Exploring Fractional Order Calculus as an Artificial Neural Network Augmentation

by

Samuel Alan Gardner

A project document submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

April, 2009

APPROVAL

of a project document submitted by

Samuel Alan Gardner

This project document has been read by each member of the project committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the Division of Graduate Education.

Dr. John Paxton

Approved for the Department of Computer Science

Dr. John Paxton

Approved for the Division of Graduate Education

Dr. Carl A. Fox

STATEMENT OF PERMISSION TO USE

In presenting this project document in partial fulfullment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this project document by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this project document in whole or in parts may be granted only by the copyright holder.

Samuel Alan Gardner

April, 2009

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

ABSTRACT

Fractional calculus has been credited as being the natural mathematical model for power-law relations. These relations are often observed as accurate descriptors for natural phenomena. This project seeks to explore potential advantages that might be gained in applying fractional calculus to artificial neural networks.

A typical artificial neural network (NN) was augmented by applying the differ-integral operation from fractional calculus to the data stream through each neuron in the neural network. The NN and resulting fractionally augmented neural network (FNN) were compared within the context of evolution based learning, on a fox/rabbit artificial life simulation. Several experiments were run to compare the two network types in multiple evolution scenarios. The comparison was performed on the bases of (1) achieved fitness, (2) behavioral differences and (3) simulation specific metrics.

A graphical user interface (GUI) for a generalized evolutionary algorithm (EA) was developed to run the experiments and collect data required for the network type comparisons in the context of each experiment. Path diagrams indicated some potential differences between the NN and FNN in evolved behavior. T-tests of 95% confidence showed that their fitness results were no different for any experiment in this work, with the exception of topology size variation. Therefore, no direct advantages of the fractional augmentation were observed.

Some effects of applying the fractional augmentation were explored. Analysis of the experimental results revealed interesting directions for future exploration.

CHAPTER 1

INTRODUCTION

Artificial neural networks (NNs) have been studied extensively[1, 2, 3, 4]. Numerous variations in basic structure and operation[5, 6, 7, 8] have been investigated, as well as changes in the learning algorithm[9, 10]. Many of these alterations have improved upon the basic NN in certain applications as in [10].

This project proposes another variation on the classic NN architecture. It is an augmentation based in fractional calculus. Fractional calculus has a long history dating back to the 1600s with recent growing interest in its potential applications. In his 2000 doctoral thesis[11] Bohannan connects fractional calculus to power-law dynamics as the natural mathematics of power-law relations. He further hypothesizes that a broad range of measured phenomena are best described as exhibiting power-law dynamics. This puts fractional calculus in a position to accurately model that broad range of phenomena in a natural way. This idea has been successfully put to use in applied fractional order control (FOC)[12, 13, 14]. In an attempt to allow NNs to internally model the broad range of problem spaces exhibiting power-law dynamics, the differintegral was integrated into the classic NN architecture. In the simplest case, this was expected to provide lossy, long-term memory, intrinsic to the network structure.

A problem space allowing for a provided benefit from memory utilization seemed ideal for testing, so an artificial life simulation was developed in an attempt to fit this criteria. Neural networks with the fractional augmentation (FNNs) were compared directly against classic NNs in the artificial life simulation. The comparison was based primarily on achieved fitness, but also examined behavioral differences and simulation

specific performance metrics.

Most training was performed on fixed topology networks through methods that adjust the network parameters (weights and orders of differintegration). This allowed for relatively simple comparison between the network types, but some experimentation with learned topologies was also desired. Virtually any evolutionary algorithm (EA) may be used for parametric learning, but structural mutation requires specific modification of the algorithm to handle the additional complexity. Several such modifications exist [15, 16]. The software solution created for this work implemented a variation on the *GeNeralized Acquisition of Recurrent Links* (GNARL) algorithm[17] developed by Angeline, Saunders, and Pollack. GNARL was specifically designed to simultaneously evolve NN structures and parameters. Additional components for a typical genetic algorithm (GA) were also developed in this work for comparison.

Chapter 2 provides an overview of artificial neural networks, learning algorithms, and fractional calculus with a narrow focus on aspects of each relating to this work.

Chapter 3 describes relevant implementation details regarding the experiments performed for this work. This includes an overview of the neural network model and the learning algorithm used. Specifics of the fractional order calculus augmentation are also described.

Chapter 4 explains the problem domain with a detailed description of the artificial life simulation. Tables of experimental parameters common among the experiments run for this work are provided. Each experimental setup and its results are discussed along with their implications. A summary of the key results is provided at the end of this chapter.

Chapter 5 discusses numerous possibilities for continued exploration of fractional calculus as applied to neural networks. Chapter 6 summarizes the conclusions that

can be drawn from this work.

Appendix A describes the custom software solution developed for this work. Screen shots of its operation and a discussion of its features are provided.

Appendix B contains partial source code listings for the selected parts of the custom software solution considered most relevant to gathering experimental data for this work.

CHAPTER 2

BACKGROUND

Artificial Neural Networks

An artificial neural network (NN) consists of a set of processing units, also referred to as neurons or nodes, connected to one another by directed links to form a network.

Neurons

Each neuron in a NN functions independently of all other neurons in the network. As shown in Figure 1 each neuron has a set of n input links $[x_0,\ x_1,\ \ldots\ x_{n-1}]$ with

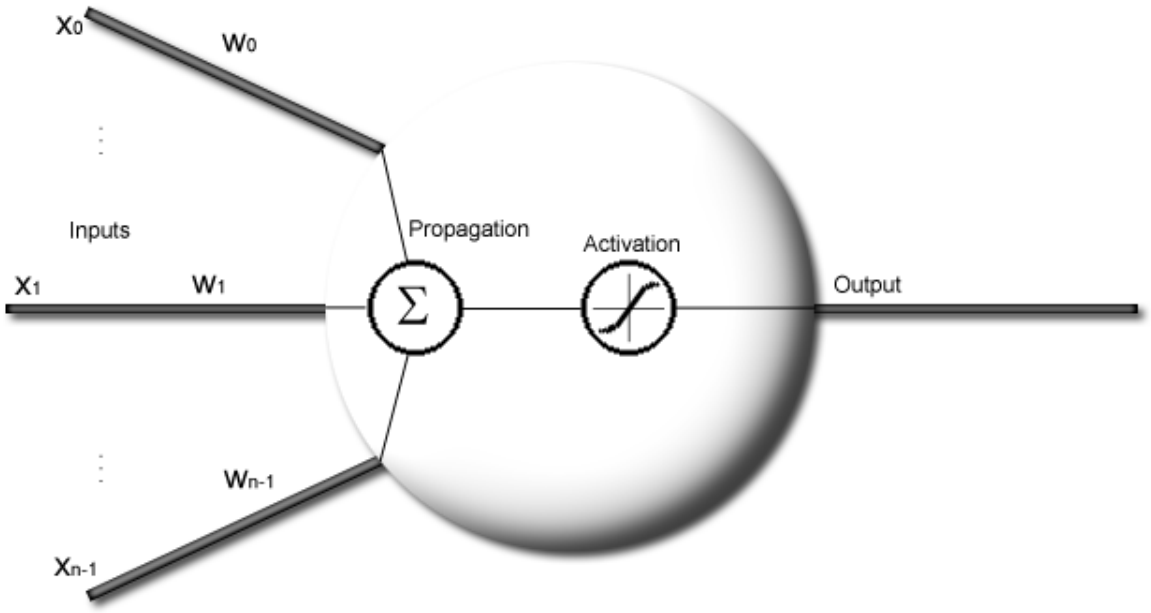


Figure 1: Anatomy of a Neuron

associated weights $[w_0,\ w_1,\ \ldots\ w_{n-1}]$ which provide the propagation rule with input values. Typically the propagation rule returns the weighted sum of all input values.

So the propagation value z_j for node j becomes

$$z_j = \sum_{i=0}^{n-1} w_{ji} \cdot x_{ji}, \tag{1}$$

where x_{ji} is the value provided at the ith input to node j and w_{ji} is the associated weight. The resulting propagation value is then passed through the activation function which is sometimes linear, but more typically will be a sigmoid or hyperbolic tangent function. The resulting output value is usually "squashed" to a convenient range such as $[-1, 1]$ to prevent weight values from outgrowing the representation space. The output of the neuron may then be redirected through any number of links to other downstream nodes in the network. A more generalized and complete introduction to neural networks can be found in [18].

Topologies

There are countless possible configurations of interconnections between the nodes of a NN. Such configurations are referred to as *topologies*. When arranged to form a network, nodes are assigned specific roles and vary slightly in their operation depending on the role they fulfill in the network. *Input nodes* have no incoming links, no propagation rule, and no activation function. Each simply passes one of the externally provided values of the network input vector on to other nodes in the network through their outgoing links. *Output nodes* have no outgoing links. Each of their activation levels are passed out of the NN as part of the network output vector. *Bias nodes*, like input nodes, have no incoming links, propagation rule, or activation function. They simply provide a constant value to other nodes in the network through their outgoing links. The provided value is usually 1.0 by convention. *Hidden nodes* are neither inputs nor outputs of the network and exhibit all the processing functionality of the neuron in Figure 1.

One common topology, which is used extensively in this work, is a *multi-layer neural network*. The multi-layer NN is composed of multiple layers of nodes, as shown in Figure 2. Each layer consists of a set of nodes which are connected to all

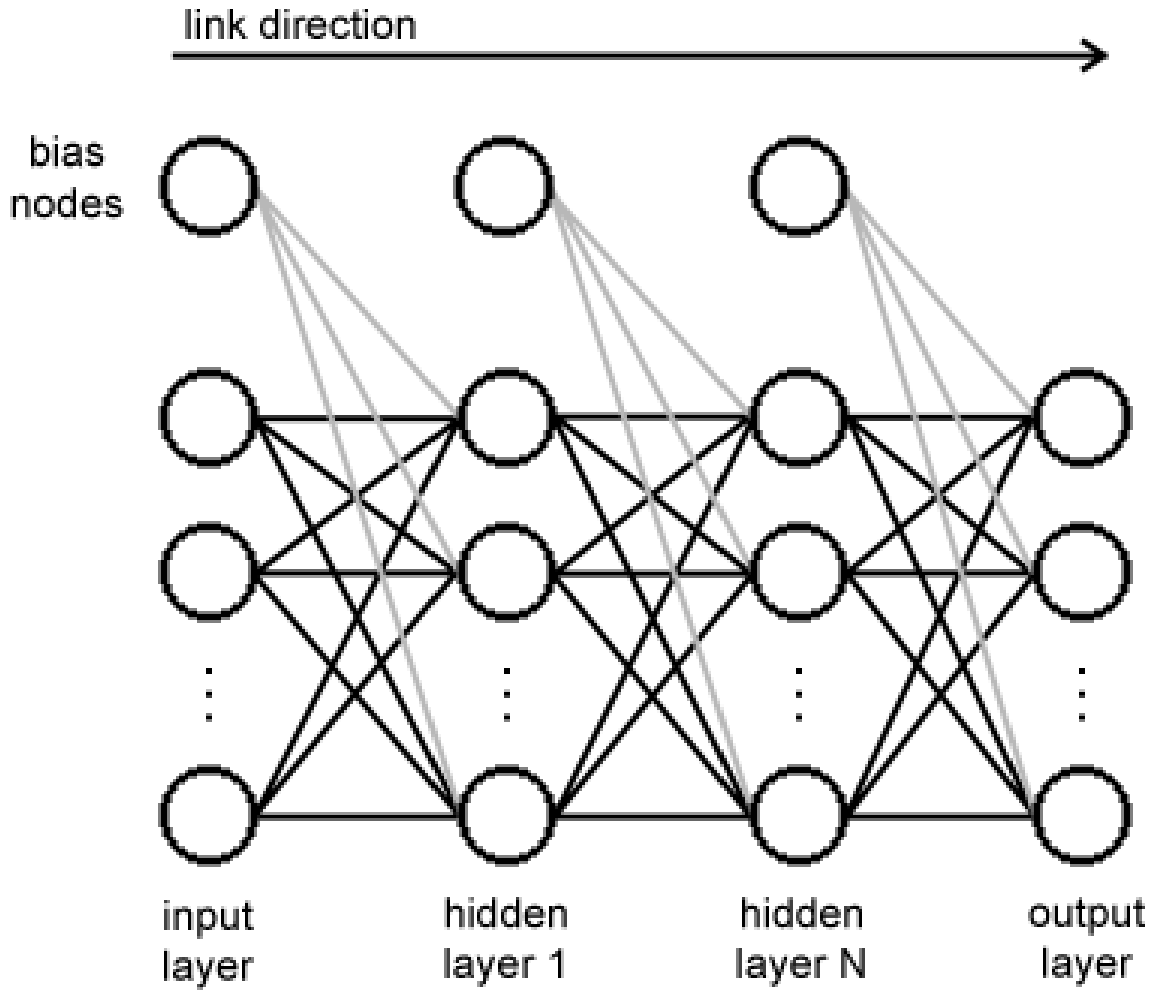


Figure 2: A Feed-forward, Multi-layer Neural Network

the nodes in adjacent layers. There is always exactly one input layer which is made up entirely of input nodes, the number of which should match the size of the network's input vector. There is always exactly one output layer which is composed entirely of output nodes, the number of which should match the size of the network's output vector. The multi-layer NN may also include zero or more hidden layers which are

made up of hidden nodes, the number of which may be arbitrary and vary among layers. Typically a bias node will be connected to each node in the hidden and output layers. Sometimes there is one bias node per layer for convenience of implementation. Either arrangement is equivalent.

