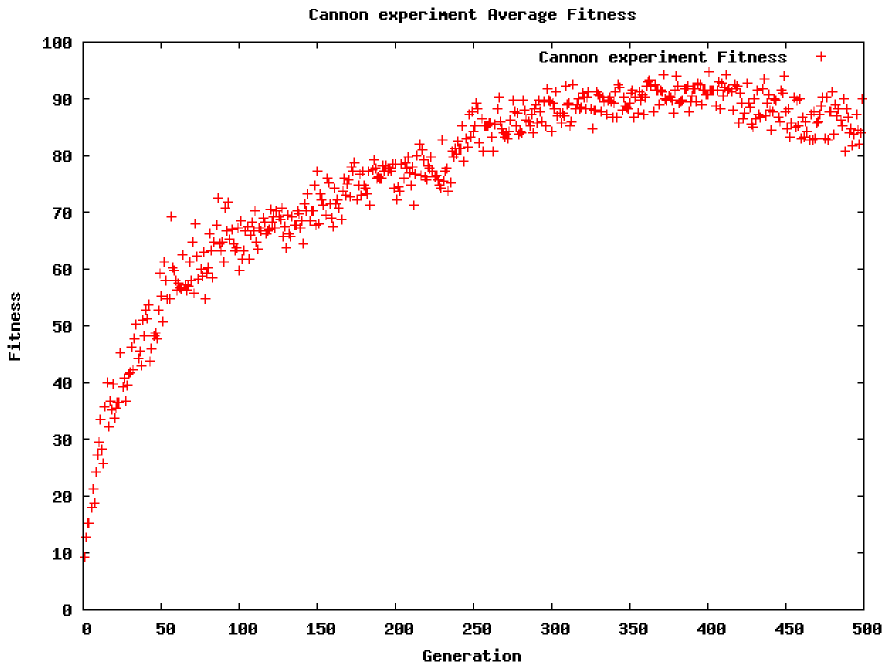


Figure 4.1: Average Fitness of recurrent cannon solutions

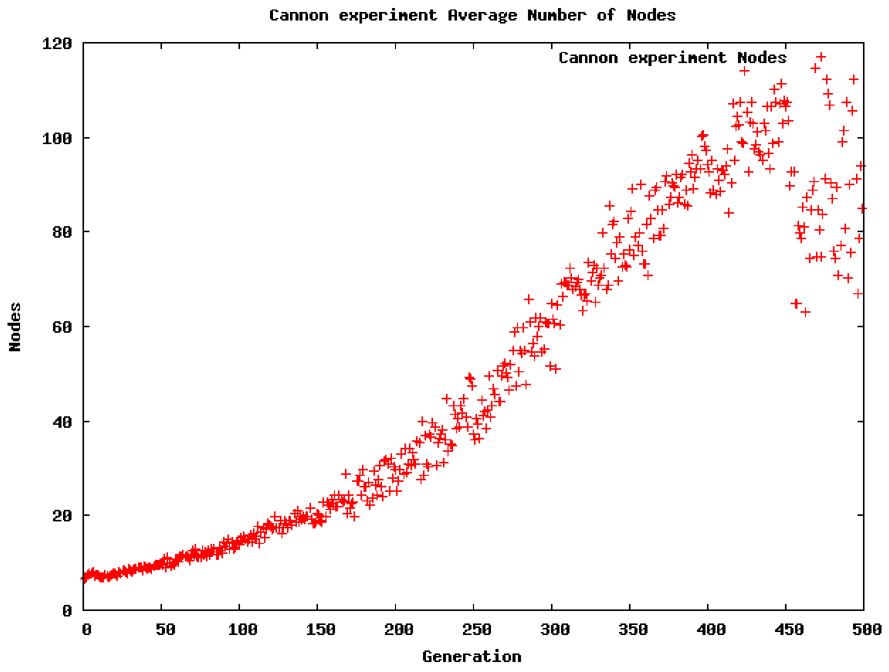


Figure 4.2: Average number of nodes for recurrent cannon solutions

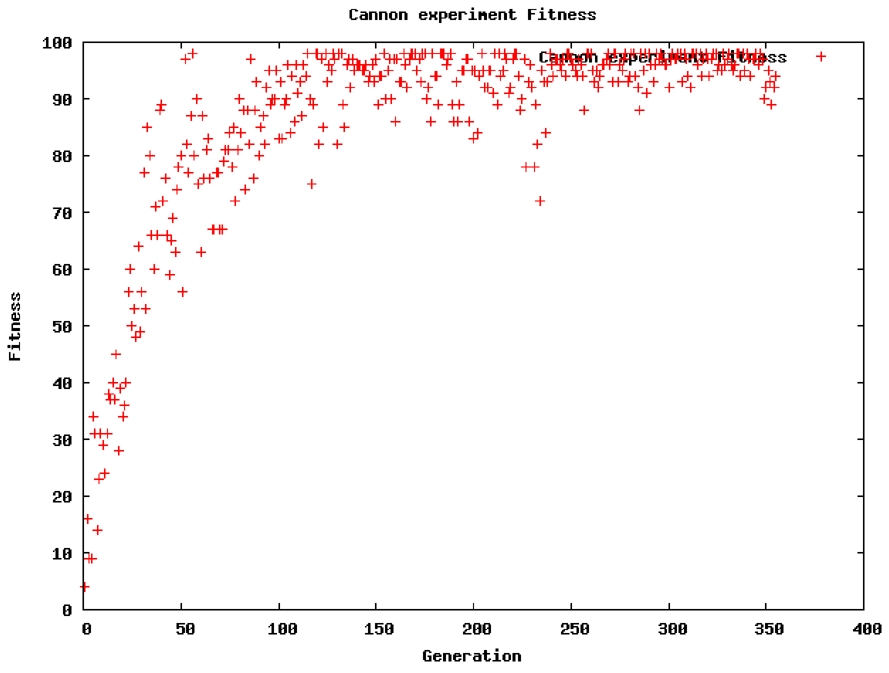


Figure 4.3: Cannon best run fitness

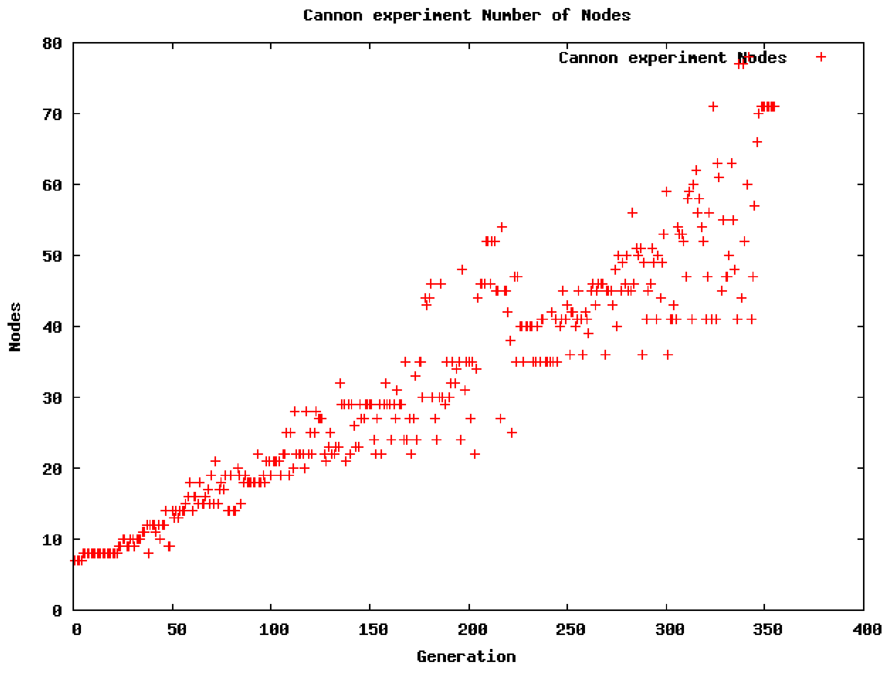


Figure 4.4: Cannon best run number of nodes

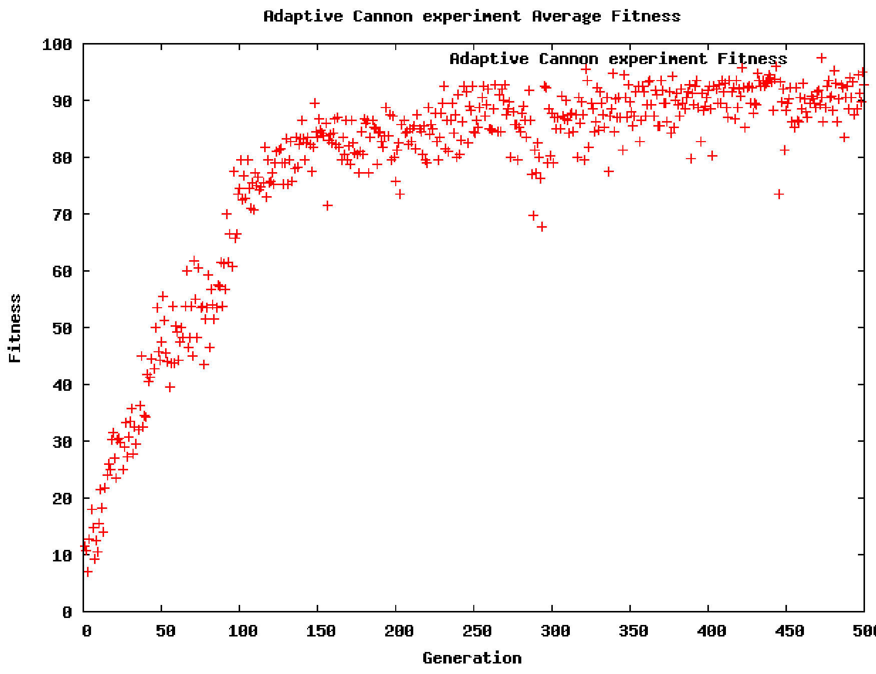


Figure 4.5: Cannon average adaptive fitness

Each run of the cannon simulation took between 4 and 6 hours on a 3.4 ghz AMD Phenom II processor. As the computer is a quad core machine, 3 simulations could be run in parallel. With recurrent connections, the average fitness curve as seen in Figure 4.1 shows that the experiment was able to evolve improvements into the solutions. The average tends towards convergence until those solutions that have already been converged upon start to drop out of the pool being averaged. With recurrent connections, NEAT was able to find a solution in three of the six runs. The best recurrent node run, seen in Figure 4.3, quickly approaches a fitness of 100 and manages to converge on a solution shortly after generation 350. As seen in Figure 4.2, the number of nodes also increases following the expected curve.