In a *feed-forward* network like the one pictured in Figure 2, all links are directed downstream. In the *fully-connected* variant, nodes connect not only to the adjacent downstream layer, but also to all nodes in all downstream layers. An example fully-connected, feed-forward, multi-layer NN is pictured in Figure 3.

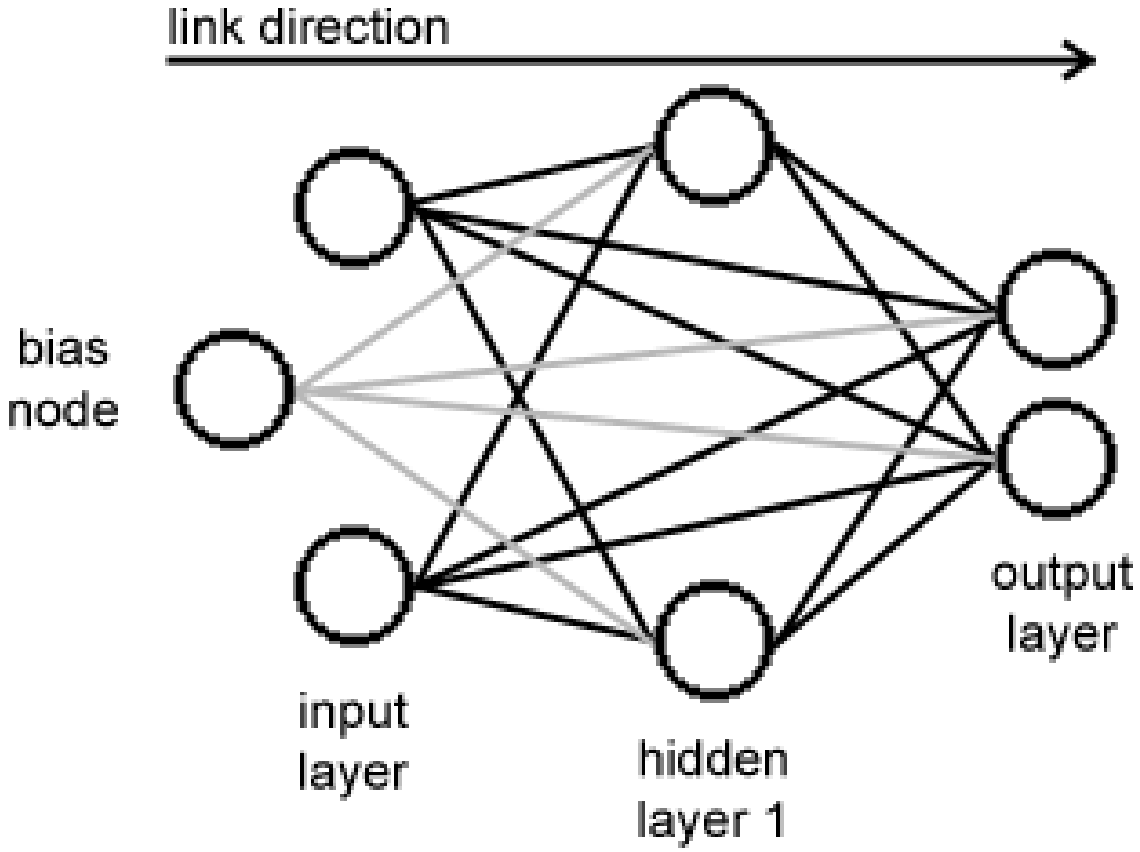


Figure 3: An Example of a Fully-connected, Feed-forward, Multi-layer Neural Network

Learning Algorithms

In order for a neural network to perform useful computation, it generally needs to be trained. Training consists of setting the link weights to values appropriate to the desired computation. This is often done through an iterative process known as a *learning algorithm*.

There are numerous learning algorithms for neural networks, often they are tied closely to the network structure. Common algorithms for the types of networks used in this work include backpropagation and evolution.

Backpropagation

Backpropagation is a supervised learning method introduced by Paul Werbos in his 1974 Harvard doctoral thesis[19]. This method is particularly simple to implement for feed-forward neural networks but can be adapted to handle recurrent structures as well[20].

Backpropagation works by computing the effective error at each node in the network given a set of test input vectors $[\vec{i_0},\ \vec{i_1},\ \ldots\ \vec{i_n}]$ with corresponding target output vectors $[\vec{t_0},\ \vec{t_1},\ \ldots\ \vec{t_n}]$. The weights on each node's incoming links can then be adjusted proportionally to reduce the error. Over the course of many iterations, the error can be minimized, resulting in correct outputs, even for inputs not included in the test set. Making the simplifying assumption that the test set consists of only one input vector with its corresponding output vector $\vec{o}$, the error $E(\vec{w})$ as a function of the network weights $\vec{w}$ can be specified as

$$E(\vec{w}) = \frac{1}{2}\sum_{k\epsilon\vec{o}}(t_k - o_k)^2, \tag{2}$$

where $\vec{t}$ is the target output specified by the test set. The error term δ_j for node j can then be defined as

$$\delta_j = \frac{\partial E(\vec{w})}{\partial z_j}. \tag{3}$$

Through application of the chain rule and substitution, the gradient of the error function with respect to the weight w_{ji} corresponding to the ith input of node j becomes

$$\begin{aligned} \frac{\partial E(\vec{w})}{\partial w_{ji}} &= \frac{\partial E(\vec{w})}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ji}} \\ &= \frac{\partial E(\vec{w})}{\partial z_j} x_{ji} \\ &= \delta_j x_{ji}, \end{aligned} \tag{4}$$

where z_j is the propagation value for node j given by Equation 1 from the "Artificial Neural Networks" section, and x_{ji} is the ith input value to node j. Because the idea is to minimize the error using gradient descent, the inverse of the computed gradient is used in the weight update rule.

$$w_{ji} = w_{ji} + \eta \cdot \Delta w_{ji}, \tag{5}$$

where Δw_{ji} is given by

$$\Delta w_{ji} = -\delta_j x_{ji}, \tag{6}$$

and η is known as the *learning rate*. The learning rate specifies the step size along the error function surface which is traversed upon each update to the neural network weights. This value may remain constant throughout the training. However, gradually decreasing it as the training progresses is a common optimization to allow faster convergence on a minimum error while also allowing settling very near to that minimum.

For the case where j is an output node, δ_j can be computed using Equation 2 along with application of the chain rule

$$\begin{aligned} \delta_j &= \frac{\partial}{\partial z_j} \frac{1}{2}(t_j - o_j)^2 \\ &= -(t_j - o_j) \cdot \frac{\partial o_j}{\partial z_j}, \end{aligned} \tag{7}$$

which is effectively the negative difference between the target and the actual output, multiplied by the derivative of the activation function. When instead j is a hidden node, its effective error is determined by propagating the error backward through the network from the outputs

$$\begin{aligned} \delta_j &= \sum_{k \epsilon Downstream(j)} \frac{\partial E(\vec{w})}{\partial z_k} \cdot \frac{\partial z_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial z_j} \\ &= \sum_{k \epsilon Downstream(j)} \delta_k \cdot w_{kj} \cdot \frac{\partial o_j}{\partial z_j}, \end{aligned} \tag{8}$$

where $Downstream(j)$ is the set of nodes whose immediate inputs include the output of node j.

Though backpropagation is a common NN learning algorithm and would provide useful data for comparison, it is not used in this work due to time constraints and is left as a point for future research. Mention of this and descriptions of other ways backpropagation may be applicable to related research can be found in Chapter 5. More in-depth examination of the backpropagation algorithm and its theory can be found in [18] and [21].

Evolutionary Algorithms

Evolutionary algorithms (EAs) act on *populations*, or sets of prospective solutions. To evolve a neural network solution, each population member is typically a list of the weight values associated with each link in the network.

The initial population members are randomly generated with reasonable values for the problem space. Each iteration of the EA is called a *generation* because at the end of each one, all the populations have been replaced by their successors.

As written in Algorithm 1 every member of each population is first evaluated

```
DO-GENERATION(population)

for m ⇐ 0 to population.Size do
    population[m].Fitness ⇐ EVALUATE(population[m])
end for
newPopulation ⇐ new Population()
while newPopulation.Size < population.Size do
    member ⇐ SELECT-FROM(population)
    member ⇐ MUTATE(member)
    newPopulation.ADD(member)
end while
return newPopulation
```

Algorithm 1: EA Generation

to determine its fitness. The fitness function should be carefully chosen to ensure that higher fitness values correlate with better solutions to the problem being solved. A replacement population for the next generation is then generated by repeatedly selecting members from the existing population and mutating them. The selection must bias toward higher fitness members so that the evolution iteratively progresses toward better problem solutions. Mutation simply adjusts values in the member data structure. This is usually done stochastically. The magnitude of the adjustments may be inversely proportional to the member's fitness value such that members with higher fitness values are changed by smaller amounts, resulting in progressively finer adjustments as the evolution converges on a near optimal solution. This is referred to as *simulated annealing*[22].

Each generation is completed by replacing all the populations in this way. The EA continues running one generation after another until a stopping condition is reached. The stopping condition might be to end after completing a fixed number of generations, or to end when a particular fitness value is reached.

Numerous enhancing features may be added to the basic evolutionary algorithm construction to aid in faster or otherwise improved convergence[23]. When the top n most fit population members are preserved un-mutated between generations, those members are referred to as *elite members*. When the fitness function has no stochastic components, *elitism*[24] provides a monotonic increase in the peak fitness value from one generation to the next.

Genetic Algorithms

Though genetic algorithms (GAs) and EAs have developed separately from one another, their basic operation varies little. The most significant difference between them is the inclusion of a crossover operation in the GA during replacement population creation. The *crossover* operator combines features from two parent population members to produce either one or two (depending on the implementation) child members for the next generation. The children are then mutated as usual.

The addition of crossover is illustrated by Algorithm 2 which is otherwise the same as Algorithm 1. Instead of selecting a single member and mutating it, two members are selected, producing the child to be mutated and added to the replacement population.

A selection mechanism frequently used in GAs is *roulette selection*. Roulette selection is a popular selection algorithm used for GAs, wherein each member is allocated a non-zero fraction of the selection space proportional to its fitness value.

```
DO-GENERATION(population)

for m ⇐ 0 to population.Size do
    population[m].Fitness ⇐ EVALUATE(population[m])
end for
newPopulation ⇐ new Population()
while newPopulation.Size < population.Size do
    mother ⇐ SELECT-FROM(population)
    father ⇐ SELECT-FROM(population)
    child ⇐ CROSSOVER(mother, father)
    child ⇐ MUTATE(child)
    newPopulation.ADD(child)
end while
return newPopulation
```

Algorithm 2: GA Generation

Figure 4 illustrates this concept with a circular selection space where each member's allocated fraction is represented by the arc length for its associated sector. The selection point is then chosen at random, uniformly within the selection space. In the circular space shown in Figure 4 the selection point is simply a real valued angle x such that $0° \leq x < 180°$. The member associated with the fraction of selection space containing the selection point is returned as the selected member. The highest fitness individuals occupy more of the selection space and are therefore more likely to be chosen than low fitness individuals, yet selection of low fitness individuals remains possible.

The crossover operator combines values from both parent solutions and, like mutation, often includes a stochastic component. An example crossover operation for value lists (such as lists of NN weights) is shown in Figure 5. The values to the left of the crossover point are copied from the first parent and values to the right of the crossover point are copied from the second parent. This method of data combination from two parents is referred to as *single point crossover*. Mitchell,

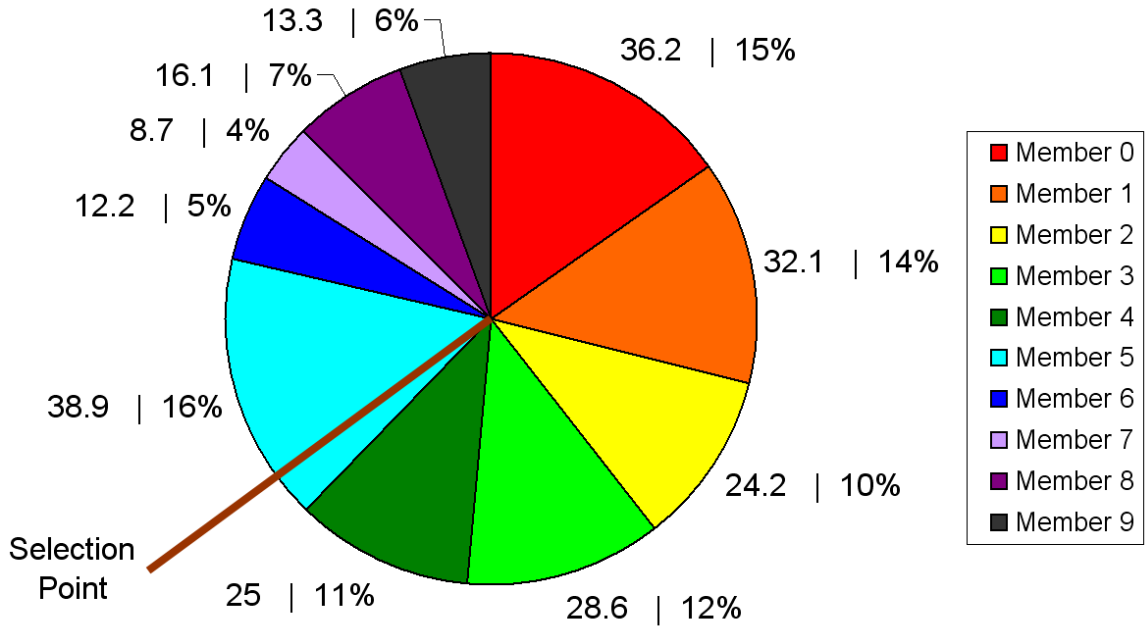


Figure 4: Roulette Selection Illustration with Population Size = 10

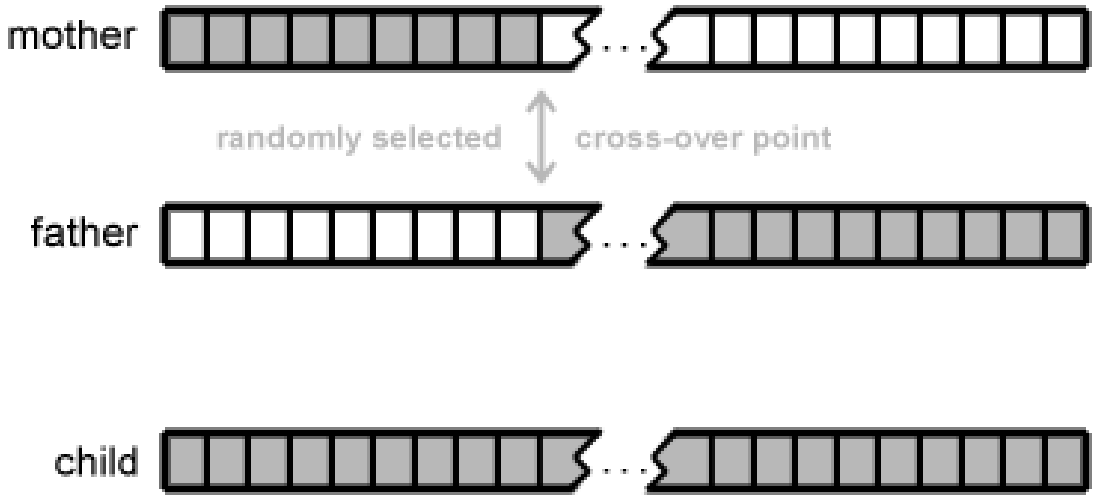


Figure 5: Single Point Crossover

Forest, and Holland[25] showed that GAs with crossover can converge upon high fitness solutions significantly faster than GAs without crossover.

GNARL

Angeline, Saunders, and Pollack developed an EA specifically for evolving neural network topologies. In the words of the authors:

> "GNARL, which stands for *GeNeralized Acquisition of Recurrent Links*, is an evolutionary algorithm that nonmonotonically constructs recurrent networks to solve a given task. The name GNARL reflects the types of networks that arise from a generalized network induction algorithm performing both structural and parametric learning. Instead of having uniform or symmetric topologies, the resulting networks have 'gnarled' interconnections of hidden units which more accurately reflect constraints inherent in the task."

One of the resultant networks that the authors are referring to is pictured in Figure 6. The unchanging inputs and outputs are shown in black, and the mutable hidden nodes

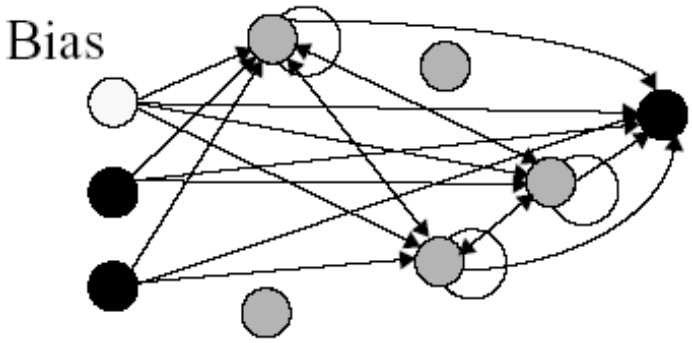


Figure 6: An Example GNARL Produced Neural Network

are grey. This pictured network is one of the results achieved in their research, at generation 765 of an evolution intended to solve the "Williams' Trigger Problem" discussed in [17].

Due to an interest in exploring evolved topologies as well as parametric evolution, the evolutionary approach to neural networks taken in this work was based upon GNARL[17].

The inputs and outputs of a GNARL evolved neural network are fixed by the problem being solved and cannot be changed by the algorithm. Links may be placed between any two nodes with a few restrictions:

R_1**:** There can be no links *to* an input node.

R_2**:** There can be no links *from* an output node.

R_3**:** Given two nodes x and y, there is at most one link from x to y.

Initialization

Initial networks are generated with a random number of hidden nodes and a random number of links. Both numbers are selected from a uniform distribution within supplied ranges. The incident nodes for all links are chosen randomly from possibilities that satisfy the restrictions given above. All link weights are assigned random values selected uniformly from the range $[-1, 1]$.

Selection

Networks scoring in the top 50% of the population by fitness survive to become parents of the next generation. The lower 50% are discarded. The replacement population for each generation is composed of mutated copies of the surviving members of the previous generation.

Mutation

Two types of mutation may be performed. *Parametric mutations* are adjustments to weight values. Parametric mutations do not alter network structure. *Structural mutations* are changes to the number or relative placement of nodes and links in the network. For both types, the severity of mutation for a given population member m is dictated by its *temperature* $T(m)$ as given by Equation 9:

$$T(m) = 1 - \frac{f(m)}{f_{max}}, \tag{9}$$

where f_{max} is the maximum fitness for the problem being solved. The mutation severity adjustment has the effect of simulated annealing on the evolution, allowing for a coarse grained search initially and proceeding to a finer-grained search as the algorithm converges on a solution.

GNARL also uses an *instantaneous temperature* $\hat{T}$ which helps avoid parametric local minima during the search:

$$\hat{T}(m) = U(0,1)T(m), \tag{10}$$

where $U(a,b)$ is a uniform random variable in the range $[a,b]$, and in this case $[0,1]$. In effect the instantaneous temperature allows for a low frequency occurrence of large mutations.

Parametric Mutation: Parametric mutations are performed by perturbing each value with Gaussian noise. For a link weight w the adjustment would be:

$$\Delta w = N(0, \alpha\hat{T}(m)), \tag{11}$$

where α is a user-defined proportionality constant, and $N(\mu, \sigma^2)$ is a Gaussian random variable with mean μ and variance σ^2.

Structural Mutation: Structural mutations alter the number of nodes and/or links in the network. All structural mutations strive to preserve neural network behavior. Accordingly, nodes are added without any initial links and new links are added with weights of zero. Such preservation is usually impossible when removing nodes and links. Removing a node involves the removal of all incident links as well, which can significantly affect network behavior. When a link is removed, the nodes it connects are not and may be left with no connections. Future mutations may remove the floating node, or reconnect it.

The number of additions or deletions Δ for both nodes and links is restricted to a user defined range $[\Delta_{min}, \Delta_{max}]$. The range is defined independently for each of the four structural mutation types. Selection of Δ from the available range for a given mutation type depends upon the individual's instantaneous temperature:

$$\Delta = \Delta_{min} + \lfloor U[0,1]\hat{T}(m)(\Delta_{max} - \Delta_{min}) \rfloor. \tag{12}$$

Selection of a node for removal is uniform across all nodes excepting inputs and outputs. Similarly, selection of a link for removal is uniform across all links in the network. Adding a link involves another parameter specifying the probability that each endpoint will be selected from the network input and output nodes instead of from the bias and hidden nodes. Every time the selection of a link endpoint is required, a node class selection is made according to the specified probability. A node is then chosen uniformly from the selected class to become the link's endpoint.

Fractional Order Calculus

Fractional order calculus has remained a topic of scholarly interest since the beginning of differential calculus. As is frequently noted, Leibniz and L'Hôpital

corresponded on the subject as early as 1695[26, 27]. Over the past 30 years it has received increased interest in its applications. Numerous recent applications in the field of control theory[14, 28] have been successful. Gorenflo and Mainardi[26] aptly describe it as "... the field of mathematical analysis which deals with the investigation and applications of integrals and derivatives of arbitrary order."

The definition of the differintegral upon which this work is based depends on the Gamma function as given in Equation 13

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t}\, dt, \tag{13}$$

which for this usage can be succinctly explained as an extension of the better known factorial function given in Equation 14, to real and complex numbers.

$$x! = \prod_{i=1}^{x} i \tag{14}$$

This definition of the differintegral is one first presented by Grünwald and later extended by Post. As taken from the text by Oldham and Spanier[27]:

$$\frac{d^q f}{[d(t-a)]^q} = \lim_{N\to\infty} \left\{ \frac{\left[\frac{t-a}{N}\right]^{-q}}{\Gamma(-q)} \sum_{k=0}^{N-1} \frac{\Gamma(k-q)}{\Gamma(k+1)} f\left(t - k\left[\frac{t-a}{N}\right]\right) \right\} \tag{15}$$

Collecting the gamma terms and performing the substitution given by Equation 16

$$\omega_k = \frac{\Gamma(k-q)}{\Gamma(-q)\Gamma(k+1)}, \tag{16}$$

yields the formula used in this work to implement the numerical evaluation of the q^{th} order differintegral of function f between the limits a and t for $t > a$.

$${}_aD_t^q f(t) = \lim_{N\to\infty} \left\{ \left[\frac{t-a}{N}\right]^{-q} \sum_{k=0}^{N-1} \omega_k \cdot f\left(t - k\left[\frac{t-a}{N}\right]\right) \right\}, \tag{17}$$

where ω_k is the k^{th} differintegral weight, N is the number of divisions evaluated between the limits and the size of the infinitesimal dt is accordingly

$$dt = \frac{t-a}{N}. \tag{18}$$

Positive orders of differintegration q correspond to derivatives, and negative values of q correspond to integrals. The case where $q = 1.0$ reduces to the well known definition of the first derivative of function f at point t given in Equation 19.

$$\frac{df(t)}{dt} = \lim_{h \to 0} \frac{f(t) - f(t-h)}{h} \tag{19}$$

Similarly, the case where $q = -1.0$ reduces to a familiar Riemann integral based definition of the first integral of function f between the limits a and t for $t > a$ given in Equation 20

$$\int_a^t f(t) = \lim_{h \to 0} \; h \sum_{i=0}^{N-1} f(t - ih), \tag{20}$$

where $N = \frac{t-a}{h}$.