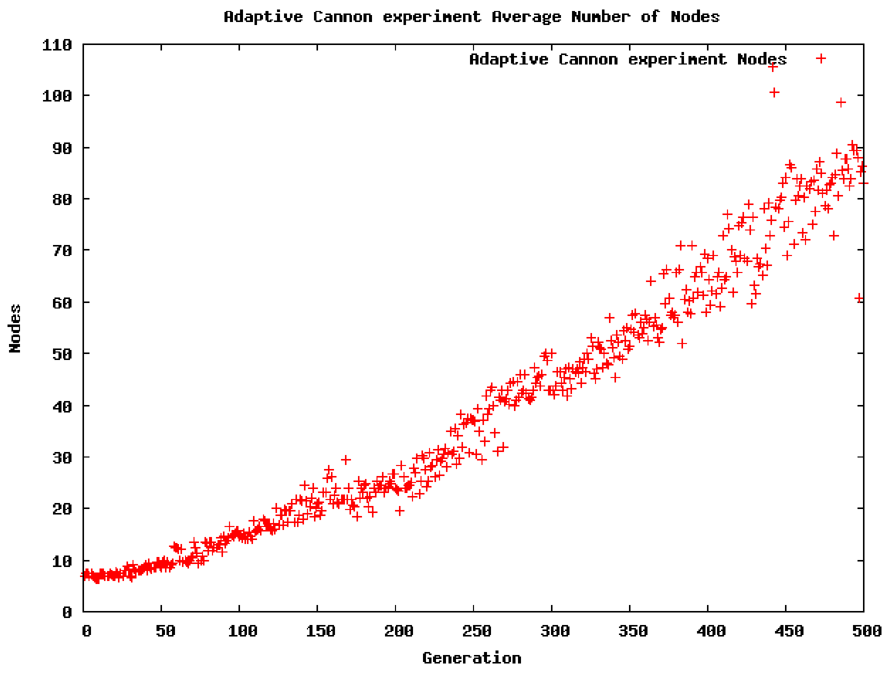


Figure 4.6: Cannon average number adaptive nodes

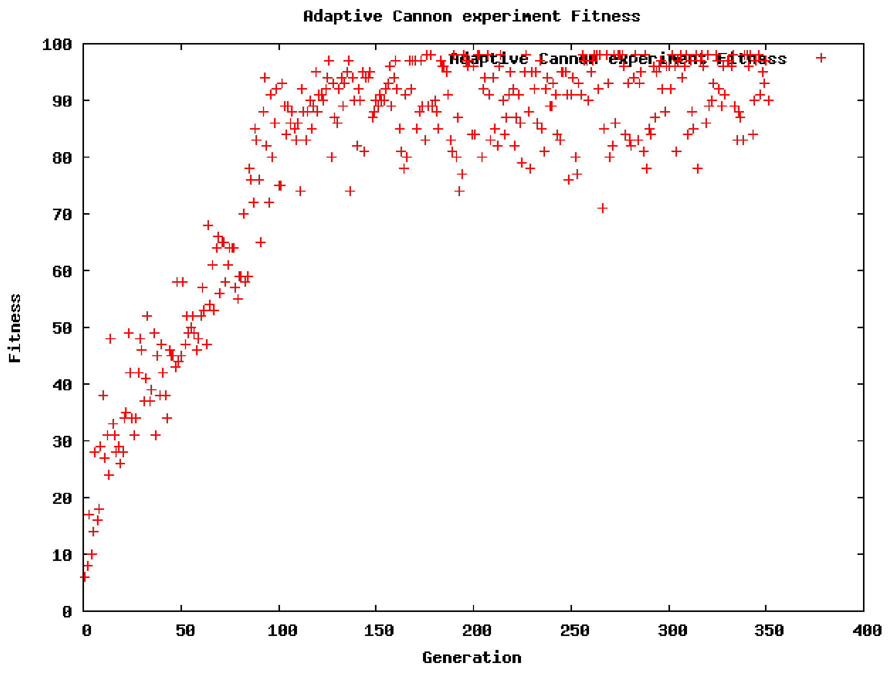


Figure 4.7: Cannon best adaptive run fitness

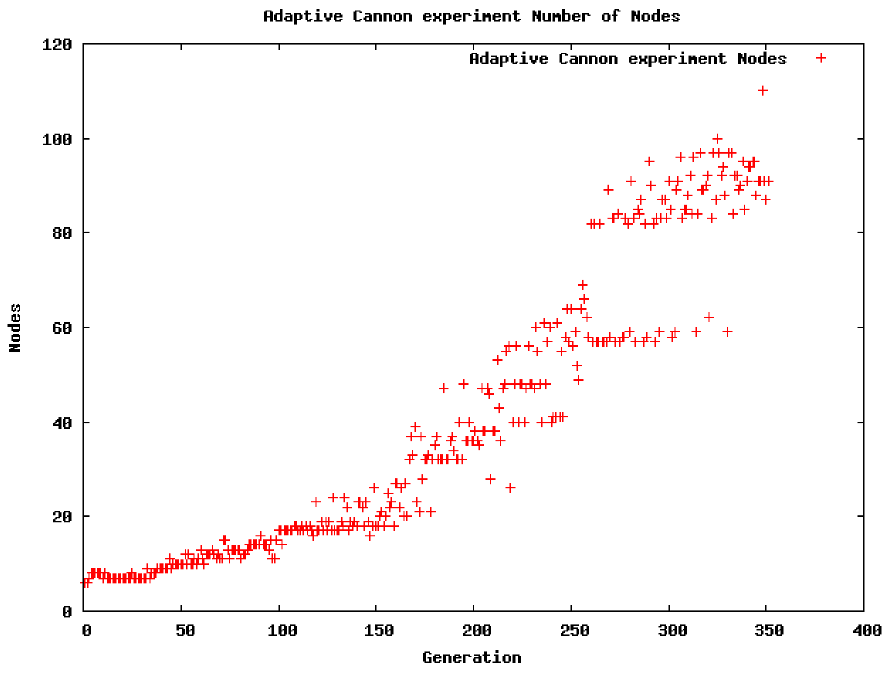


Figure 4.8: Cannon best adaptive run number of nodes

The results for the canon experiment using adaptive synapses were similar. The time taken per run was in the same range as for the recurrent networks. The average fitness curve (Figure 4.5) was similar to that of the recurrent solutions. In the case of the adaptive runs, two of the six runs managed to converge on a solution. In the best case, as seen in Figure 4.7, this was found after roughly 350 generations. Comparing Figure 4.2 and 4.6, it can be seen that the number of nodes needed for the adaptive solution was similar to the number needed for the recurrent solution.

What these sets of experiments demonstrated was that the NEAT system is being interfaced with correctly in the experiments written for this thesis. Every other experiment written in this thesis used the same boilerplate code. The only changes from experiment to experiment were the implementation of simulated world and the fitness function.

4.3 Food Foraging

The food foraging experiment is a simplified version of the dangerous food foraging experiment described in Stanley et al. (2003). The experiment was modified so that there was no poisonous food. The modification was made by modifying the experiment to no longer check whether a food item was or was not poisonous.

Experiment Setup

The neural controller was designed to control a robot with sensors for both type A and type B items and two wheels. The rough design of the robot can be seen in 4.9. The sensors are arranged around the robot in pairs. At each sensor position, there is one sensor for type A items, and a second sensor for type B items. Each sensor returns v according to the function $v = \frac{200-d}{200}$ where d is the distance to the nearest

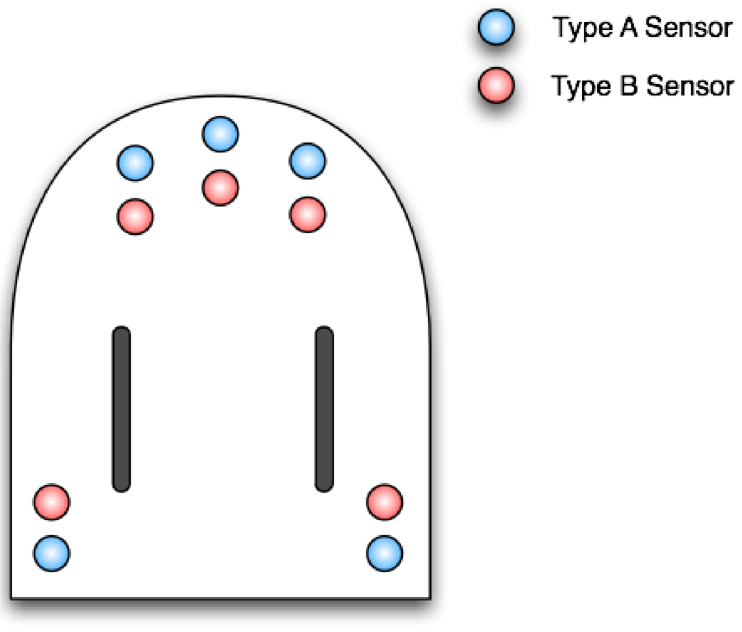


Figure 4.9: Design of the robot for food foraging. it has sensors for items of both type A and Type B. These are arranged around the front and back. It also has two wheels. There are three outputs. The forward speed and the amount to turn left and the amount to turn right.

object within the arc of the sensor of the appropriate type. If there is no element of the appropriate type in the arc of the sensor, the value is set to 0. In addition to what is shown in the diagram, there is a pleasure sensor.

There are three forward looking sensors, and two that detect behind the robot. The left forward sensor will detect the appropriate objects between 310° and 350°. Mirroring that, the right forwardsensor has a detection arc from 10° to 50°. The sensor pointed directly forward has a narrower arc from 355° to 5°. This is intended to make it easier for the robot to move directly forwards if an object is there, without moving past it due to the object being off center at too high an angle. rear sensors detect from 150° to 180°, and then from 180° to 210°.

The robot is capable of turning up to 4° per time step. This is small enough

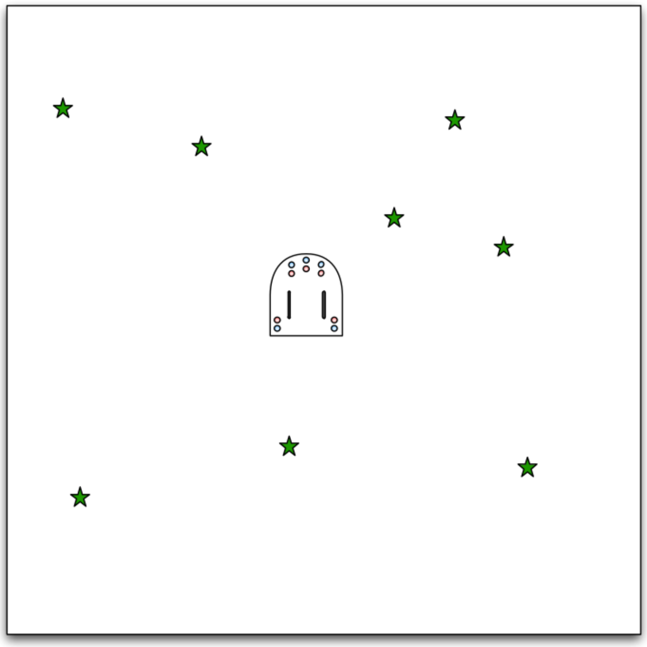


Figure 4.10: The robot for food foraging is placed at the center of the experiment world with 8 food items scattered throughout the world.

that over the course of two time steps, while turning, the robot should be able to roughly isolate the position of a nearby item that was in sensor range. The robot can also move forwards up to 1 unit each time step, Finally, if there are any food items within 3 units of the robot at the end of a time step, that item is consumed. Upon consumption of food, the pleasure sensor is activated for 20 time steps. If the robot moved to the edge of the world, it could no longer advance on that axis. So, if the robot were traveling at 45° when it hit the right wall, it could continue to move forward and would slide up the wall until trapped in the corner. Alternately, it could turn and move back into the arena.

The goal of the robot was simply to find and consume food sources. The robot was placed at the center of a 50 by 50 area[1]. Eight items of either type A or type B are then randomly placed around the area that the robot can travel in. The type of

[1]There are no units needed here, as the implementation was only in the simulation

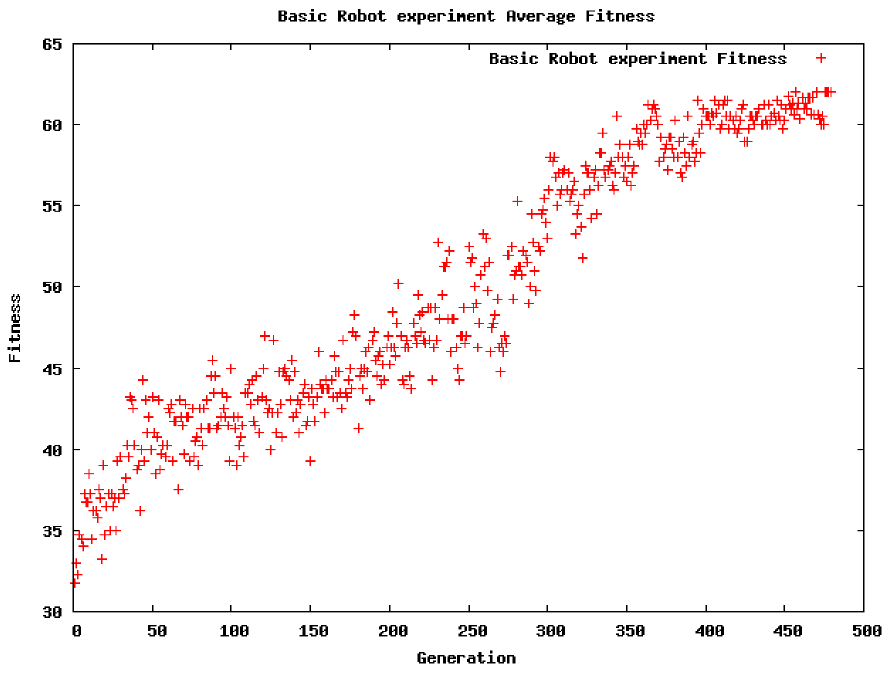


Figure 4.11: Food Foraging average fitness

item is specified on creation of the experimental world.

The experiment was initialized with a random initial neural network with up to 3 hidden nodes. The population size was 500, and the experiment was run for 500 generations. Each generation tested the ability of each individual to consume all eight food items in each of 8 worlds that were randomly generated for that generation. The fitness function was simply the sum of total items of food eaten in the 8 experimental worlds. This gives fitness ranges from 0 to 64. The experiment was run five times. The experiment was only run allowing recurrent networks. Another adaptive set of tests was not run. Achieving a fitness of 64 ten times in a row marked that experiment as having converged upon a solution.