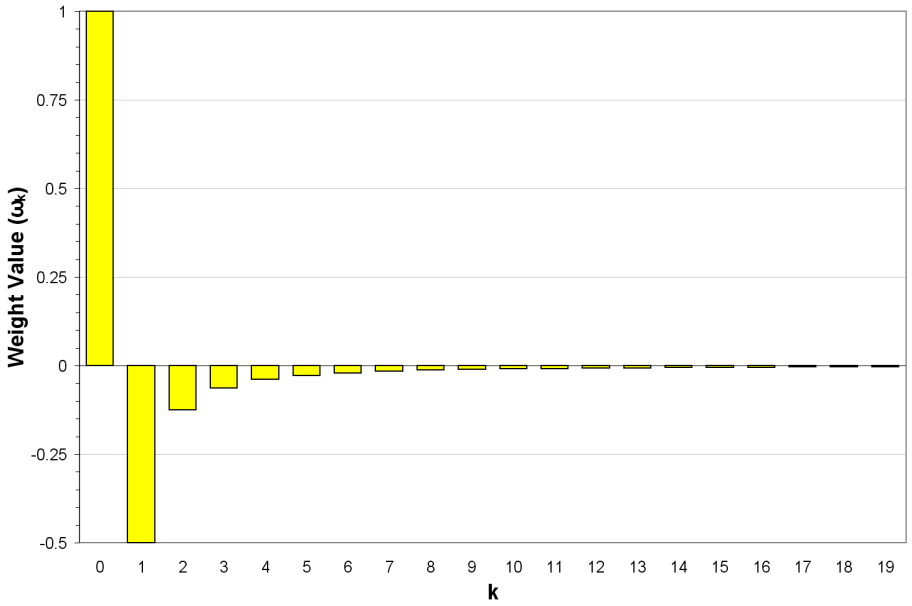
It is interesting to consider the differintegral weight values ω_k for specific values of q. A simple recursive relationship for computing them can be derived from Equation 16 using the properties of the gamma function given in Equation 21.

$$\Gamma(n+1) = n\Gamma(n) = n!, \text{ for } n \; \epsilon \; \mathbb{Z} \tag{21}$$

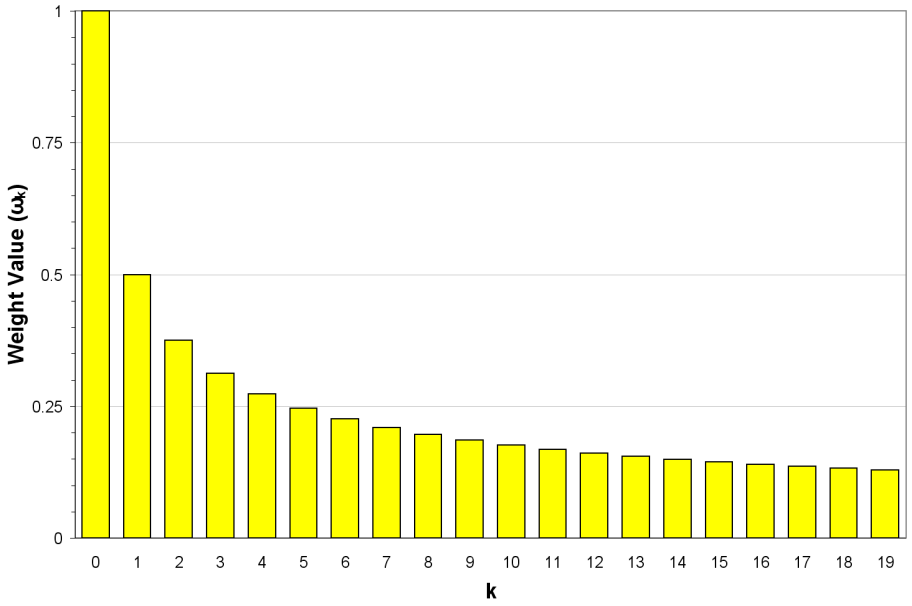
Using these properties, it can be shown that $\omega_0 = 1$ for all values of q. Substitution of $k = k + 1$ into Equation 16 also reveals that

$$\omega_k = \omega_{k-1} \cdot \frac{k - 1 - q}{k}. \tag{22}$$

When $q = 1.0$, $\omega_0 = 1$ and $\omega_1 = -1$. All other weights are zero in this case. This agrees with the simple difference operation applied in Equation 19. Similarly, when $q = -1.0$, ω_1 and all other weights are equal to 1. This agrees with the simple summing behavior demonstrated in Equation 20. Results are similar with other integer values of q. When instead q is a fractional value, the differintegral weights take on values like the ones shown in Figure 7.



(a) $q = 0.5$



(b) $q = -0.5$

Figure 7: Fractional Order Differintegral Weight Values

CHAPTER 3

SYSTEM DESCRIPTION

All experimental results for this project were obtained using a custom software solution developed specifically for this work. The software itself is described in detail in Appendix A, along with images of the graphical user interface (GUI) in operation. Selections from the source code are available in Appendix B. This chapter describes the differences between the system's implementation and the background theory discussed in Chapter 2.

Neural Network Implementation

The neural network implementation used allowed for any number of nodes to be connected in virtually any configuration. Links to input nodes were ignored and any non-input node with no inputs was implicitly a bias node. Aside from the mentioned constraints, links could be made between any two nodes whether the result be forward, backward, repeated, symmetric, or otherwise. This implementation scheme resulted in maximum flexibility in running experiments, though very few experiments strayed from the fully-connected feed-forward topology.

Discrete Differintegral Computation

To apply the differintegral operation shown in Equation 17 to neural networks, a method based upon the *G1-algorithm* as described by Bohannan[11] was applied. The method makes use of a recursive multiplication-addition scheme for computing the weights of the differintegral while applying them to the input function. The

same recursion was used for weight computation in this work, and is described by Equation 22, from the "Fractional Order Calculus" section of Chapter 2. For this work, the weights were computed as needed and stored for reuse at each differintegral computation.

Algorithm 3 shows the method used to compute the weights, where *weights* is a dynamic array of differintegral weights ($\omega_0,\ \omega_1,\ \ldots\ \omega_n$) and q is the order of the differintegral to which the weights apply.

```
COMPUTE-NEXT-WEIGHT(weights, q)

k ⇐ weights.size()
if k = 0 then
    weights.add(1.0)
else
    weights.add(weights[k - 1] * (k - 1 - q) / k)
end if
return weights
```

Algorithm 3: Discrete Differintegral Weight Computation

The order q discrete differintegral for the function $f(t)$ over the range $[0, t]$ was computed using the convolution demonstrated by Algorithm 4, where f is an array

```
DIFFERINTEGRAL(f, weights, q, dt)

sum ⇐ 0.0
N ⇐ f.size()
for i ⇐ 0 to N - 1 do
    index ⇐ N - 1 - i
    sum ⇐ sum + f[index] * weights[i]
end for
return sum / pow(dt, q)
```

Algorithm 4: Differintegral

holding N samples of the function $f(t)$ taken at intervals of size dt, and *weights* is an array holding at least the first N weights as computed by Algorithm 3. Algorithm 4

makes the simplifying assumption that a mechanism is in place to ensure all necessary weights are computed and available. Such a mechanism was implemented in the software used for this work.

Due to the finite nature of computer memory an approximation may be required to make computation of the differintegral practical over lengthy simulations. The software implementation used in this work was capable of applying four different approximation methods. Each resulted in a restriction of the sample history to a fixed maximum length. The approximation method considered to most closely match the results using a perfect sample history was referred to as *ratio approximation*. This method worked as shown in Algorithm 5, where *value* is the next sample to be added

```
ADD-SAMPLE(value, f, weights, L)

if f.size() ≥ L then
    ratio ⇐ weights[L - 1] / weights[L - 2]
    discard ⇐ f[0]
    f.remove(0)
    f[0] ⇐ discard * ratio
end if
f.add(value)
return f
```

Algorithm 5: Ratio Approximation

to the sample history f, *weights* is the array containing at least the first L weights computed by Algorithm 3, and L is the maximum sample history length. The oldest sample $f[0]$ is scaled by the ratio of the smallest weights used in the differintegral. The result is then added to the next oldest sample $f[1]$. The oldest sample $f[0]$ is then discarded to make room for the new *value* at the end of the sample history.

Most of the experiments performed for this work were short enough to allow practical computation over the entire sample history and for all those experiments

presented in Chapter 4, no approximation was applied. Instead perfect memory of the sample space was maintained throughout each simulation.

Fractional Calculus Neuron Augmentation

The fractional augmentation applied in this work entailed adding a fractional differintegration unit between the propagation rule and the activation function, as illustrated by Figure 8. For input and bias nodes, the additional processing was

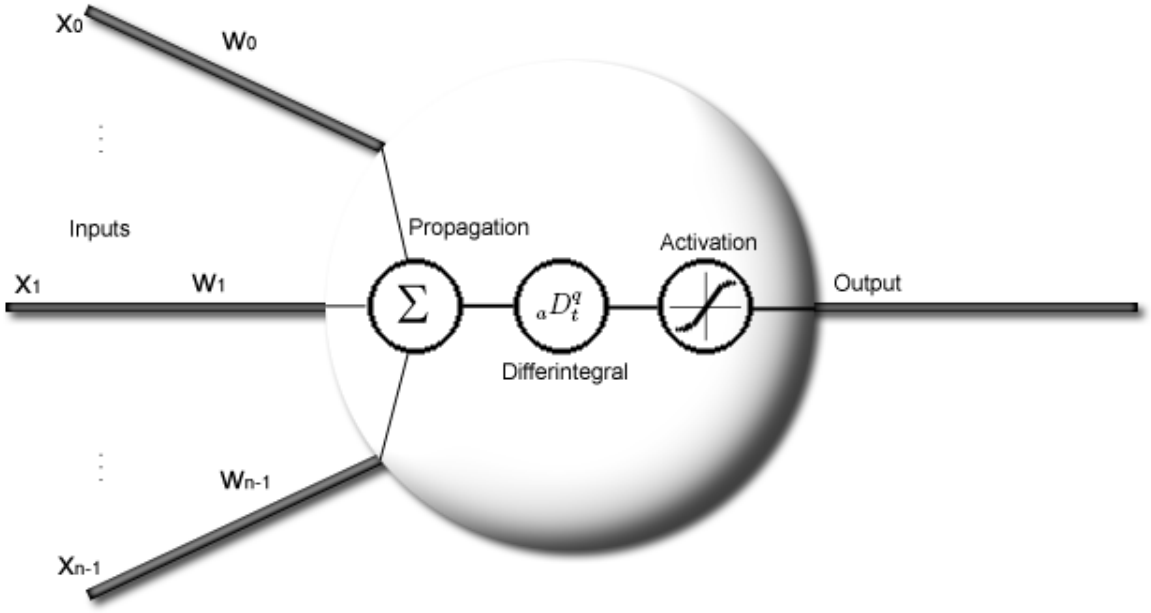


Figure 8: Fractional Calculus Neuron Augmentation

skipped along with the activation function. As in the un-augmented architecture, both node types still passed their values directly to the output.

The order of differintegration q was exposed to the learning algorithm for potential mutation. The other differintegration parameters such as dt and the history size limit were also exposed to the learning algorithm, but none of the experimental settings used in this work allowed their modification.

Learning Algorithm

Most of the experiments described in Chapter 4 were performed using fixed topologies. Learning was achieved by modification of network weights and q values. This was done primarily to simplify experiments. Some exploration using evolved topology types was desired however, so to accommodate such experiments a learning algorithm based on the GNARL algorithm described in Chapter 2 was adopted. Most of the changes made consisted of adapting rules of operation into adjustable options. The software was designed to function exactly as described by Angeline, Saunders and Pollack when configured with the appropriate settings.

The first two topological restrictions R_1–R_2 listed in the "GNARL" section of Chapter 2 remained unchanged. Violation of the third restriction by creating duplicate links was allowed. Further link restriction options were added to the software implementation, and are described in Table 1. With these additional

Option	Description
Disallow Self Links	Disable links connecting a node with itself.
Disallow Duplicate Links	Disable links with endpoints that match an existing link.
Disallow Mirror Links	Disable links with exactly opposite endpoints from an existing link.
Disallow Cross Links	Disable links with endpoints in the same layer.
Disallow Backward Links	Disable links connecting a node to one further from the output layer.

Table 1: Link Restriction Options

restrictions it was possible to constrain structural evolutions to produce only feed-forward topologies, among other variations.

Initialization

Initialization could be defined by the problem space, population member type, or population type, depending on preference. For the fox/rabbit problem used in this work, initialization options were provided by the neural network. Several predefined topologies were available, as well as the random topology used by GNARL. The weights could be initialized from either a uniform or a Gaussian distribution. The range of initial values was adjustable.

Selection

The software implementation allowed for the population survival rate between generations to be altered from the top 50% to any other valid quantity for a given population size. Any fraction of the population could be designated as elite, or *refreshed*. Refreshed members were freshly initialized members added to the population during each generation in place of mutated members from the previous generation. The software also implemented some GA-like selection mechanisms including roulette selection as described in the "Genetic Algorithms" section of Chapter 2.

Crossover

A crossover operation was also implemented to allow the possibility of configuring the software as a GA. This was a *multi-point* crossover algorithm generalized to allow crossover to occur between parents with varying network topologies. The operator essentially treated each network as a list of its mutable components. A parental selection was made for every component in the list to determine which parent the child's component would be inherited from. Each parent was assigned

equal probability in the selection. In effect this allowed for $[0, N-1]$ crossover points, where N was the length of the mutable component list.

To allow for varying topologies, the different component types were handled as sublists. The actual implementation is given in the "Fox/Rabbit Evolutionary Algorithm Code" section of Appendix B.

Mutation

An option was provided to disable the effect of the temperature value of Equation 10 from the "GNARL" section of Chapter 2. Instead, a fixed probability of mutation could be specified, making the evolution behave like a GA. The software implementation also added options to toggle mutation of all the available parameters and structural adjustments.

Parametric Mutation: The GNARL algorithm adjusted only network weights. The software implementation for this work added several other options to the list of mutable parameters. These are listed in Table 2. Node values were important

Parameter	Description
Weights	Network link weights.
Node Values	Network node values.
Activation Functions	Network node activation function types.
Activation Slopes	Network node activation function slopes or binary thresholds.
Activation Limits	Network node activation function upper and lower output limits.
q	Fractional unit orders of differintegration.
dt	Fractional unit sample rates.
History Size Limits	Fractional unit maximum sample history lengths.
Approximation Methods	Fractional unit sample history approximation methods.

Table 2: Mutable Parameter Types

because they fully specified the output of bias nodes, as well as providing an initial value for hidden nodes in recurrent networks to be used while unwrapping loops. An additional option was provided to force all bias nodes to a constant value. This was used to maintain bias values of 1.0 for the experiments in this work.

Although none of the experiments performed for this work allowed parametric mutation of the activation function parameters, options were provided to allow them to mutate. Activation function choices available were: *binary threshold*, *linear*, *sigmoid*, and *hyperbolic tangent*. Each of these functions imposed upper and lower limits on their output values whether they were attainable (as for binary threshold and linear units) or asymptotic (as for sigmoid and hyperbolic tangent units). The slope of the function between the limits was also adjustable where applicable. In the case of the binary threshold function, the slope parameter was used to specify the cutoff threshold determining which limit was output by the function.

The only parameter related to the fractional units that was allowed to mutate for the experiments described in Chapter 4 was q. Other mutable parameters included the sample rate dt and the maximum sample history length. The available sample history approximation functions used in limiting the sample history length were: *truncation*, *coopmans*, *exponential*, and *ratio*. Truncation took no pains to avoid ringing and other signal artifacts related to abruptly forgetting sample values. Coopmans followed the algorithm of the same name, in adding the lost values to the beginning of the sample history. Exponential used a fixed decay of 0.99 applied to lost values before adding them to the end of the sample history, which was nearly the same as the ratio approximation described in the "Discrete Differintegral Computation" section.

Structural Mutation: The GNARL algorithm allows for addition and removal of links during network evolution. The software implementation used in this work

added an ability to change the type of network nodes. The only types available for this work were un-augmented neurons, and fractionally augmented neurons. The mechanism for changing node types was identical to that for adding or removing nodes, with specified range $[\Delta_{min}, \Delta_{max}]$ for the number of changes Δ made during each mutation. The selection of Δ was made based on the instantaneous temperature, using Equation 12 from the "GNARL" section of Chapter 2.

Other changes to structural mutation included the additional link restriction options given in Table 1 and an option to add exactly one input link and one output link with each new node, ensuring that it would be connected to the rest of the network.

Statistic Collection

For each evolution run by the software implementation, basic fitness statistics were aggregated across all members in a population, across all populations in the EA, and then across all generations run by the EA. These provided information such as the maximum fitness in each population at the end of any given generation, or the maximum fitness achieved by any population throughout the entire evolution.

Problem specific statistics were also collected at each of these levels, and each problem could define additional levels for statistic collection. In the case of the artificial life simulation described in Chapter 4, these included things like distance between entities, and the speed of the fox at any given time step. The time-step-level statistics were aggregated across all time steps in the simulation before joining simulation-level statistics to be aggregated across all simulation trials for each member evaluation. These results were then joined with member level statistics to be further aggregated alongside the fitness values.

CHAPTER 4

EXPERIMENTS AND RESULTS

Several experiments were run utilizing an artificial life simulation to compare the un-augmented neural network (NN) to the fractionally augmented neural network (FNN).

Artificial Life Simulation

The simulation took place on a 2-dimensional (2D) bounded field. Two virtual creatures were placed within the bounds of the field along with optional obstacles, depending on the experiment. All entities in the simulation were circular and fully specified by a 2D position coordinate and a radius. Creatures were allowed to move by changing their acceleration vectors which indirectly affected their velocity vectors. They were not allowed to pass through the walls (field boundaries) or obstacles on the field. One of these creatures was designated as the "fox" and the other as the "rabbit". The rabbit's goal was to evade the fox until the simulation's end. The fox's goal was to capture the rabbit by touching or passing through it at some point during the simulation.

The rabbit's movement was dictated by simulation settings which varied by experiment. The movement of the fox was dictated by the outputs of a controlling neural network. The two outputs formed a 2D acceleration vector with coordinate values in the range $[-1, 1]$. These were scaled to the the range of the maximum acceleration magnitude and applied to the fox's movement at each simulation time step. The inputs to the neural network were generated by a virtual sensor array. Each

sensor in the array provided two values to the fox's neural network, one indicating the average entity type sensed and another indicating the average intensity (or closeness) of entities sensed.

The number of sensors was variable and dictated by simulation settings. As illustrated in Figure 9, the 360° around the fox was divided evenly into sensor sectors

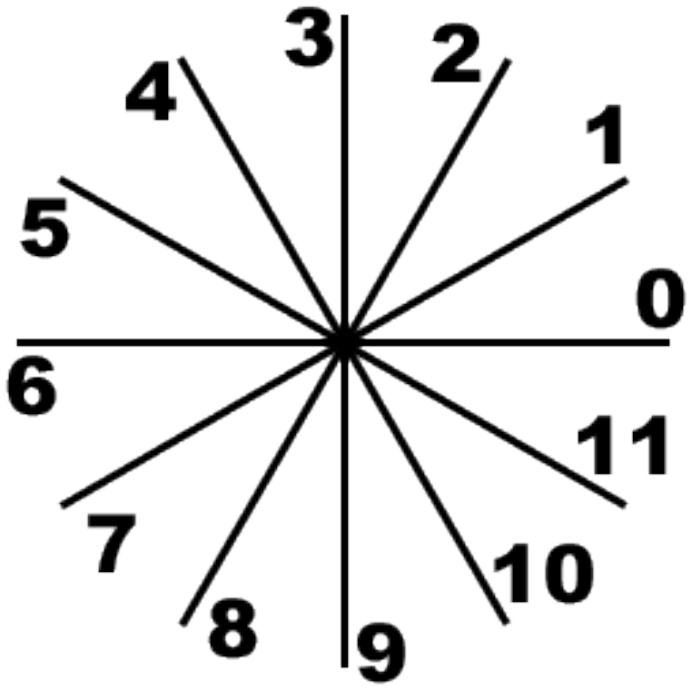


Figure 9: Fox Sensor Array

starting from 0°. The sensor array did not rotate with the fox's movement, so the boundary between the first and last sensors always pointed in the direction of the positive X-axis. Each sensor was further divided into a number of equal sub-sectors, or cells, specified by an alterable resolution setting. The real valued type and intensity for each sensor were generated by averaging the type and intensity across its cells. The type value returned by each cell was determined by the nearest entity in its sensed range and is given by Table 3. The intensity value returned by each cell was computed according to Equation 23,

Entity	Value
Wall or Obstacle	-1
None	0
Creature	1

Table 3: Sensor Cell Type Values

$$intensity = 1 - \frac{d}{d_{max}}, \tag{23}$$

where d is the distance to the nearest sensed entity and d_{max} is the maximum possible separation distance. The value of d_{max} depended on the dimensions of the field.

The sensor array was planned as described in an effort to provide a large quantity of sensory input data which included some ambiguity. Use of memory in the FNN was expected to be beneficial in resolving the ambiguous sensor states. The two layer sensor/cell scheme provided the added benefit that it required relatively few inputs to the neural network.

To measure the fox's performance and to provide neural network fitness values to the encapsulating evolution, each simulation returned a score. This was a single value which ideally indicated how close the fox came to accomplishing its goal of capturing the rabbit. Because of the stochastic nature of the simulation, the actual fitness values (in the scope of the evolution) were computed by averaging each fox's score using a given neural network over several simulation trials. Despite this technicality, the scoring function may be referred to as the *fitness function*. The fitness function could be chosen and modified by simulation settings.

Common Experimental Parameters

The software solution used to run each of the experiments in this work allowed for the configuration of a large variety of settings. To relieve any possible confusion,

the base set of parameters common among the experiments in this work are provided in this section. Variations on these parameters are described with the experiments they affect. Table 4 provides all common evolution related parameters while Table 5

Name	Value	Description
Population Size	32	Members per population.
Elite Members	2	Number of top members maintained as elite. (5%)
Generations	1000	Length of the evolution in generations.
Trials	10	Simulations run to determine the fitness of each member.
Runs	4	Copies of each experiment run for data collection.

Table 4: Base Experimental Evolution Parameters

Name	Value	Description
Field Size	100 x 100	Dimensions of the field.
Time steps	40	Maximum length of the simulation in time steps.
Radius	0.5	Radius of both creatures.
Max Speed	5	Maximum velocity magnitude allowed for either creature.
Max Accel	1	Maximum acceleration magnitude allowed for either creature.
Velocity Decay	10%	Decay rate of velocity magnitude per time step.
Sensors	12	Number of sensors in each creature's sensor array.
Resolution	30	Number of cells in each sensor.

Table 5: Base Experimental Simulation Parameters

provides all simulation related parameters. All experiments utilized the feed-forward, multi-layer topology unless otherwise noted. These networks were composed of an input layer, an output layer, and one hidden layer. A single bias node was connected to both the hidden layer and the output layer. The total number of nodes in each of the resulting networks was 37, and the total number of links was 272. Other common neural network related parameters are summarized in Table 6.

In addition to the settings listed in the tables, many of the experiments shared a common fitness function and allowed creature starting positions. The fitness function

Name	Value	Description
Bias Nodes	1	Number of bias nodes. (value = 1.0)
Input Nodes	24	Number of input nodes. (2 per sensor)
Hidden Nodes	10	Number of hidden nodes. (1 layer)
Output Nodes	2	Number of output nodes.
Activation Function	Linear	Activation function used by network nodes.
Activation Slope	1.0	Activation function slope.
Activation Max	1.0	Maximum activation function output value.
Activation Min	-1.0	Minimum activation function output value.
Initial Weights	N(0, 1.0^2)	Distribution of initial random weight values.
Initial Node Values	N(0, 1.0^2)	Distribution of initial random node values.
Initial Qs	N(0, 0.5^2)	Distribution of initial random FNN Q values.
History Limit	None	Maximum FNN sample history length.

Table 6: Base Experimental Neural Network Parameters

applied was referred to as *bidirectional-approach* and generated values in the range $[0.0, 100.0]$. Its value for a given simulation was 100.0 if the fox successfully captured the rabbit. Otherwise, its value was computed using Equation 24,

$$Score = 40 + 40 \cdot \frac{\sum_{s=0}^{s_{max}-1} Approach_s}{s_{max}} \tag{24}$$

where s_{max} is the maximum number of time steps in the simulation, and $Approach_s$ is given by Equation 25,

$$Approach_s = \begin{cases} 1, & \text{if } \dot{d} < 0; \\ 0, & \text{if } \dot{d} = 0; \\ -1, & \text{if } \dot{d} > 0. \end{cases} \tag{25}$$

such that $\dot{d}$ is the instantaneous derivative of the separating distance between the fox and rabbit with respect to simulation time.

The rabbit starting position, unless otherwise specified, was at the origin (0,0) which was located in the center of the field. The fox starting position was at any randomly chosen point on a circle of radius 49.5 and centered at the origin. This

arrangement was such that the fox (having radius = 0.5) would touch the field boundary if the starting angle happened to place it on either the X or Y-axis.