Results

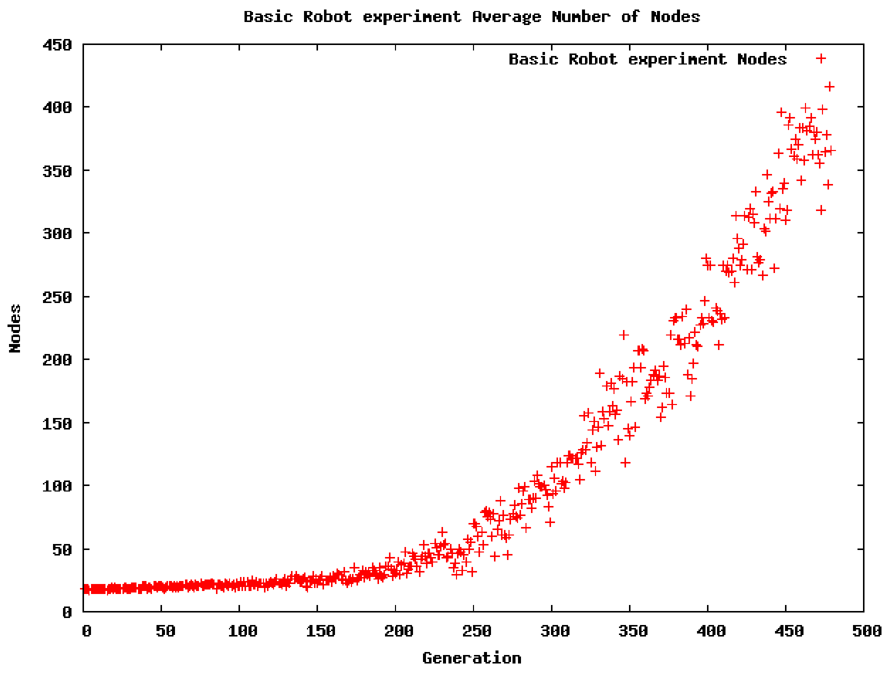


Figure 4.12: Food Foraging average number nodes

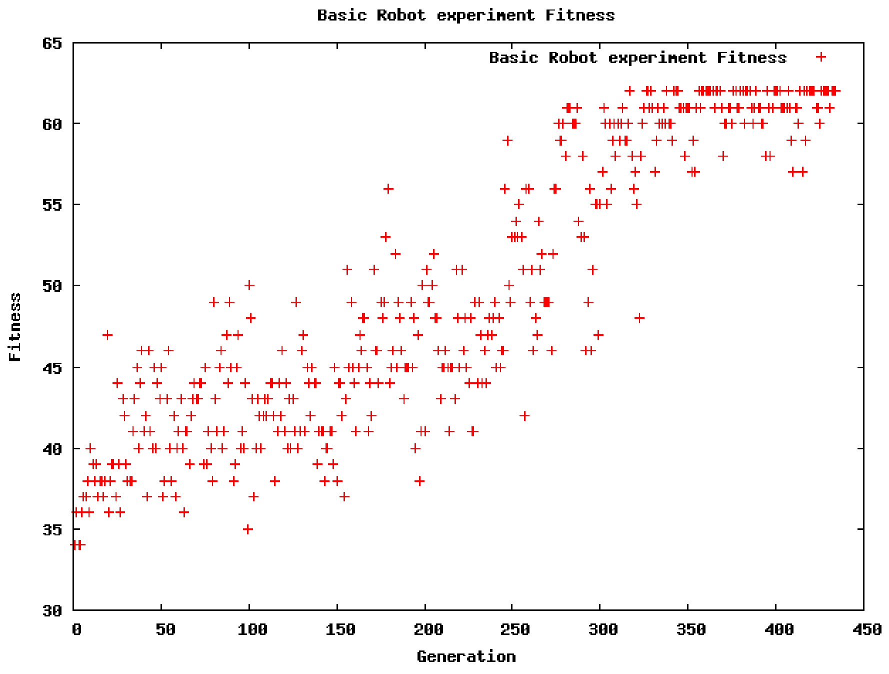


Figure 4.13: Food Foraging best run fitness

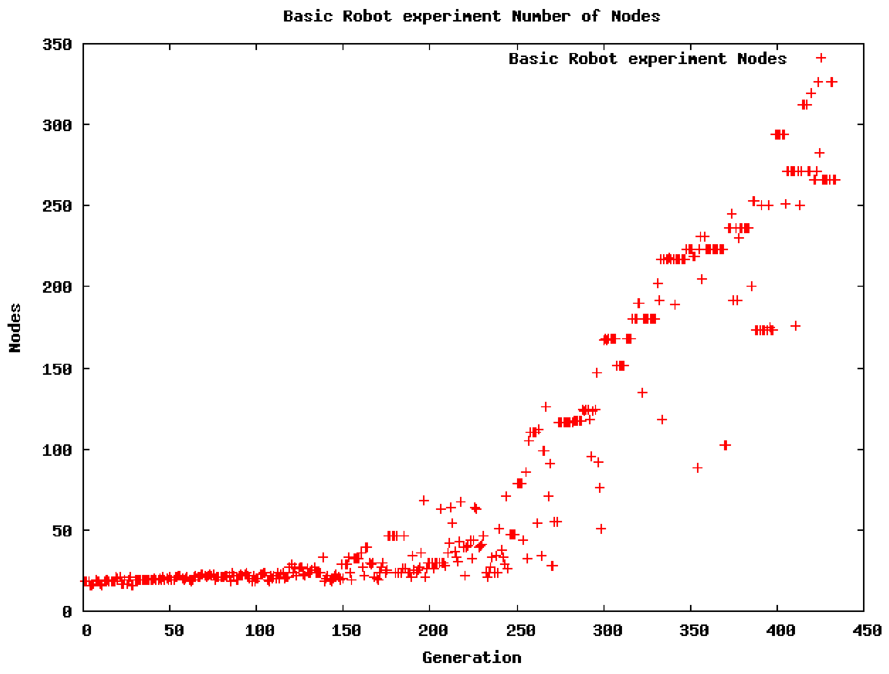


Figure 4.14: Food Foraging best run number of nodes

Each run of the robot control simulation took between 5 and 7 hours on a 3.4 ghz AMD Phenom II processor. Again, several of these experiments were run in parallel. The experiment was able to find a solution in four of the five runs. The fifth run was incomplete due to hardware issues.

Examining the results shows that the average fitness again follows the expected results curve (Figure 4.11). Comparing this fitness with the number of nodes in the best solution (Figure 4.12) shows that early fitness improvements were made with few additional nodes. However, as the ideal solution was approached, the number of nodes needed increases exponentially. Also of note is the specific time it takes for a solution to be found. While Stanley et al. (2003) was able to reliably find recurrent solutions for the more difficult dangerous food foraging problem, the results from Figure 4.13 show that even in the best case, this implementation takes just over 400 generations to converge on a solution. This result will be examined more thoroughly in the conclusion.

4.4 Dangerous Food Foraging

The dangerous food foraging experiment is a reimplementation of the experiment described in Stanley et al. (2003). If the implementations were identical, the results should match those found by Stanley. However, as this is a reimplementation based on a general description without all of the specific details available, one should expect a similar, but not necessarily identical performance.

Experiment Setup

The code for this experiment is virtually identical to the code used for the basic food foraging experiment (See section 4.3). The description of the function of the experimental world is the same, as is the description of the function of the robot. The only difference in implementation is that the experiment was changed so that in half of the instances of the experiment world each generation, the food was poisonous. This matches the experiment described in Stanley et al. (2003). Related to this addition, a pain sensor was added to the robot that would be triggered for 20 steps if poison were consumed.

The addition of poison food items necessitated a change in the fitness function from counting the total number of items consumed to being a total of the good items of food consumed. This gives the exact fitness function used in the original research (see equation 3.1). While the range for potential fitness values remains 0 to 64, the desirable fitness value is now 60. The only way for the robot to discover if the world contains food or poison is to consume at least one item. Therefore, at least four poison items must be consumed to determine the correct course of action.