Baseline

The *baseline* experiment compared the NN to the FNN using the parameters detailed in the previous section. This provided a general idea how the performance of the un-augmented networks and the fractionally augmented networks compared in the context of one of the simplest experiments possible using the fox/rabbit artificial life simulation. This experiment also served as a baseline for comparison with other experiments.

Figure 10 compares plots of the NN and FNN's maximum fitness values averaged

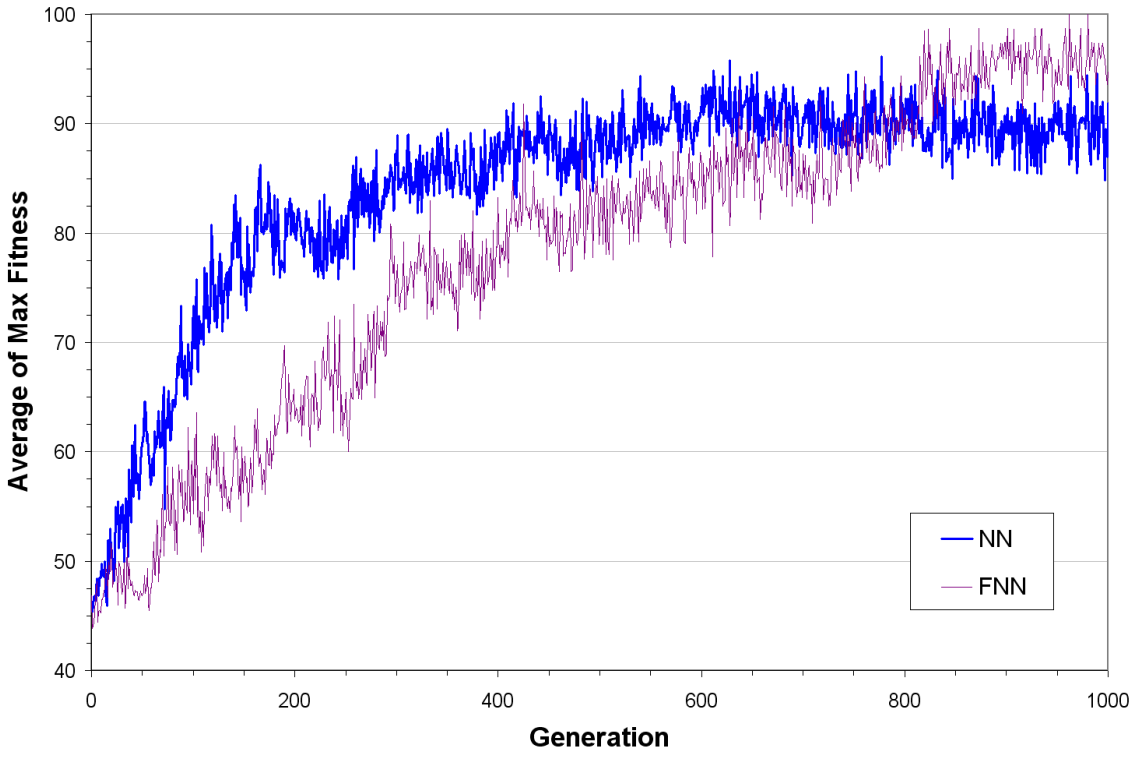


Figure 10: Baseline – Average Maximum Fitness vs Generation

across all four evolutions for each network type. From the plot, the NN's maximum fitness curve appears to rise quickly early in the evolution. It then levels off at about 600 generations. The FNN's maximum fitness looks more linear with some leveling in the last 100 generations.

Peak and final maximum fitness values for each network type are provided as 95% confidence intervals in Table 7. The *peak maximum fitness* is the highest maximum

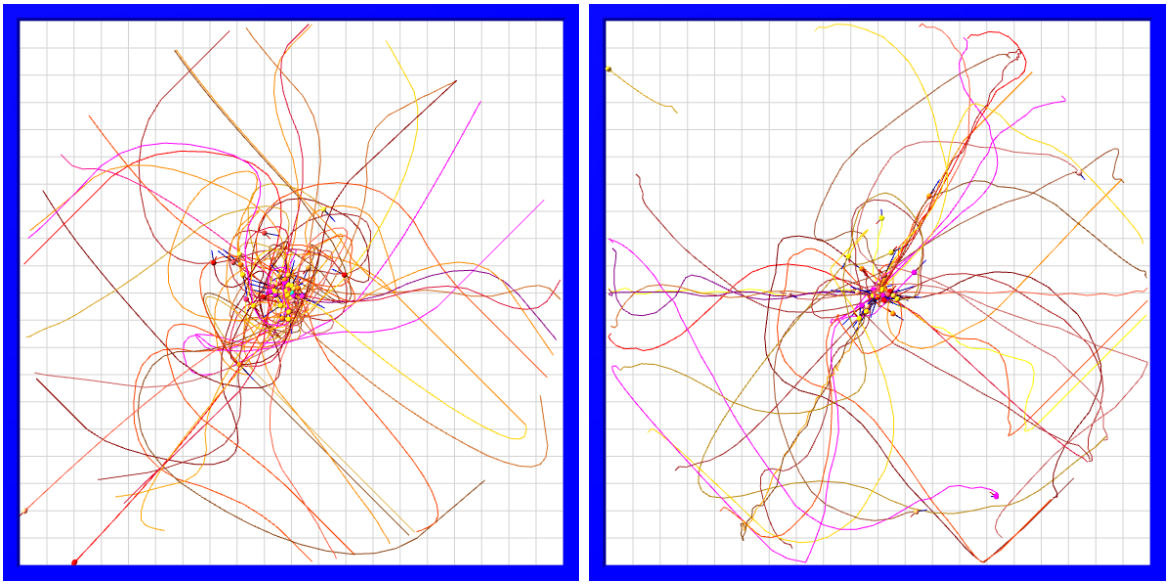
Type	Final Max Fitness	Peak Max Fitness
NN	91.9 ± 26.84	98.7 ± 7.35
FNN	94.8 ± 11.72	100.0 ± 0.00

Table 7: Baseline – 95% Confidence Intervals for Maximum Fitness

fitness value achieved at any point during the evolution. The *final maximum fitness* is the maximum fitness at the end of the evolution. It is often lower than the peak maximum fitness due to the stochastic component of the fitness evaluation. The confidence intervals given in Table 7 cover a wide range, indicating that no significant difference is likely between the results from the two network types. T-tests with $\alpha = 0.05$ confirm that the differences are statistically insignificant.

For each of the evolutions run, the highest fitness neural network was saved. This resulted in four peak fitness individuals from baseline NN evolutions and four from baseline FNN evolutions. Ten simulations were run for each of these individuals to observe their evolved behaviors. The path diagrams for the 40 resulting NN simulations are displayed in Figure 11a, and those for the 40 resulting FNN simulations are shown in Figure 11b.

The blue bars along the edges of the figure are the field boundaries. In the baseline experiment these stopped the creatures' movement but were not perceived. Grid lines are spaced at five unit intervals and the rabbit is represented by a stationary blue dot

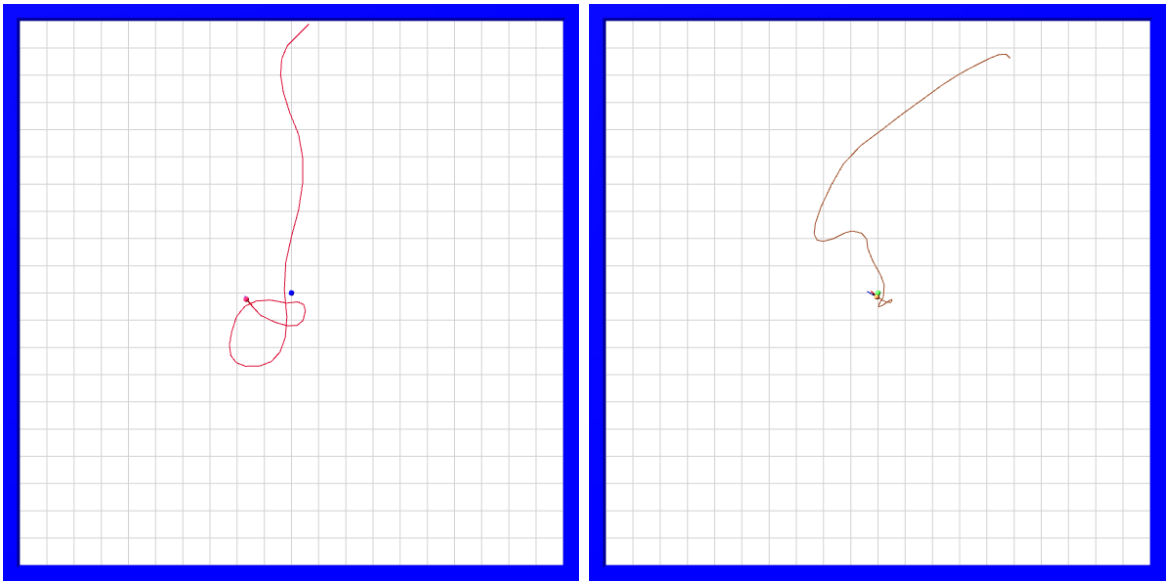


(a) NN (b) FNN

Figure 11: Baseline – Behavior Visualization

in the center of the field. The 40 foxes are represented by warm colored dots (reds, oranges, and yellows) with matching colored lines showing the paths they followed from their starting positions in a circle around the rabbit. All 40 simulations are overlaid in the same field, but no interaction occurred among them.

The area of high path density around the rabbit in Figure 11a is larger than that of Figure 11b. This is because the NNs exhibit relatively slow and broad, smooth arcing movement. The FNNs, on the other hand, tend to make more abrupt changes, though often arcing smoothly as well. This behavioral discrepancy is illustrated more clearly in Figure 12. The NN behavior pictured in Figure 12a is fairly typical in that, upon missing the rabbit in its first pass, the fox circles back in a large gentle arc to try again. In this case, it misses a second time, but the second loop is also tighter. If the simulation were allowed to continue beyond the 40 time step limit, the fox might have eventually caught the rabbit. In contrast, the FNN behavior displayed in Figure 12b shows a much tighter turn-around mechanism. Upon first missing the



(a) NN (b) FNN

Figure 12: Baseline – Selected Behavior Visualization

rabbit, the FNN fox makes a very small figure-eight before hitting its target on the second try. Examples can also be found for both network types where the fox very directly captures the rabbit on the first try. There are also a few instances of each type which meander about in undirected fashion or stop against a field boundary.

Capture rates and average final scores for these random simulations are shown in Table 8.

Statistic	NN	FNN
Capture %	42.50	62.50
Average Final Score	69.73	79.77

Table 8: Baseline – Simulation Data From Peak Fitness Individuals

Non-Resetting

In all other experiments, the foxes' controlling neural networks were reset between each simulation trial to clear any state information that may have been stored in the network. For this *non-resetting* experiment the networks were allowed to retain their states from one trial to the next during any given generation. The feed-forward topology prevents the NNs from holding state information, so only differences in the FNN were considered. A comparison between the non-resetting average maximum fitness values and those of the baseline experiment is shown in Figure 13. No significant differences are evident from the plot. Quantitative analysis

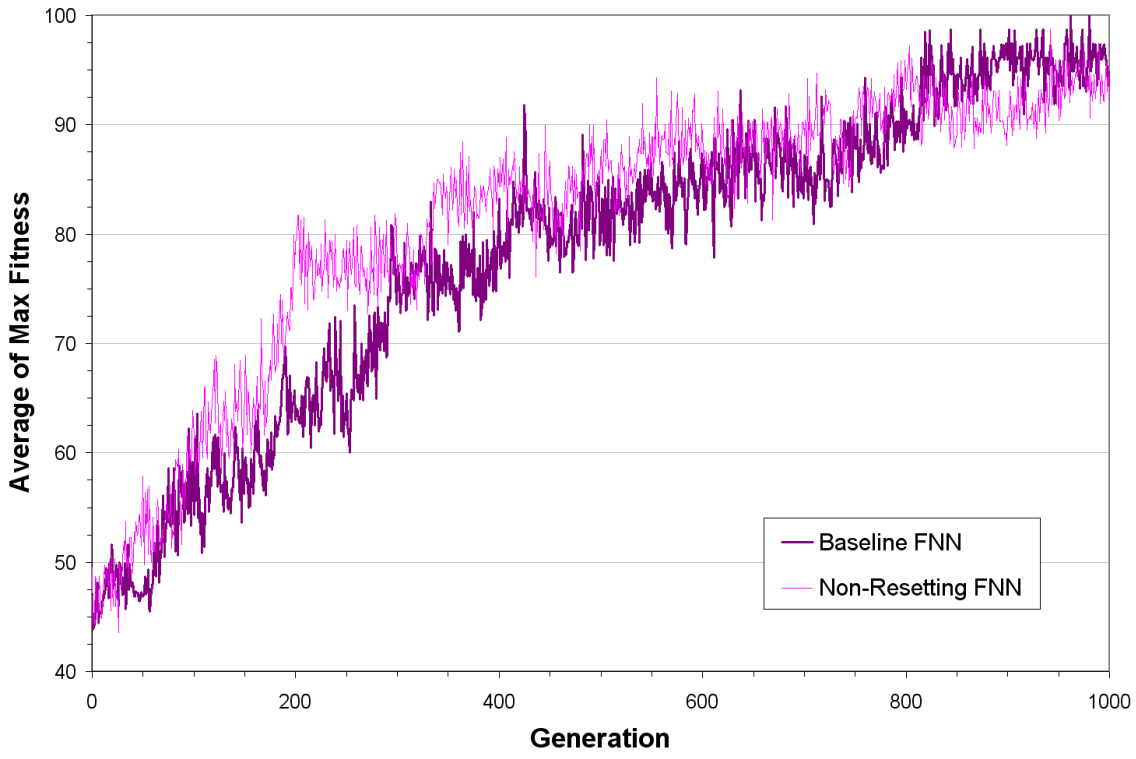


Figure 13: Non-Resetting – Average Maximum Fitness vs Generation

verifies this conclusion. A t-test with $\alpha = 0.05$ shows that the two plots are

statistically equivalent. Peak and final maximum fitness values for both the non-resetting experiment and the baseline are provided as 95% confidence intervals in Table 9, for comparison. The overlap between the confidence intervals is even more convincing that the two results are the same.