Results

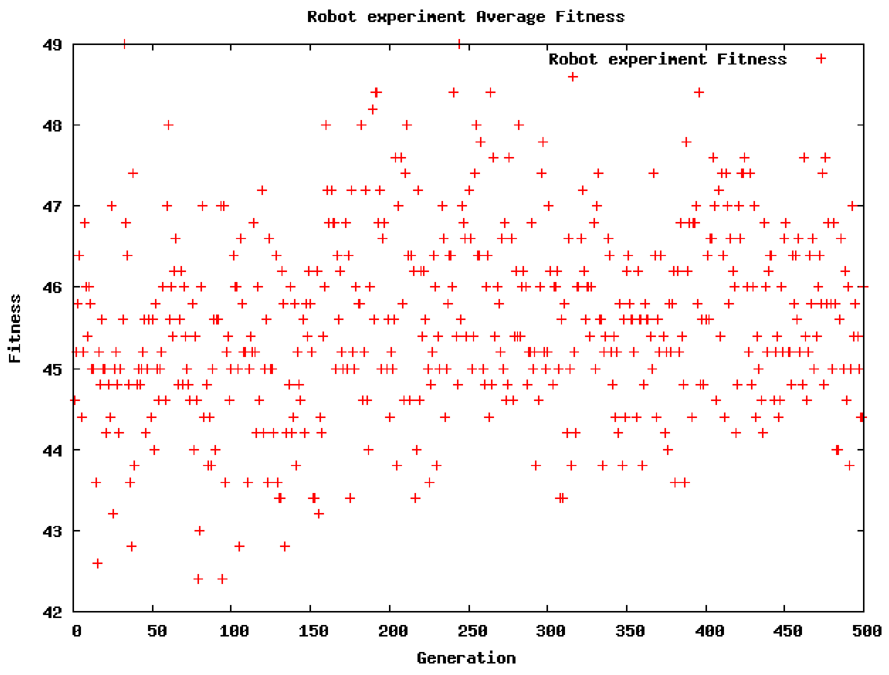


Figure 4.15: Dangerous Food Foraging average fitness

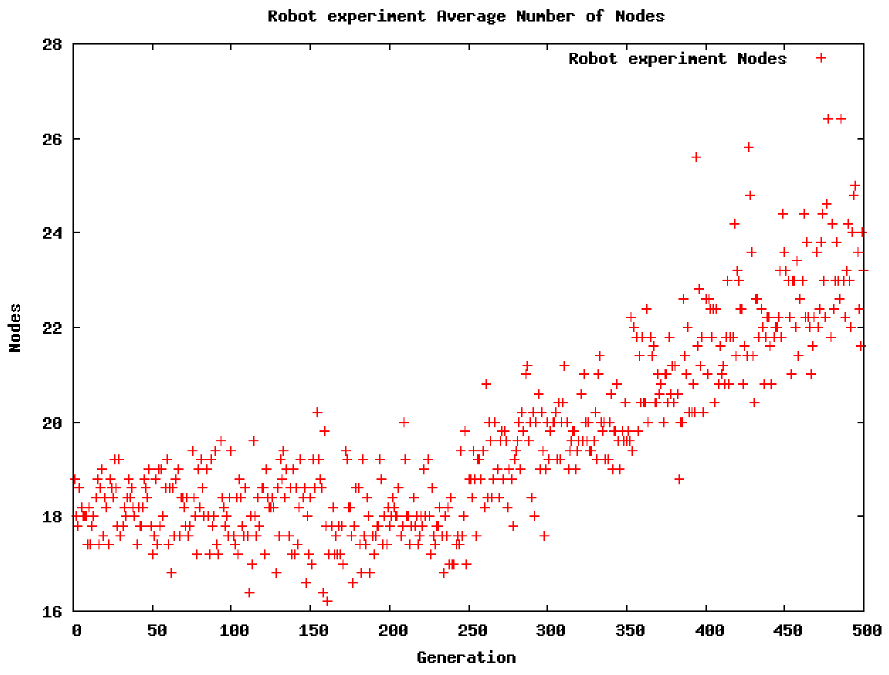


Figure 4.16: Dangerous Food Foraging average number nodes

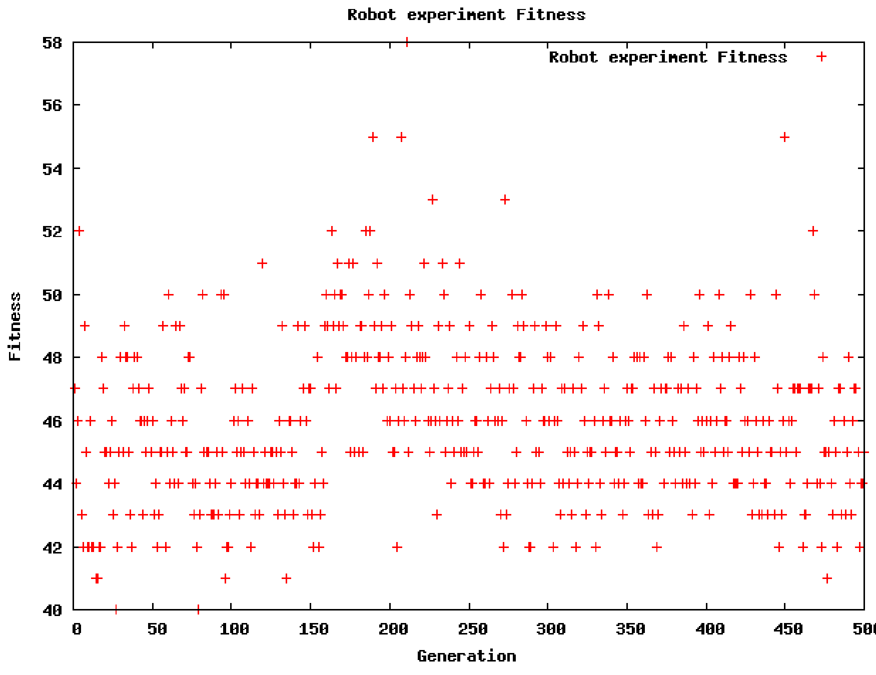


Figure 4.17: Dangerous Food Foraging best run fitness

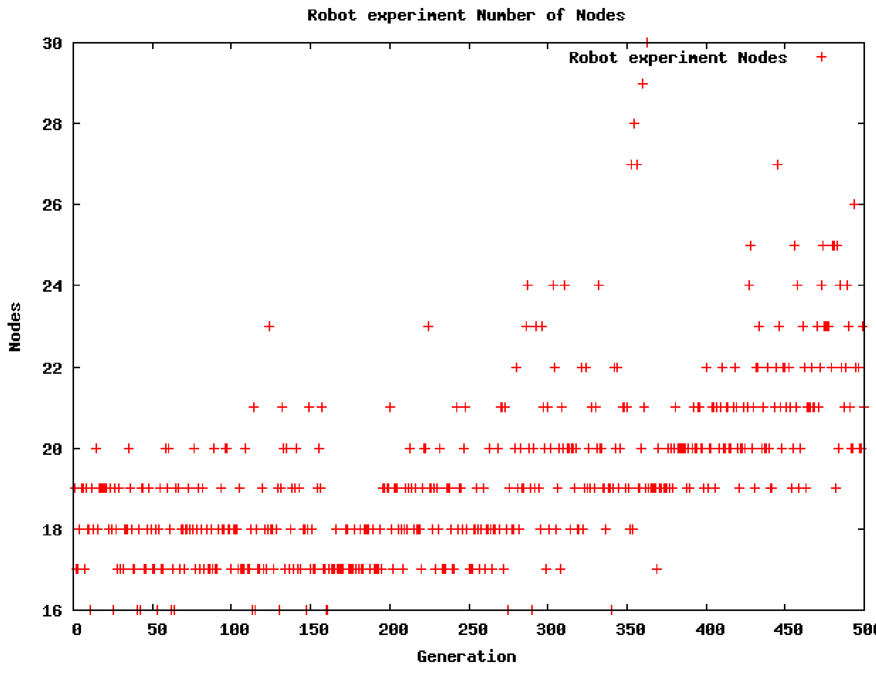


Figure 4.18: Dangerous Food Foraging best run number of nodes

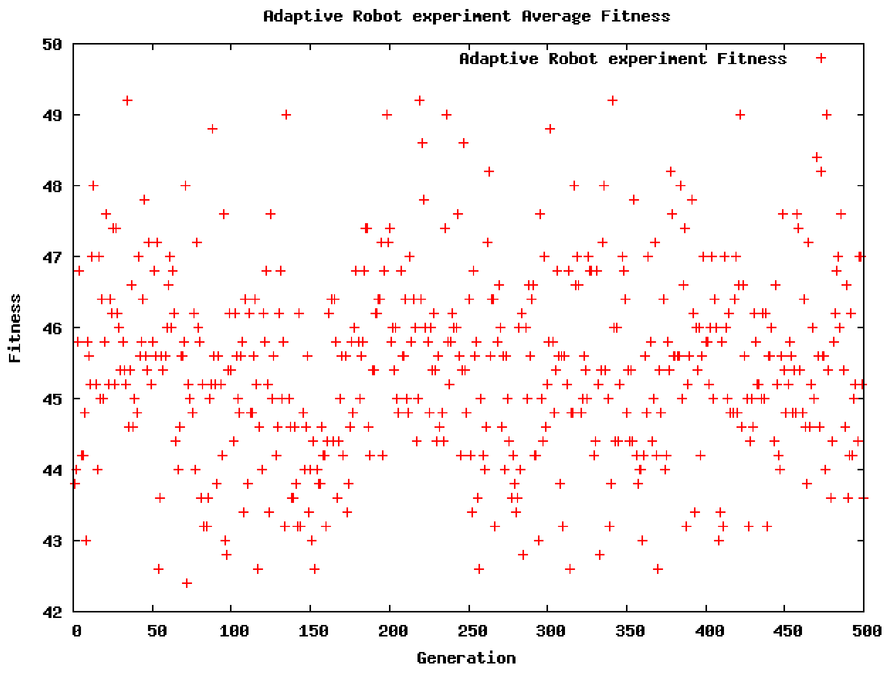


Figure 4.19: Dangerous Food Foraging average adaptive fitness

Each run of the dangerous food foraging experiment took between 4 and 7 days on a 3.4 ghz AMD Phenom II. Initial evaluations were completed quickly, but as additional structure was added each iteration slowed to taking over an hour. The average fitness no longer shows an upward trend (Figure 4.15 and no solutions were found in any 500 generation run. This stands in contrast to the results found by Stanley et al. (2003). Examining the complexity of the potential solutions, Figure 4.16 shows that the size of the neural network did not increase as quickly as it did in the non-dangerous foraging domain (compare to Figure 4.12).

Similar results are seen through the five adaptive runs. The fitness represented in Figure 4.19 is effectively random. Additionally, the increase in the number of nodes in the best elements in the population is very slow (Figure 4.20).

The fitness and node complexity results differ significantly from what was found in the basic food foraging domain. Figure 4.23 compares the two directly. While in the

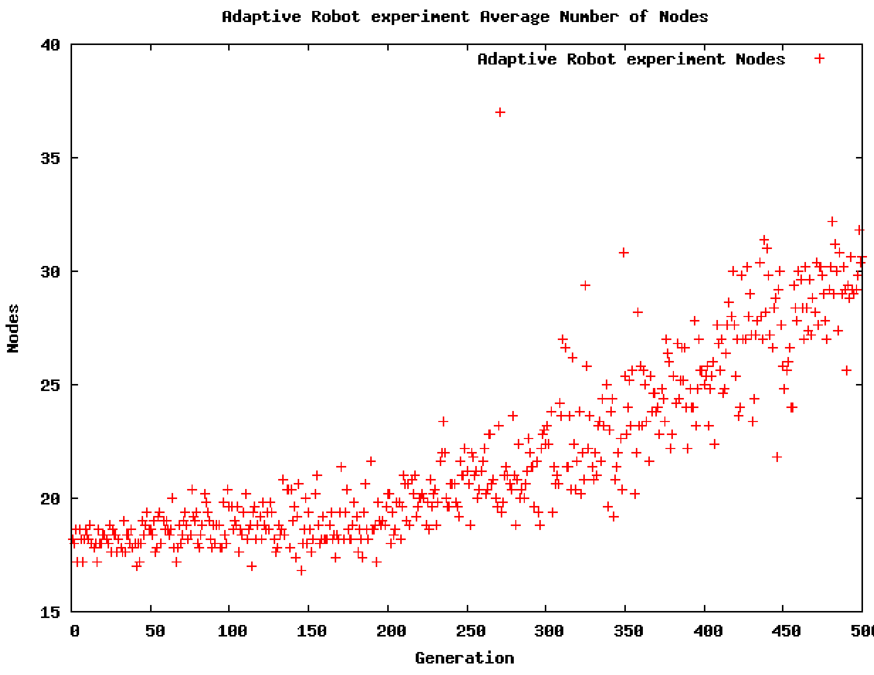


Figure 4.20: Dangerous Food Foraging average adaptive number nodes

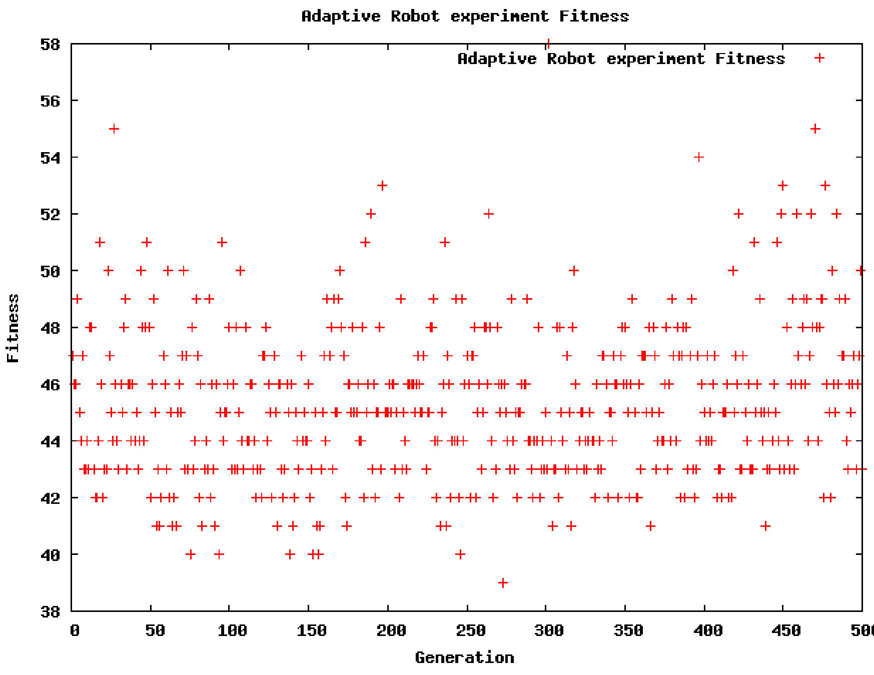


Figure 4.21: Dangerous Food Foraging best adaptive run fitness

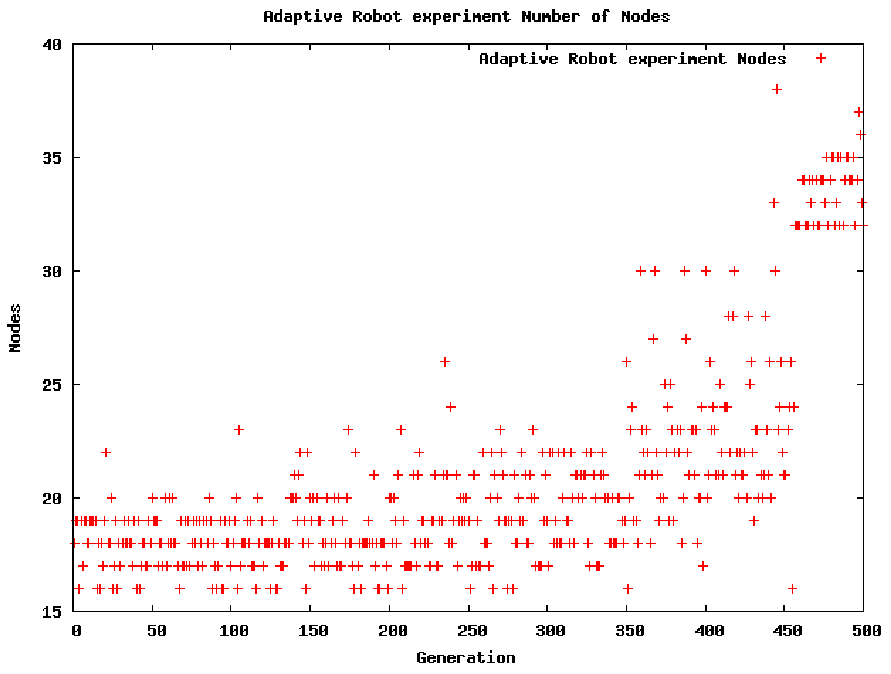


Figure 4.22: Dangerous Food Foraging best adaptive run number of nodes

safe food foraging domain, the evolutionary controller is able to improve gradually. Meanwhile, when half of the items are poisonous, the evolutionary process cannot discover any fitness improvements. The difference is even more drastic when comparing the node complexity between the two, as seen in Figure 4.24 While the dangerous food foraging realm eventually has 30 nodes the safe domain has evolved over 400. This can only be a result of the problem domain itself, because the only changes were the addition of poisonous food elements. No other parameters were changed.

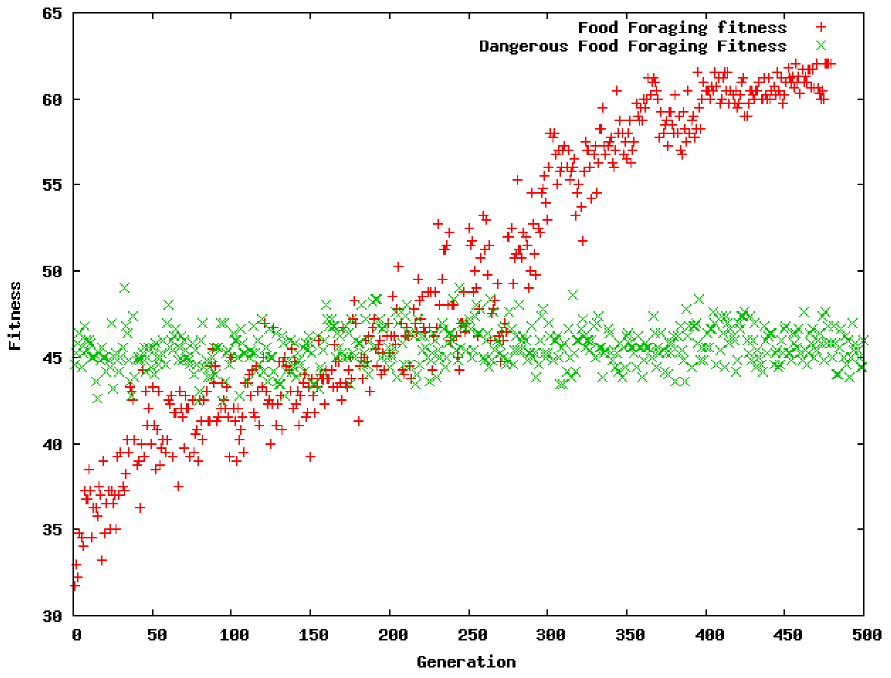


Figure 4.23: Comparison of the fitness of safe vs. dangerous food foraging experiments.

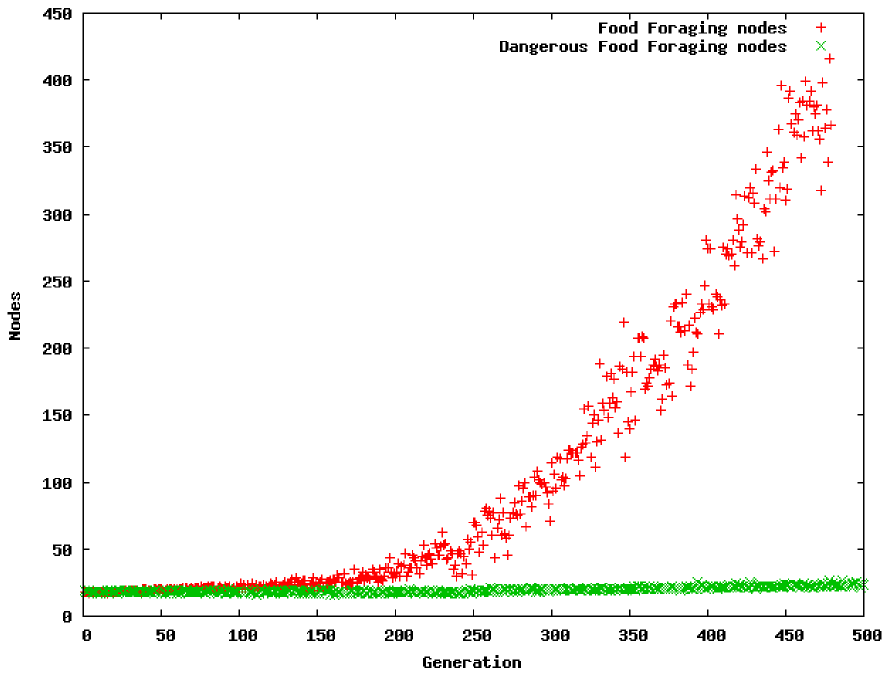


Figure 4.24: Comparison of the number of nodes between safe vs. dangerous food foraging experiments.

5 Flight Control Experiment

The goal of this experiment is to take the NEAT approach to evolving adaptive networks and see whether the results of Stanley et al. (2003) scaled to a larger problem domain. In order to do this, it was necessary to find a problem domain that required exploration of the environment in order to find successful solutions, as well as required more complex behavioral decisions than those found in the dangerous food foraging domain.

5.1 Problem Domain

The overall problem area decided upon was the adaptive control of a simulated aircraft. Controlling aircraft in flight requires a much more complicated set of behavioral decisions than the rather simple robots used in Floreano and Urzelai (1999); Stanley et al. (2003). To necessitate adaptive behavior requiring exploration of the environment, the neural controller would be required to adapt to flying different models of aircraft. While the sensor input and control output of each aircraft is identical, the performance of the aircraft would change depending on the model of aircraft being controlled.

5.1.1 JSBSim

To simulate the flight of aircraft, the open source flight simulator JSBSim was used. JSBSim was designed to be extremely available tool to aid in the design of aircraft and aircraft controls. Its implementation is described in Berndt (2004). JSBSim has previously been used in neurocontroller research as well (Gomez and Miikkulainen, 2003)

JSBSim was compiled as an external library, and then linked to the experiment, enabling its use. After each iteration, all JSBSim objects were deleted and reinitialized. During each iteration, in order to keep the runtime manageable, the ResetToInitialConditions function was used to return the simulated environment to its initial conditions. The library was only modified to allow for compilation with gcc 4.

5.1.2 The Aircraft

Three aircraft models were used as the primary aircraft for the flights. Each of the aircraft models are included with the JSBSim release. A P51 provides a high speed, low wing monoplane. This plane was intended to be the most difficult of the aircraft for the neural controller to adapt to. The second aircraft included is the Cessna 172R. It is a common, stable, easy to fly high wing aircraft. Finally Piper PA-28 Cherokee is included as a low wing monoplane that is simple to fly. If a single run was completed while moving through all waypoints and not crashing with all three of the aircraft, an additional test was performed with a Cessna 182 to ensure that the neural network was able to generalize its control approach to a new aircraft. The Cessna 182 is similar to the 172, but with more horsepower available. If the neural network can successfully fly the Cessna 182, it is considered a winning generation.

5.1.3 The Flight

Each generation of experiment creates a new set of flights, one for each aircraft. The parameters of each flight are identical for every neural controller running a single aircraft model.

Each flight starts off at a random altitude directly over the ECU science and technology building headed in a random direction. The altitude is initially set between 500 and 2500 feet above the ground. The initial velocity is set between 90 and 110 knots. This was verified to be sufficient for the aircraft to maintain flight. The aircraft is trimmed so that it has a sufficient speed and stability to maintain level flight. The objective of the flight is to fly from the initial location through a series of waypoints.

Each flight also generates a random set of waypoints through which the aircraft should fly. The first waypoint is directly in front of the airplane and a random altitude between 500 and 2500 feet above the ground located between 1000 and 3000 meters away. Subsequent waypoints will have the same random variation in altitude and distance from the previous waypoint, but may radially be located in any direction from the previous waypoint. Between 2 and 10 waypoints are generated. This means that the maximum distance needed to fly through all the waypoints is 30 kilometers. As each flight can last up to 15 simulated minutes, and all of the aircraft can travel near 200 kilometers per hour, there is sufficient time for any aircraft to reach each waypoint.

5.2 Networks

Both the fixed weight recurrent neural networks, and the adaptive neural network use the same set of parameters for input and output. The overall design of the neural network can be seen in 5.1. The only difference between the fixed weight networks and those with plastic synapses was whether or not adaptive synapses were allowed to evolve.

5.2.1 Inputs

The input layer of the flight controller was represented as a 17 element array. Each element of the array could take as an argument a double. The first three elements specified the current waypoint towards which the aircraft ought to be flying. This could be considered a tuple of longitude latitude and altitude. The next set of three

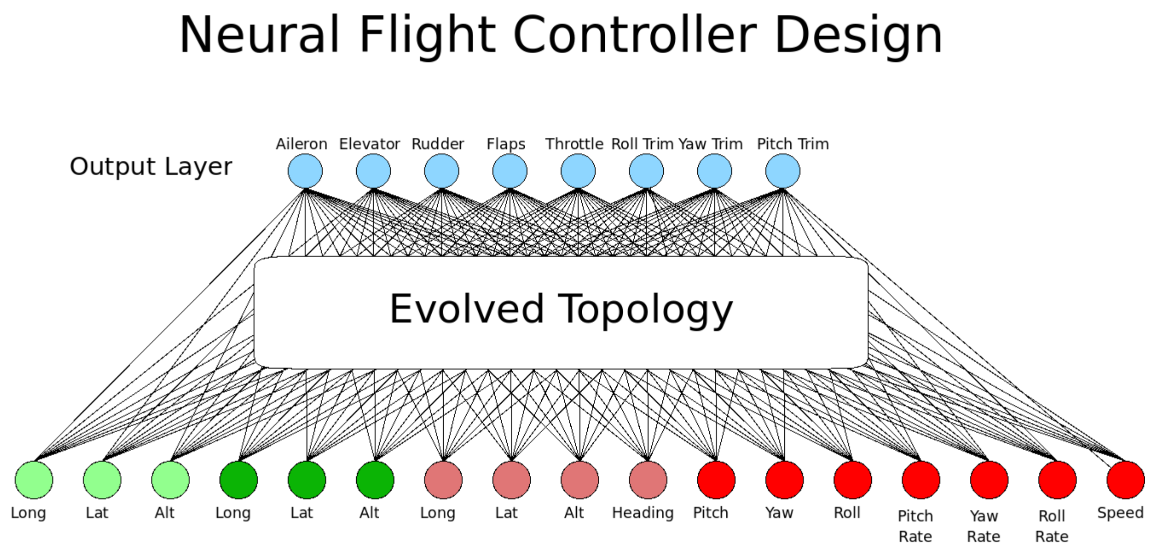


Figure 5.1: Neural Flight Controller ANN

inputs was a tuple representing the next waypoint that the aircraft should fly towards after it reached the location of the current waypoint. The seventh through ninth point is a tuple containing the current location of the aircraft. The 10th point is the current heading of the aircraft. The next three points detail the attitude of the aircraft. These are the Euler angles. These correspond with the pitch, yaw and roll of the aircraft. the next three input elements of the Euler rates of change. This allows the aircraft to be aware of how its attitude in space is changing. The final input parameter is the indicated airspeed. This is not necessarily the actual airspeed of the aircraft, but is instead the value that a sensor on the airplane would provide.

Furthermore, as the inputs are being fed into neurons that use sigmoid activation function, the input values are normalized to have a range between -1 and 1. This was done by determining the maximum value for each variable and dividing by that value.

Due to the possibility of bad output values from the neural network creating a divide by zero inside of JSBSim that propagates to the location of the aircraft (a possibility found in debugging), all input parameters are checked to make sure that they are a number. If an input parameter is not a number, the fitness of that controller is set to one, and testing of controllers completed. This problem did not occur in any of the actually recorded runs.

5.2.2 Outputs

After the inputs have been set, the network is activated, which gives output values ranging from 0 to 1 in an eight element output array. Each element of the output array corresponds with one of the available aircraft controls. As seen in 5.1, this allows the neural controller to set values for the aileron, elevator, rudder, flaps, throttle, and then roll pitch and yaw trim. These output values are then propagated into JSBSim while being normalized to the input values expected by the simulation kernel.

5.3 Population

The population is initialized as described in 2.4. The population size is set to 1000 controllers. This large population was chosen to increase the likelihood of finding a viable solution to the large search space that a complex problem like this entails. Additionally, an argument is provided to the population initialization function that describes whether or not adapted synapses are allowed. This is a run-time parameter specified on the command line. This was the same mechanism used for the earlier experiments.

5.4 Running the simulation

Each generation tested each neural controller in the population in the world created for each aircraft. each flight lasts a simulated 15 minutes, barring a crash previous to the completion of the flight. Since each step of the JSBSim simulator simulates 50 milliseconds, this leads to a total of 18,000 simulation stops. The neural controller operates at 1/5 of that rate. Thus for each neural network activation, five simulated time steps using the output values of neural network are made. At each step of the simulation, the location of the aircraft is checked for either crashing into the ground or succeeding in flying to a waypoint.

If the altitude of the aircraft is detected to fall below 50 feet, The flying parameter on the aircraft object is set to false. This will cause the next iteration of control on that flight not to begin with and affects the fitness of that neural controller. This is considered a crash.

Checking whether or not the current waypoint has been reached is done by comparing the current location and the location of the waypoint. In order for a waypoint to count as being reached the current location must be within 50 meters of longitude and latitude, and within 100 feet of altitude. Once a waypoint has been reached, the current waypoint parameter is set to the next waypoint, and the next waypoint is incremented to the waypoint after that. If there are no further waypoints for the next waypoint to be set to, all elements of the next point are set to 0.

5.5 Fitness Evaluation

Fitness is evaluated in two steps. Firstly, the fitness of a single neural controller in a single flight is evaluated using the fitness function detailed below. The fitness of each of the flights made by a single controller are then summed to determine the overall fitness of a controller in a specific generation. Second, if the overall fitness of a single neural-controller in a single generation is sufficient to declare it a winner[1],

[1] In this case, being declared a winner means that the fitness of the network is equal to or greater than the maximum fitness for that generation

then further testing is done of that controller to assure it is consistently capable of providing a generalized flight controller. The further testing is done by creating a new flight and attempting to fly using the Cessna 182.

5.5.1 Fitness Function

The fitness function is a summation of multiple factors. It is accumulated in two general ways. First, fitness increases by lengthening the time of flight. Each simulation time step that the aircraft is flying increases the fitness by one. Furthermore, each waypoint that has been reached by the flight adds 3000 to the fitness of controller. Finally, in order to help during the early search of the solution space, a bonus is added that is related to how close to the next waypoint the aircraft managed to get. The complete fitness of the neural flight controller is the sum of the fitness of each of the individual flights.