Experiment	Final Max Fitness	Peak Max Fitness
Non-Resetting FNN	92.2 ± 14.31	100.0 ± 0.00
Baseline FNN	94.8 ± 11.72	100.0 ± 0.00

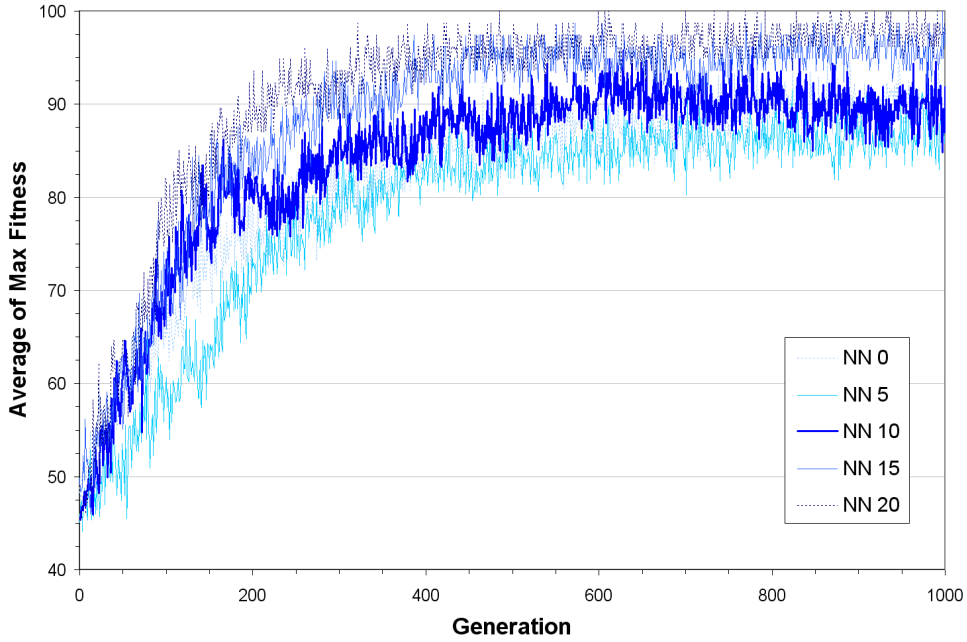
Table 9: Non-Resetting – 95% Confidence Intervals for Maximum Fitness

This finding is somewhat surprising because the discontinuity between trials was expected to leave the networks in inappropriate states at the beginning of each successive trial. Instead the FNN seems to be adaptable to such inconsistencies.

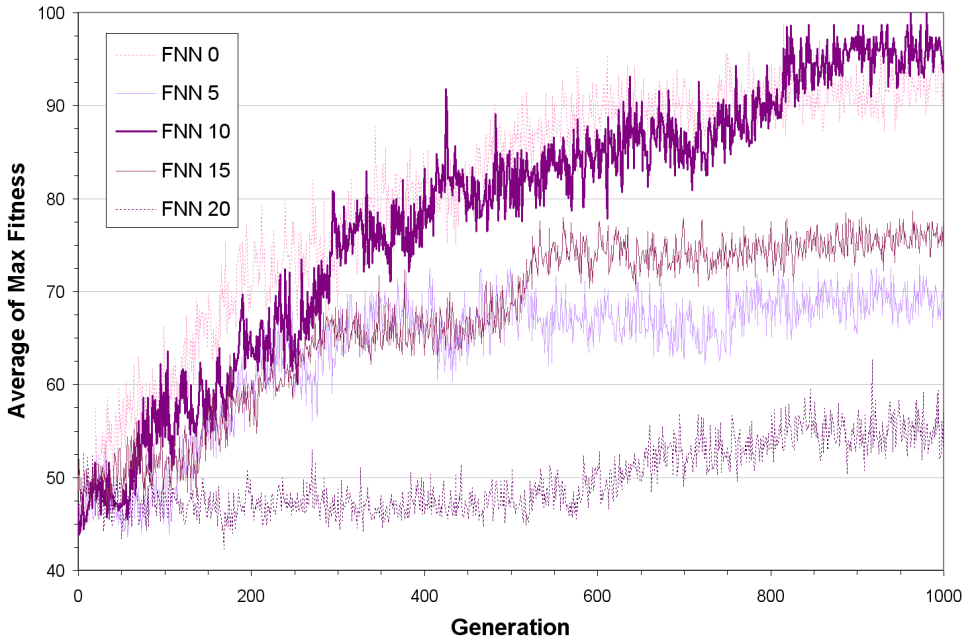
Hidden Layer Size Comparison

To determine if the size of the hidden layer affected the fitness results for either the NN or FNN, the baseline test was repeated several times with varying numbers of hidden nodes in the controlling networks. In addition to the 10 hidden node baseline experiment, data was collected for 0, 5, 15, and 20 node hidden layers for both the NN and FNN.

The average maximum fitness values plotted over the entire length of the evolution are shown in Figure 14. In visually examining the NN plots shown in Figure 14a, a relationship between the number of hidden nodes and the fitness data appears evident. With the exception of the zero node case, the plots for networks with higher numbers of hidden nodes appear higher on the fitness plot. This indicates that the NN produced higher fitness results with an increase in the number of nodes in the hidden layer. No direct relationship between the number of hidden nodes and the



(a) NN



(b) FNN

Figure 14: Hidden Layer Size – Average Maximum Fitness vs Generation

fitness data is apparent in the FNN plots displayed in Figure 14b. There does appear to be wider variation between the plots, than for the NN data.

Peak and final maximum fitness values for each NN experiment are provided as 95% confidence intervals in Table 10. The confidence intervals show a large proportion

Experiment	Final Max Fitness	Peak Max Fitness
NN 20	97.5 ± 8.03	100.0 ± 0.00
NN 15	96.2 ± 6.97	100.0 ± 0.00
NN 10	91.9 ± 26.84	98.7 ± 7.35
NN 5	85.7 ± 50.79	94.0 ± 33.12
NN 0	89.4 ± 12.89	98.5 ± 8.48

Table 10: Hidden Layer Size – NN 95% Confidence Intervals for Maximum Fitness

of overlap indicating that the differences between the curve values are statistically insignificant. The overlap is most likely due to the availability of only four evolutions for each experiment. The trend in the data observed visually is therefore unverifiable and may only be considered as a possibility.

Peak and final maximum fitness values for each FNN experiment are also provided as 95% confidence intervals in Table 11. Confidence intervals for the FNN data

Experiment	Final Max Fitness	Peak Max Fitness
FNN 20	57.2 ± 57.55	70.4 ± 46.05
FNN 15	77.6 ± 71.44	84.4 ± 50.13
FNN 10	94.8 ± 11.72	100.0 ± 0.00
FNN 5	69.9 ± 63.15	80.4 ± 62.76
FNN 0	94.2 ± 1.29	100.0 ± 0.00

Table 11: Hidden Layer Size – FNN 95% Confidence Intervals for Maximum Fitness

indicate high variability. The data available shows configurations with zero or ten nodes produced the best fitness results. The 95% confidence intervals for these configurations are also much narrower than the others, indicating that their values

are repeatable. Note that when no hidden nodes were present in the network, the fractional differintegration was still applied in the output nodes.

Fixed Q-Value Comparison

To determine whether some orders of differintegration provided better evolutionary performance than others, the FNN baseline experiment was run with several different Q values which were not allowed to mutate during the evolution. Data was collected for Q values of ± 0.25, ± 0.5, ± 0.75, and ± 1.0. The baseline NN experiment was effectively the same as running an FNN experiment with a fixed Q value of 0.0. In the case of $Q = 1.0$ the fractional processing units in each node were performing simple derivation of their inputs. In the case of $Q = -1.0$ the fractional processing units were performing simple accumulation of their inputs.

The experiments using negative Q values all produced plots that looked similar and provided little information to the observer. Most maintained a maximum fitness value near 45.0 throughout the evolution. The exception was for $Q = -0.25$ which achieved a final fitness value above 55.0. These negative Q value plots are not shown. The positive Q value average maximum fitness values are plotted in Figure 15, along with the baseline NN ($Q = 0.0$) and FNN (Variable) results for comparison. Large differences in fitness results produced by each fixed Q value experiment are visually apparent from the plot. The network with $Q = 0.25$ produced the best result, with the baseline variable Q data ranking second. The experiments with $Q = 0.0$ and $Q = 0.5$ also produced good results. Qualitatively the other experiments produced low fitness values and can be grouped with the negative Q value experiments as ineffective.

Peak and final maximum fitness values for each fixed Q value experiment are provided as 95% confidence intervals in Table 12, along with the baseline NN and

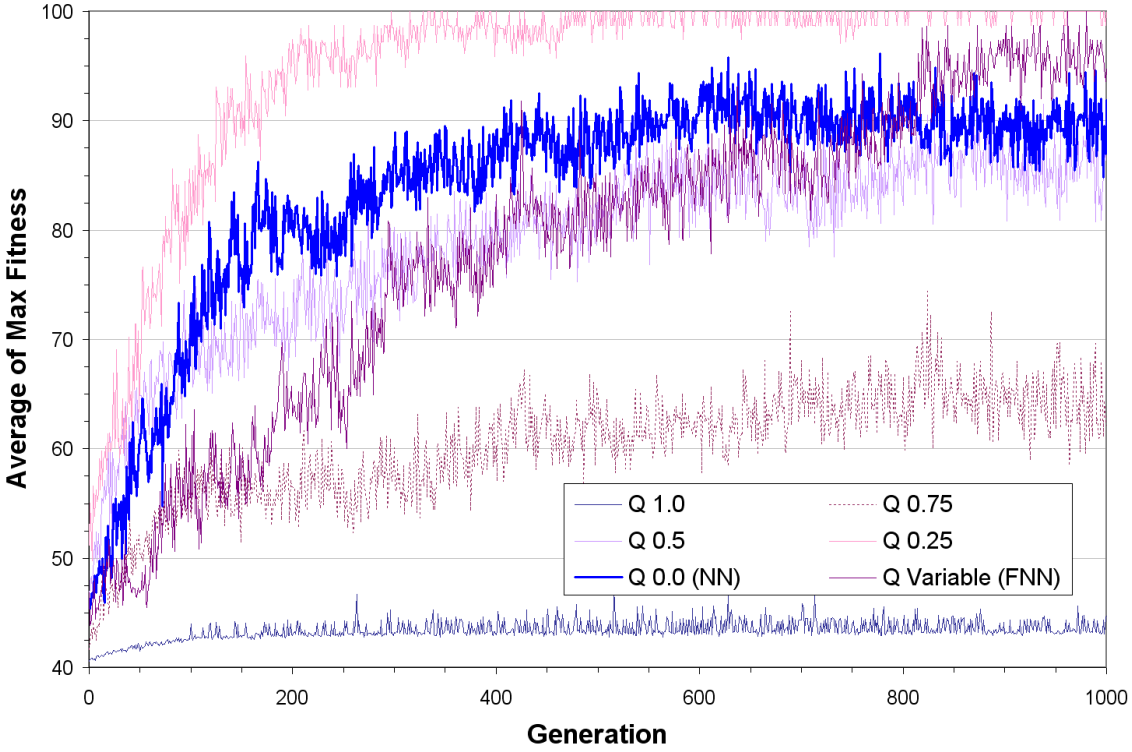


Figure 15: Fixed Q Value – Average Maximum Fitness vs Generation

Experiment	Final Max Fitness	Peak Max Fitness
Variable	94.8 ± 11.72	100.0 ± 0.00
Q=1.00	44.8 ± 5.97	49.4 ± 0.49
Q=0.75	62.1 ± 10.09	81.3 ± 7.48
Q=0.50	89.2 ± 21.16	98.5 ± 8.07
Q=0.25	100.0 ± 0.00	100.0 ± 0.00
Q=0.00	91.9 ± 26.84	98.7 ± 7.35
Q=-0.25	60.6 ± 33.15	70.4 ± 14.73
Q=-0.50	45.1 ± 15.90	60.1 ± 15.72
Q=-0.75	45.3 ± 7.42	55.5 ± 10.47
Q=-1.00	44.8 ± 6.23	58.5 ± 2.25

Table 12: Fixed Q Value – 95% Confidence Intervals for Maximum Fitness

FNN results for comparison. The confidence intervals indicate that the visual analysis is reasonably accurate. The $Q = 0.25$ experiment produced optimal fitness results with high confidence. The baseline NN and FNN results overlap considerably with the $Q = 0.5$ result. With data from only four evolutions for each experiment, these three are all statistically equivalent with 95% confidence. The remaining experiments can all be statistically categorized in a lower fitness group, though the $Q = 0.75$ experiment doesn't quite belong with the rest.