In equation form, the fitness of the flight is then:

$$f = t_{steps} + 3000W + \frac{3000\frac{5000}{d_{nearest}}}{5000} \tag{5.1}$$

In this case, f is the fitness of the flight, t_{steps} is the number of time steps in flight, W is the number of waypoints reached, and $d_{nearest}$ is the closest the aircraft had approached to the next waypoint. Additionally, if an airplane is still flying at the end of the 15 minutes, a bonus is added of the maximum number of time steps. In this experiment, that means a bonus of 18000.

Each generation has a maximum fitness calculated. The maximum fitness of a flight is the maximum number of time steps plus the number of waypoints times

3000. This is the equation:

$$f_{max} = 2t_{max} + 3000W \tag{5.2}$$

where f_{max} is the maximum fitness, t_{max} is the maximum time, and W is the number of waypoints reached. The total maximum fitness of a controller is then the sum of the maximum fitness of each of three flights that will be made. This leads to maximum fitness ranges between 123,000 and 204,000.

5.5.2 Winner Evaluation

If the fitness of any neural controller equal to or exceeds the calculated maximum fitness of that controller, is then tested with a fourth airplane (the Cessna 182) which is used to ensure generalization of the solution. If several generations of neural networks are capable of generalizing to the Cessna 182 and matching the calculated maximum fitness for that flight, then the experiment is determined to have converged on a solution, and no further generations are run.

6 Results

The flight experiment was run four times with only recurrent connections allowed. Each run of the recurrent flight control experiment took between 15 to 20 days on a 3 ghz Intel Core 2 duo. As the memory requirements for a single run were near 8 GB, only one run per computer could be performed at a time despite the computer having multiple cores.

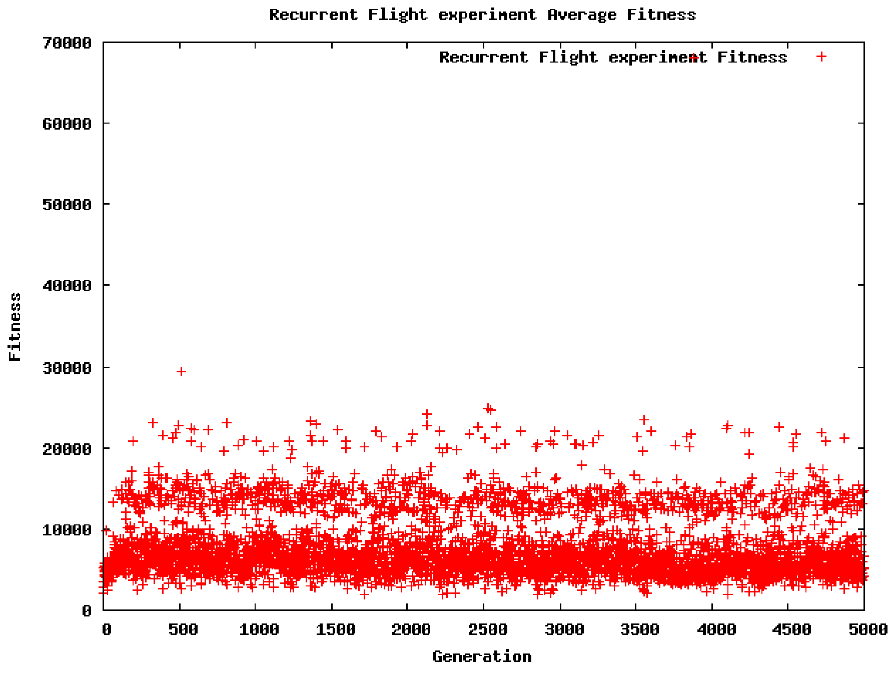


Figure 6.1: Recurrent Flight Controller average fitness

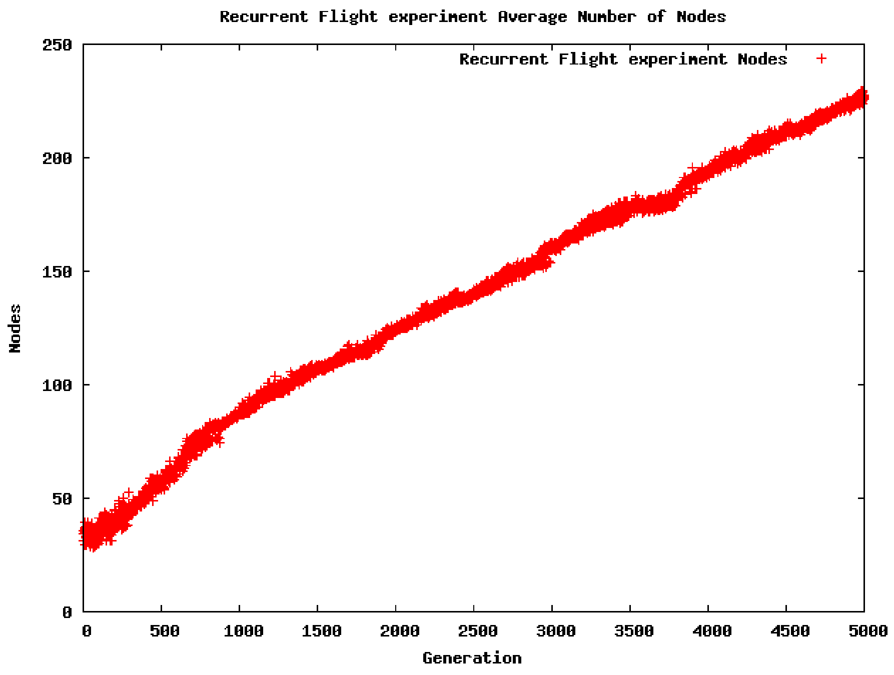


Figure 6.2: Recurrent Flight Controller average number nodes

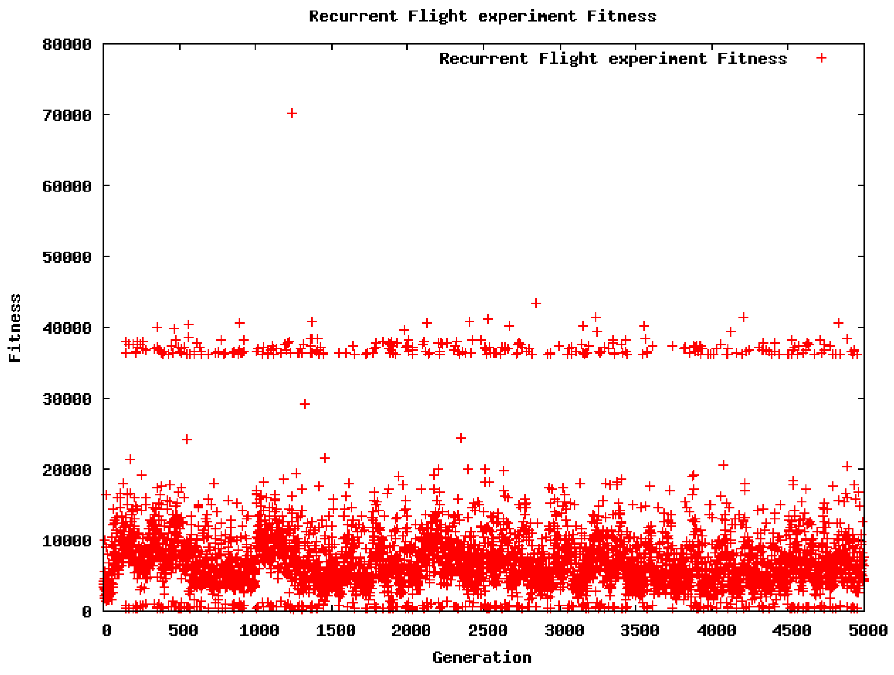


Figure 6.3: Recurrent Flight Controller best run fitness

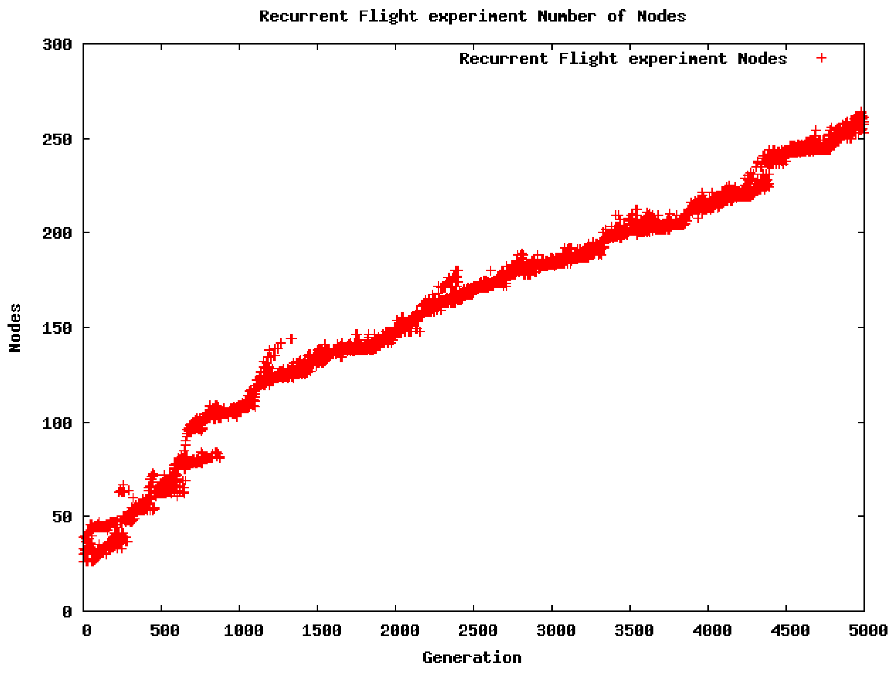


Figure 6.4: Recurrent Flight Controller best run number of nodes

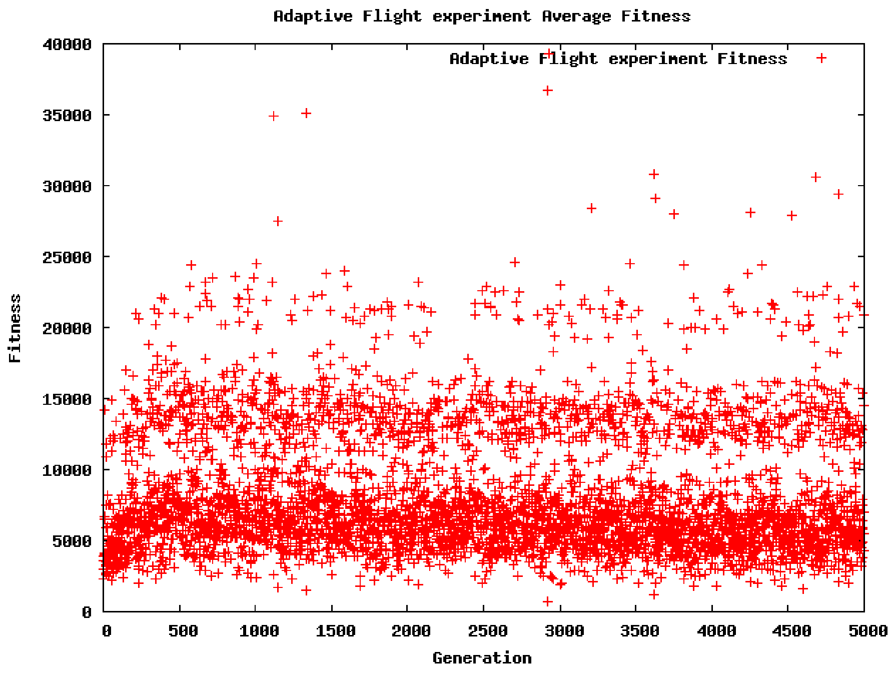


Figure 6.5: Flight Controller average adaptive fitness

Examining the results shows that the average fitness of the recurrent networks show little improvement (Figure 6.1). The graph shows three distinct bands of fitness results, but none of the improvement that would be expected to be generated by NEAT. Examining Figure 6.3 explains the banding seen in the average fitness. Recall that the fitness function applies a bonus of 18,000 to any flight which does not crash. While in general, the performance of each controller was poor, outliers in single flights were occasionally able to avoid crashing. These successes were indicated in early testing via output to the screen. This occasional success on a single flight garnered a bonus to the fitness at a fairly constant point just above the fitness of 38,000. When the four flights are averaged together, this results in one band 4500 more fit than the average, and a second, sparser band roughly 9000 more fit than the average from those instances where more than one run happened to stumble upon a full flight during the same generation.

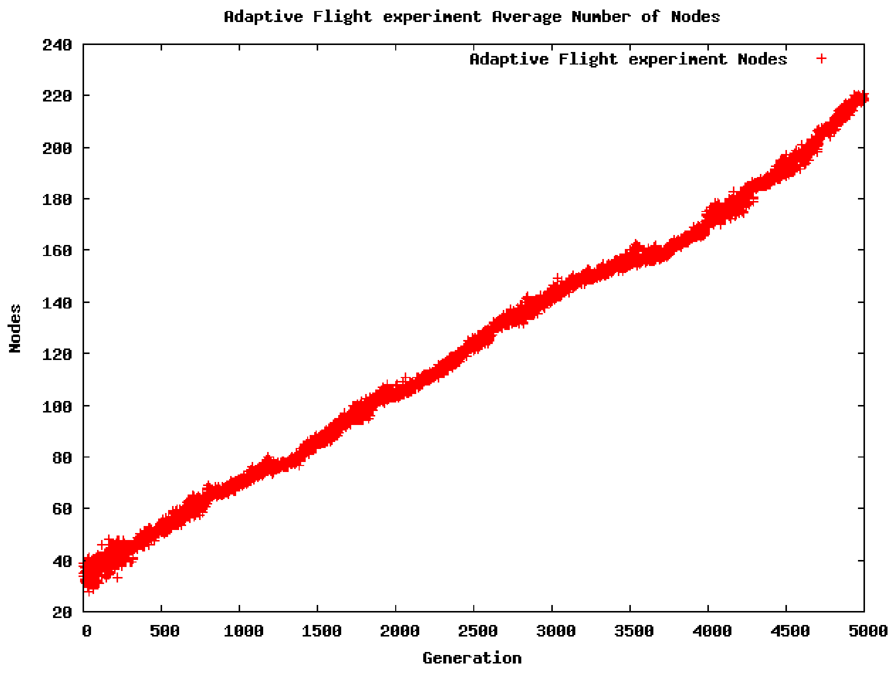


Figure 6.6: Flight Controller average adaptive number nodes

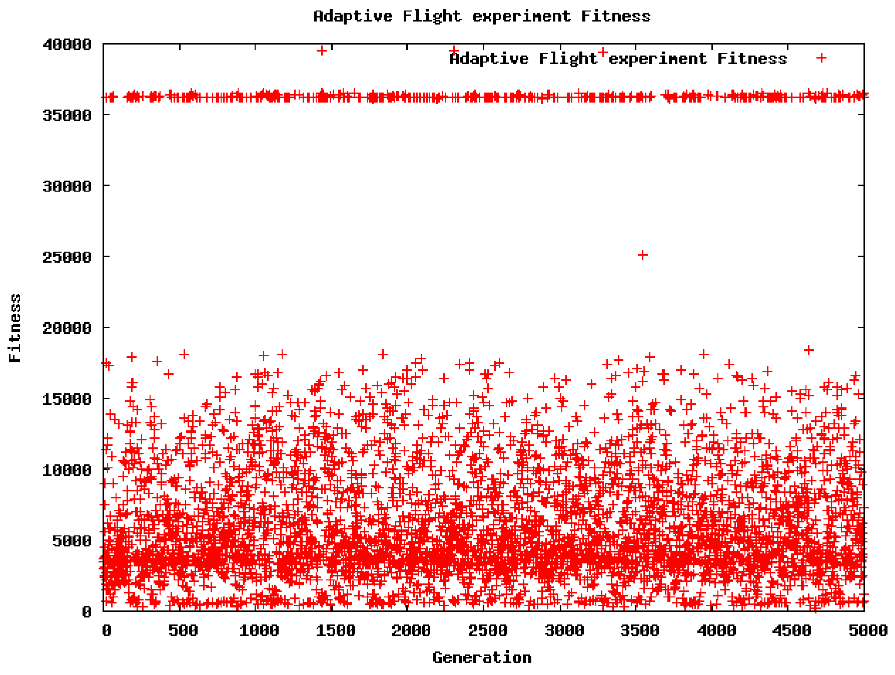


Figure 6.7: Flight Controller best adaptive run fitness

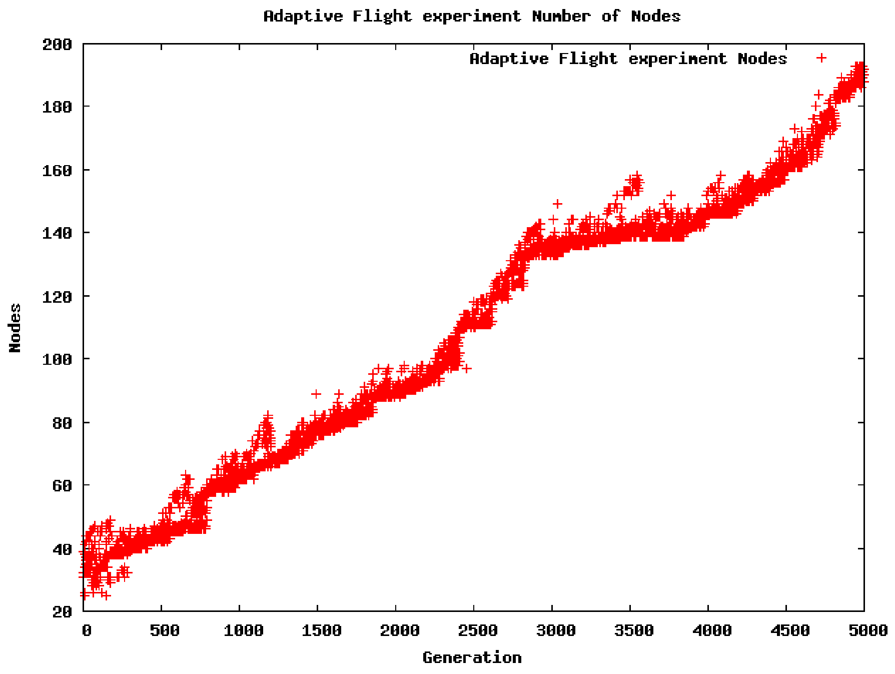


Figure 6.8: Flight Controller best adaptive run number of nodes

Examining the results from the adaptive runs shows almost identical performance to the recurrent only runs. Figures 6.5 and 6.5 both show the same sort of banding at roughly the same fitness levels. These results indicate that NEAT is unable to evolve any improvement in the neural controllers. Unlike the results found in the food foraging domain, the node complexity increases at a constant and reasonable pace (Figures 6.6 and 6.2).

6.1 Eliminated Problems

There are several problems that were considered that could have caused the failure to converge on a solution. Each of these potential changes to the experiment was tried to eliminate the possibility that it was the cause of the failure to convert.

6.1.1 Coding error

It is possible that the poor results are simply the result of a coding error. It is for that reason that the experiments detailed in chapter 4 were run. These established at least that the baseline implementation of NEAT was used correctly, as other problem domains were able to find viable controllers.

Proving that JSBSim was used correctly is more difficult. Firstly, it can be shown that JSBSim itself runs correctly. The library linked in to these experiments was also linked in to FlightGear, an open source flight simulator that JSBSim is capable of powering. Working through that program, flights using the JSBSim kernel seemed to model flight dynamics well. Proving that the code implemented against JSBSim is correct is more difficult. By inspection it certainly appears to conform with the guides on how to use JSBSim. Additionally, if the fundamental problem were with JSBSim, one would not have expected the Dangerous Food Foraging domain to also fail to find a solution. This is enough to indicate that the problem in evolving better controllers likely lies elsewhere.

6.1.2 Too Many Planes

One possibility is that attempting to evolve a neural controller to fly a set of airplanes adds unnecessary complexity to an already complicated problem domain. Therefore it may be possible to evolving neural controller capable of flying only a single airplane, but not capable of being generalized to fly any of a set of airplanes. The first thing to note about this potential problem is that this change removes impetus towards adaptivity that was originally designed into the experiment. To eliminate this as a potential problem, an experiment was designed to fly only a single airplane.

The overall design of the experiment was almost identical to the initial design of the experiment using three airplanes. The difference was that rather than constructing a list of three different airplanes to use for the experiment, only a single airplane(the P-51) was used. This ended up producing results similar to those found with multiple airplanes, only with smaller overall fitness values as one third the number of flights were attempted per iteration.

6.1.3 Fitness Function too Complicated

Another possible problem was that the fitness function, being a summation of time and flight and number of waypoints flown to was simply too complicated for the neural network to evolve to. Another experiment was concocted, again using only a single airplane, which used as its fitness value the maximum altitude of the aircraft achieved. This was a further modification of the previous single plane experiment which changed the fitness function to simply be the maximum altitude. At each time step, the current altitude was compared with the previous maximum altitude, and if the current altitude was greater than the previous maximum altitude, the previous max was set to the current altitude. Additionally, the iteration initialization was changed to specify a constant starting altitude, thereby eliminating fitness variations based on random starting altitude's.

Just as with the single playing experiment, the results that appear similar to those found in the full experiment.

6.1.4 Output Too Complicated

Another potential problem area is that the trim output parameters have effects on the aircraft that overlap the effects of the primary control surfaces. The thought here is that by simplifying the output parameters of the neural network, it would be more capable of effectively controlling the aircraft while in flight. To test this hypothesis a modification of the experiment test code was made to hardcode all trim output values to zero before those values were passed to the simulation engine. Given that NEAT starts with a minimally connected network, this should mean that the networks involved would make no use of those output parameters as they have no effect on the external environment. Again, as with the other attempted modifications of the experiment, this did not have any significant impact on whether or not a solution was converged upon.

6.1.5 Input Too Complicated

A final possible problem that was tested for was that there are too many input parameters to the neural network. The minimally connected nature of NEAT, and the way it adds structure should mean that input parameters that do not help the neural network will not end up being connected. However, to further test this, the experiment was run with the current and next waypoint parameters all set to zero.

The longitude and latitude parameters of the current location were also set to zero. Again, as with other attempts to eliminate potential problems, this did not lead to any solution being converged upon.

6.1.6 Locations Fixed

Another possible difficulty is that the representation of the longitude and latitude of waypoints as absolute locations creates difficulty for the neural network and computing how it ought to fly. A version of the single plane experiment was run with all locations translated to be relative, and the results continued to appear random.

6.1.7 Add Reinforcement Parameter

Another possibility is that we may need to add additional inputs that indicate to a running network how well it is doing. This reinforcement parameter was present in the food foraging experiment, but was not used in the design of the primary experiment.

No direct test of this was tried, however if this sort of reinforcement input were to be capable of solving the difficulties with convergence that were discovered in the experiment, then we should have expected that the simple altitude fitness function experiment would have succeeded in converging upon a solution. Since in all cases, the altitude of the aircraft is provided as an input to the neural network, and the altitude is being used as the fitness function this would have created a de facto reinforcement parameter.

Furthermore, a reinforcement parameter was available to the controllers in the dangerous food foraging experiment. It did not resolve the problems that were found in converging on a solution in that domain, and should not be expected to in the flight domain either.

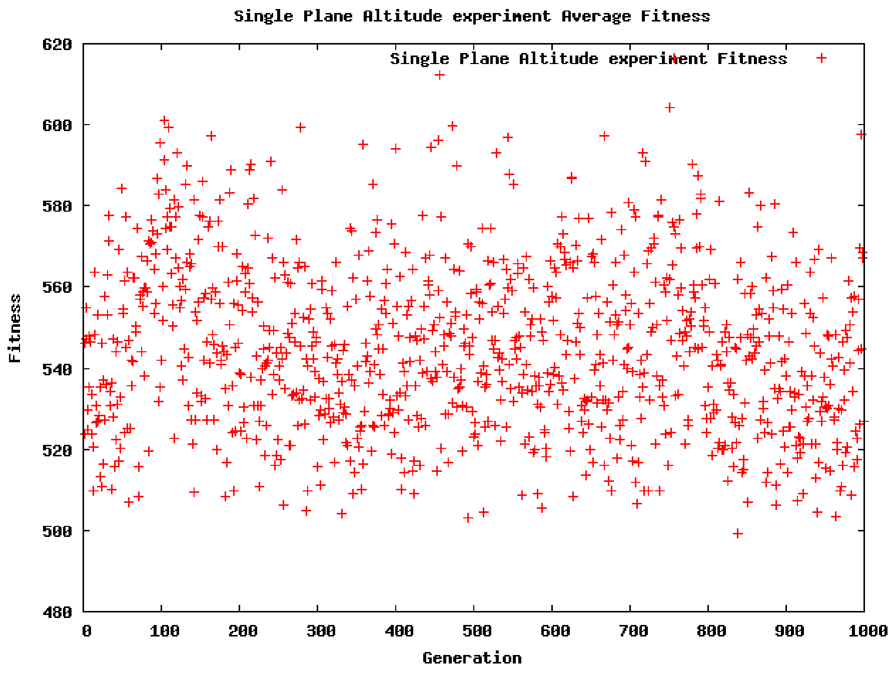


Figure 6.9: Flight Controller average maximum altitude.

6.1.8 Combination of Problems

As a final check, an experiment was run combining all of the tested hypothesis of what may be causing the experiment not to find solutions. In this case, a single P-51 is used with a simplified set of input and output parameters. The airplane starts at an altitude of 500 ft. The fitness function used is simply the maximum altitude. As seen in Figure 6.9, the results are essentially a random altitude. Just as in the full flight experiments, the complexity of the network grows linearly.

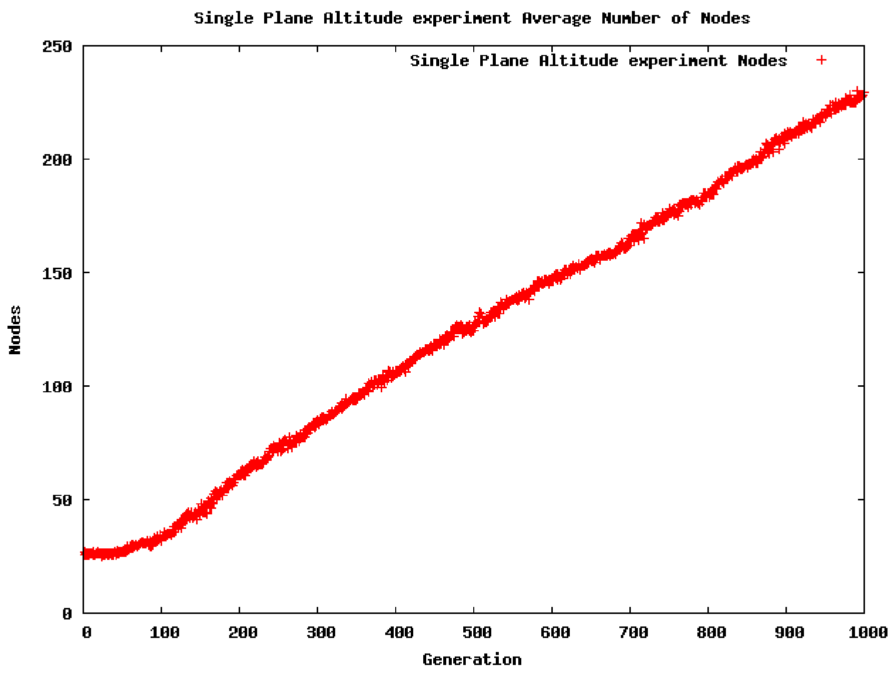


Figure 6.10: Flight Controller average number nodes

7 Conclusion

The results indicate that NEAT is not capable of scaling upwards to be some class of complicated control problems. Before looking at the causes of this failure to scale, it is important to consider some of the alternative untried approaches to solving this control problem.

7.1 Possible Alternative Approaches

While the results section discussed and eliminated as possible explanations several alternative approaches to the experiment there remain different approaches that were not tried for a number of different reasons

7.1.1 Alternative neuro-evolution pproach

Perhaps the most significant possibility in finding an neural controller capable of converging on a solution is to use a different neural evolutionary strategy. If for example the minimally connected nature of the initial network is the root of the problem, then using a traditional fixed topology network may be preferable. Another option would be to use HyperNEAT (Valsalam and Miikkulainen, 2009), a new extension of the neat approach. There are a number of possibilities here, which may prove fruitful avenues of research, but fall outside the scope of this thesis. Any change in the neuro-evolutionary back end would invalidate the comparative standard intended to be used to see whether or not the results of Stanley et al. (2003) scaled to a larger problem.

7.1.2 Alternative learning algorithm

An alternative may be to change the type of learning algorithm used. Recent work has shown that often times in complex control problems the fitness of a solution is not as important as the novelty of a solution in the initial exploratory phase of evolution. Novelty search, and a hybrid of novelty and fitness search has been proposed as a way of expanding the initial search space and finding better solutions (Risi et al., 2009; Soltoggio and Jones, 2009). Again, while this may prove a useful avenue of research going forward, it falls outside of the scope of this thesis, as it would invalidate comparative standard upon which this was based.

7.2 Limitations of NEAT

There are several limitations of NEAT that become relevant to these results. These limitations appear both in previous work, and are drawn out by the results.

7.2.1 Previous Work on Limitations

This is not the first set of experiments to discover that NEAT may fail to be able to solve certain problems. Linhardt and Butz (2009) found that while NEAT was able to make some initial improvements in the fitness for an experiment in the control of a robot arm, it was unable to solve the control problem. This work referred to an earlier paper (Kohl and Miikkulainen, 2008) that showed that NEAT struggles to find solutions in a fractured control space. In a fractured control space, the appropriate action changes suddenly based on the input, not gradually. Experiments have shown that NEAT has difficulties in solving problems that exist in a fractured control space. While the Dangerous Foraging domain may be a problem in a fractured control space, the flight controller definitely is. This may explain some of the difficulty in finding results.

7.2.2 Implementation Sensitivity

Perhaps more interesting is that in this re-implementation of the food foraging problem there was a difficulty in even improving upon the initial solutions. It appears that if initial mutations that add nodes can not gain an evolutionary advantage, those increases in the search space are not maintained long enough for an adequate enough structure to be built for there to be improvement in the overall fitness. In the case of the dangerous food foraging results, the best options were not those with a few more nodes than others, but stayed nearly constant at roughly 20 nodes for most of the experiment. The mutation parameters were identical to those in the basic food foraging experiment. This indicates that while new nodes were being added to new species, these additions were not met with an associated fitness advantage which justified keeping the additional nodes. This change in evolutionary behavior was brought about by a very small change in the implementation.

7.3 Reasons for the Results

Why was NEAT not capable of evolving a solution to these particular control problems? While there are no concrete results to confirm this, my hypothesis is that more hidden internal network complexity is required in order for a solution to be found. The neural flight controller requires a number of internal functions to be computed each of which would require its own subnetwork, taking certain sets of inputs and calculating outputs to be used in future decisions. Without some of this internal functionality available, the overall network cannot make adequate progress towards a solution to allow the evolutionary process to work.

I believe that this implementation of the dangerous foraging domain may have similar issues. Along with running in a fractured control space, the dangerous food foraging domain requires a set of substructures to exist in order for it to be able to effectively navigate its world. In the case of the original Stanley et al. (2003) experiment, the functionality of the sensors and the timing of the control mechanisms may have just been ideal to the discovery of solutions.

Because NEAT starts with a minimally connected network, and these subnetwork functions do not directly affect the fitness of a solution, they may never develop. However, because these sub-functions would be necessary for an overall solution to evolve, their failure to be discovered may prevent any progress towards an adequate solution. Instead, the fitness of the neural networks is left only to the random initial conditions of each generation.

7.4 Future Work

Further work may progress in several ways to confirm this hypothesis. One option would be to further decompose the input parameters into multiple related sets of parameters. For example, rather than simply keeping track of the altitude, we could keep track of the current altitude, altitude rate of change, positive altitude rate of change, negative altitude rate of change, positive altitude change, negative altitude change, and overall altitude change. While all of these should be discoverable and calculable by recurrent neural network, providing each of these individual parameters remove that pre-calculation step requirement from the evolution of the neural network. Similar decomposition of each of the input parameters would be required.

Another possible option is to independently evolve sub-control structures with their own fitness functions. While this may not be possible with the dangerous foraging experiment, it is workable in the flight control domain. Rather than evolving just a single controller for the entire aircraft, a controller for altitude only could be evolved, taking as input and output only those parameters needed to make decisions about altitude. Other sub-controllers could also be evolved. While this may not end up creating a fully capable neural controller, it could establish that the amount of work required in this implementation was simply overwhelming.

Finally, perhaps the most interesting result of this thesis is that a different implementation of a problem known to be solvable could, with a reimplementation, prove unsolvable to NEAT. Further work could proceed in categorizing what exact differences between the solvable and unsolvable implementation exist, and looking to understand why those differences are salient.

While these results show that NEAT has some limitations, understanding those limitations is key in improving the ability to develop sophisticated controllers. Further work in this area may lead to more powerful methods being developed that are able to generate adaptive controllers. Furthermore, it may allow a better understanding of the advantages and disadvantages that plastic neurons present. These results point out areas where further work is clearly needed.

BIBLIOGRAPHY

Attik, M., Bougrain, L., and Alexandre, F. (2005). Neural network topology optimization. *Lecture Notes in Computer Science*, 3697:53.

Auer, P., Burgsteiner, H., and Maass, W. (2007). A learning rule for very simple universal approximators consisting of a single layer of perceptrons. *Neural Networks*.

Berndt, J. (2004). Jsbsim: An open source flight dynamics model in c++. *AIAA Modeling and Simulation Technologies Conference and Exhibit*.

Beyeler, A., Zufferey, J., and Floreano, D. (2009). Vision-based control of near-obstacle flight. *Autonomous Robots*, pages 1–19.

Chalmers, D. J. (1990). The evolution of learning: An experiment in genetic connectionism.

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

Daqi, G. and Shouyi, W. (1998). An optimization method for the topological structures of feed-forward multi-layer neural networks. *Pattern recognition*, 31(9):1337–1342.

de Castro, L. (2007). Fundamentals of natural computing: an overview. *Physics of Life Reviews*, 4(1):1–36.

Emmert-Streib, F. (2006). Influence of the neural network topology on the learning dynamics. *Neurocomputing*, 69(10-12):1179–1182.

Ferrari, S. and Stengel, R. (2004). Online adaptive critic flight control. *Journal of Guidance Control and Dynamics*, 27:777–786.

Floreano, D. (1998). Evolutionary re-adaptation of neurocontrollers in changing environments.

Floreano, D. and Nolfi, S. (1997). Adaptive behavior in competing co-evolving species.

Floreano, D. and Urzelai, J. (1999). Evolution of neural controllers with adaptive synapses and compact genetic encoding. *Lecture Notes in Computer Science*, pages 183–194.

Ge, S., Hang, C., and Zhang, T. (1997). Direct adaptive neural network control of nonlinear systems. *American Control Conference, 1997. Proceedings of the 1997*, 3.

Gomez, F. and Miikkulainen, R. (2003). Active guidance for a finless rocket using neuroevolution. *Lecture Notes in Computer Science*, pages 2084–2095.

Hagan, M. and Demuth, H. (1999). Neural networks for control. *American Control Conference, 1999. Proceedings of the 1999*, 3.

Jansen, T. (2001). On classifications of fitness functions. pages 371–385.

Keane, A. J. (2001). An introduction to evolutionary computing in design search and optimisation. pages 1–11.

Kirkpatrick, S., Gelatt, C., and Vecchi, M. (1983). Optimization by simulated annealing. *Science*, 220:671–680.

Kohl, N. and Miikkulainen, R. (2008). Evolving neural networks for fractured domains. *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation.*

Konen, W. and Bartz-Beielstein, T. (2009). Reinforcement learning for games: failures and successes. *Proceedings of the 11th annual conference companion ....*

Linhardt, M. and Butz, M. (2009). Neat in increasingly non-linear control situations. *GECCO '09: Proceedings of the 11th annual conference companion on Genetic and evolutionary computation conference.*

Luger, G. F. (2001). *Artificial Intelligence: Structures and Strategies for Complex Problem Solving.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Minsky, M. and Papert, S. (1988). Perceptrons. pages 157–169.

Monroy, G., Stanley, K., and Miikkulainen, R. (2006). Coevolution of neural networks using a layered pareto archive. *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 329–336.

Moriarty, D. E. and Miikkulainen, R. (1998). Forming neural networks through efficient and adaptive coevolution.

Noriega, J. and Wang, H. (1995). A direct adaptive neural network control for unknown nonlinearsystems and its application. *American Control Conference, 1995. Proceedings of the*, 6.

Padhy, N. (2007). *Artificial Intelligence and Intelligent Systems.* Oxford University Press, New Delhi, India.

Pallett, T. and Ahmad, S. (1993). Adaptive neural network control of a helicopter in vertical flight. *Aerospace Control Systems, 1993. Proceedings. The First IEEE Regional Conference on*, pages 264–268.

Pardoe, D., Ryoo, M., and Miikkulainen, R. (2005). Evolving neural network ensembles for control problems. *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1379–1384.

Pearlmutter, B. A. (1996). Gradient calculations for dynamic recurrent neural networks: A survey.

Principe, J. C., Euliano, N. R., and Lefebvre, W. C. (1999). *Neural and Adaptive Systems: Fundamentals through Simulations with CD-ROM.* John Wiley & Sons, Inc., New York, NY, USA.

Reisinger, J. and Miikkulainen, R. (2007). Acquiring evolvability through adaptive representations. *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation.*

Risi, S., Vanderbleek, S., Hughes, C., and Stanley, K. (2009). How novelty search escapes the deceptive trap of learning to learn. *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation.*

Rojas, R. (1996). Neural networks: A systematic introduction.

Rowe, J. E. (2001). The dynamical systems model of the simple genetic algorithm. pages 31–57.

Soltoggio, A. and Jones, B. (2009). Novelty of behaviour as a basis for the neuro-evolution of operant reward learning. *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation.*

Stanley, K. (2004). Efficient evolution of neural networks through complexification.

Stanley, K., Bryant, B., and Miikkulainen, R. (2003). Evolving adaptive neural networks with and without adaptive synapses. *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, 4.

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

Thomson, J., Jha, R., and Pradeep, S. (2004). Neurocontroller design for nonlinear control of takeoff of unmanned aerospace vehicles. *Proceedings of the 42nd AIAA Aerospace Sciences Meeting and Exhibit, Reno, NV.*

Urzelai, J. and Floreano, D. (2001). Evolution of adaptive synapses: Robots with fast adaptive behavior in new environments. *Evolutionary Computation*, 9(4):495–524.

Valsalam, V. and Miikkulainen, R. (2009). Evolving symmetric and modular neural networks for distributed control. *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation.*