The results clearly indicate that integration-only FNNs produce lower fitness results than differentiation-only FNNs. FNNs composed of 1^{st} order derivative units produced similarly low fitness results. The overall impression given by the fixed Q value data is that positive Q values between 0.0 and 0.5 produce the best results for the baseline simulation. This generalization cannot be made however, without additional fixed Q value experiments. Populations evolved with variable Q values appeared to converge on solutions with Q values within the preferred range of $[0.0, 0.5]$, which would explain the baseline FNN results as compared with the fixed Q value results. Unfortunately the statistical data required to verify this observation was not available.

GA Evolution

The GNARL algorithm is not widely used to evolve neural network parameters. Evolving the network parameters using a typical genetic algorithm (GA) was important for establishing a basis for comparison. For this purpose, the baseline experiment was replicated with roulette selection, multi-point crossover, and a 1% chance of mutation for each new member. Temperature based mutation was disabled. All other settings were kept the same as in the GNARL based evolution. Node

values, network weights, and Q values were all initialized randomly from Gaussian distributions as before.

The average of the maximum fitness values for both the NN and FNN are plotted over the entire length of the evolution in Figure 16. The plotted results were similar

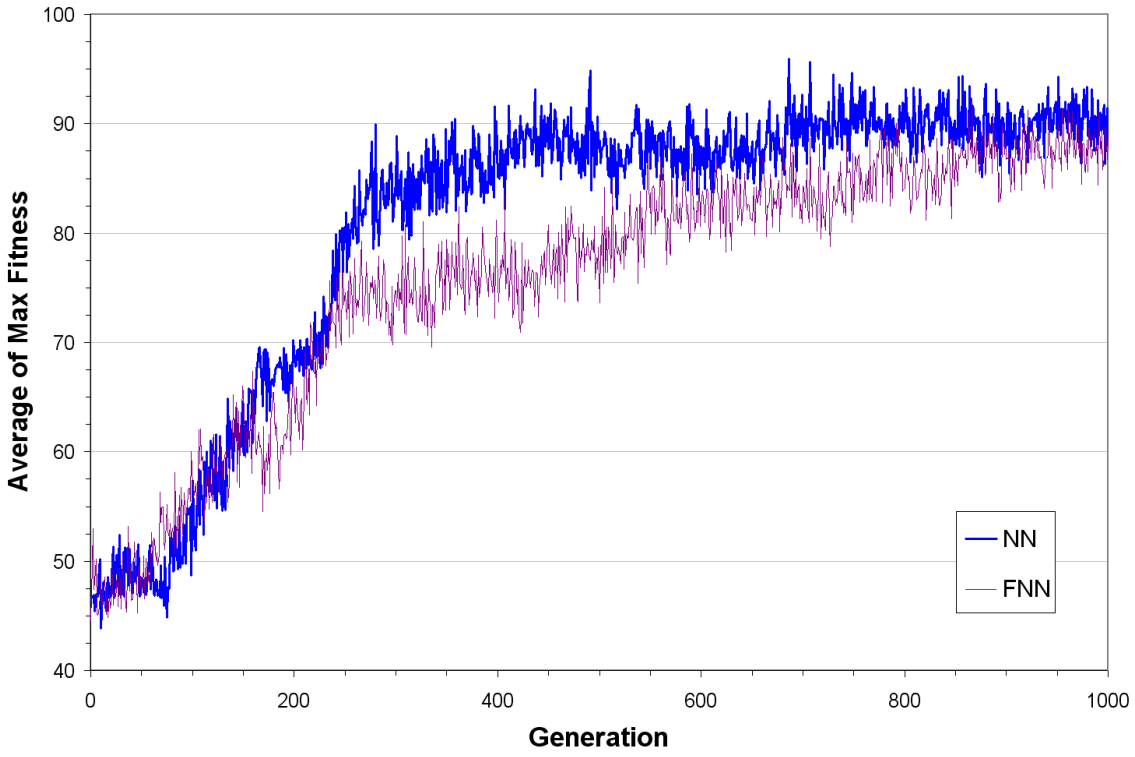


Figure 16: GA – Average Maximum Fitness vs Generation

to the baseline results given in Figure 10. T-tests with $\alpha = 0.05$ confirmed that neither the NN result or the FNN result varied significantly from its corresponding baseline experiment result. Another t-test showed with 95% confidence that no significant difference existed between the NN and FNN GA results. This matches the impression given by the plot in Figure 16. Peak and final maximum fitness values for both network types in the GA and baseline experiments are given as 95% confidence intervals in Table 14. It seems reasonable to conclude that evolution using

Experiment	Final Max Fitness	Peak Max Fitness
GA NN	87.9 ± 23.56	100.0 ± 0.00
GA FNN	88.5 ± 24.10	97.2 ± 15.42
Baseline NN	91.9 ± 26.84	98.7 ± 7.35
Baseline FNN	94.8 ± 11.72	100.0 ± 0.00

Table 13: GA – 95% Confidence Intervals for Maximum Fitness

GNARL produces results at least as good as those of a typical GA for experiments such as these.

Sensed Walls

To determine the effects of allowing the foxes to sense the boundaries of the field, an additional experiment was run with wall sensing enabled. With wall sensing enabled, each sensor cell registered a value of -1 when the field boundary was the nearest entity. Each cell also registered an intensity which resulted from linear interpolation between reference points along the field boundary. Eight points on the boundary were tracked for this purpose. These included the four corners of the field and the nearest point to the fox on each boundary. The intensity value for each of these points was computed from Equation 23, as described in the "Artificial Life Simulation" section.

The maximum fitness values averaged across all runs for each network type in the sensed wall experiments are shown in Figure 17. Qualitatively these results appear to have improved on the baseline results given in Figure 10. The plot indicates that the NN fitness values level off near 98.0 at around 600 generations, and the FNN fitness values climb to the maximum by about 500 generations. With data from only four experiments available, these differences are not statistically relevant. T-tests with

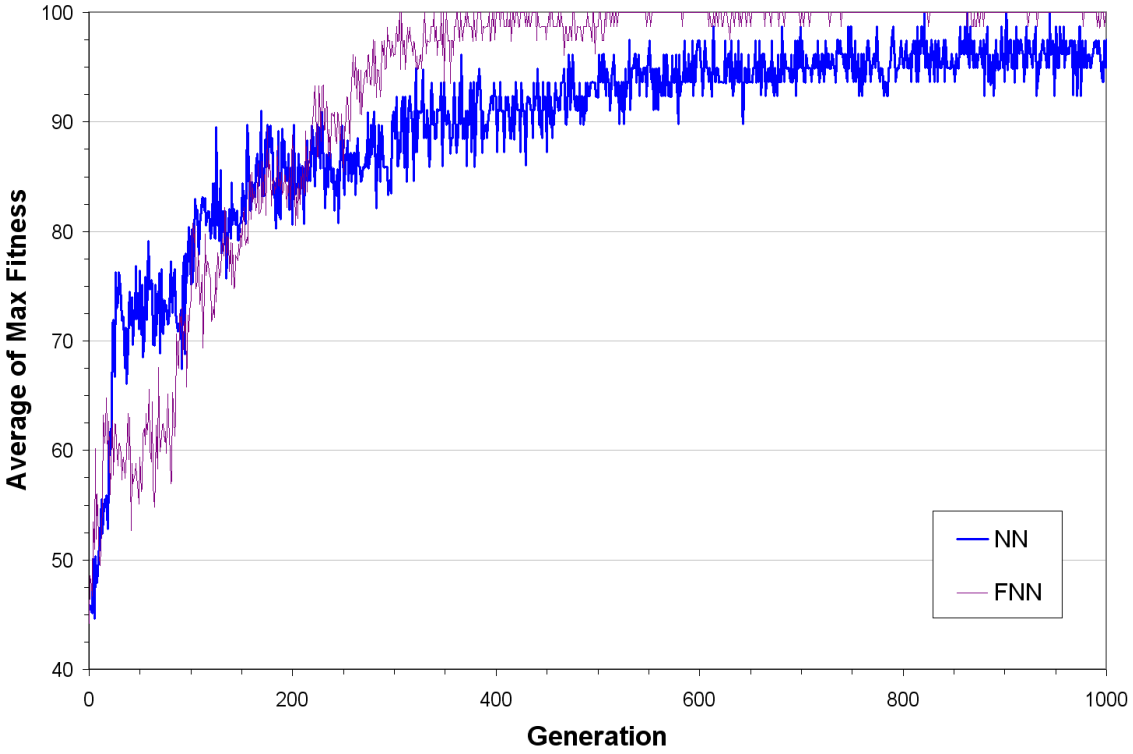


Figure 17: Sensed Walls – Average Maximum Fitness vs Generation

$\alpha = 0.05$ at each generation indicated only 16% of the data points varied significantly between the NN and FNN plots. T-tests comparing the NN and FNN results to their baseline experimental results indicated less than 3% of the data varied with 95% confidence.

Peak and final maximum fitness values for both network types in the wall sensing and baseline experiments are given as 95% confidence intervals in Table 14. The

Experiment	Final Max Fitness	Peak Max Fitness
Sensed Walls NN	97.5 ± 13.93	100.0 ± 0.00
Sensed Walls FNN	100.0 ± 0.00	100.0 ± 0.00
Baseline NN	91.9 ± 26.84	98.7 ± 7.35
Baseline FNN	94.8 ± 11.72	100.0 ± 0.00

Table 14: Sensed Walls – 95% Confidence Intervals for Maximum Fitness

confidence intervals for the two experiment types again indicate remarkable similarity. There is however, still room for speculation that more runs could result in statistically verifiable differences among them.

Track

In advancing the exploration of FNN capabilities, some additional simulation complexity beyond the simulation used in the baseline experiment was desired. The first variation along these lines was to allow the starting positions of both the fox and the rabbit to fall, at random, anywhere within the field boundaries. The resulting simulation is referred to as *track*.

The average maximum fitness value of the track NN and FNN experiments are shown in Figure 18. These results visually appeared to favor the NN slightly more than the baseline, while the FNN results looked unchanged. Statistically, there were essentially no differences between the NN or FNN and their baseline data. Neither were there differences between the NN and FNN curves in this experiment. The t-tests with $\alpha = 0.05$ indicated that only 1% of the data varied between the NN and FNN.

For completeness, peak and final maximum fitness values for both network types in the track and baseline experiments are given as 95% confidence intervals in Table 15.

Experiment	Final Max Fitness	Peak Max Fitness
Track NN	97.5 ± 8.04	100.0 ± 0.00
Track FNN	97.3 ± 8.77	98.9 ± 6.29
Baseline NN	91.9 ± 26.84	98.7 ± 7.35
Baseline FNN	94.8 ± 11.72	100.0 ± 0.00

Table 15: Track – 95% Confidence Intervals for Maximum Fitness

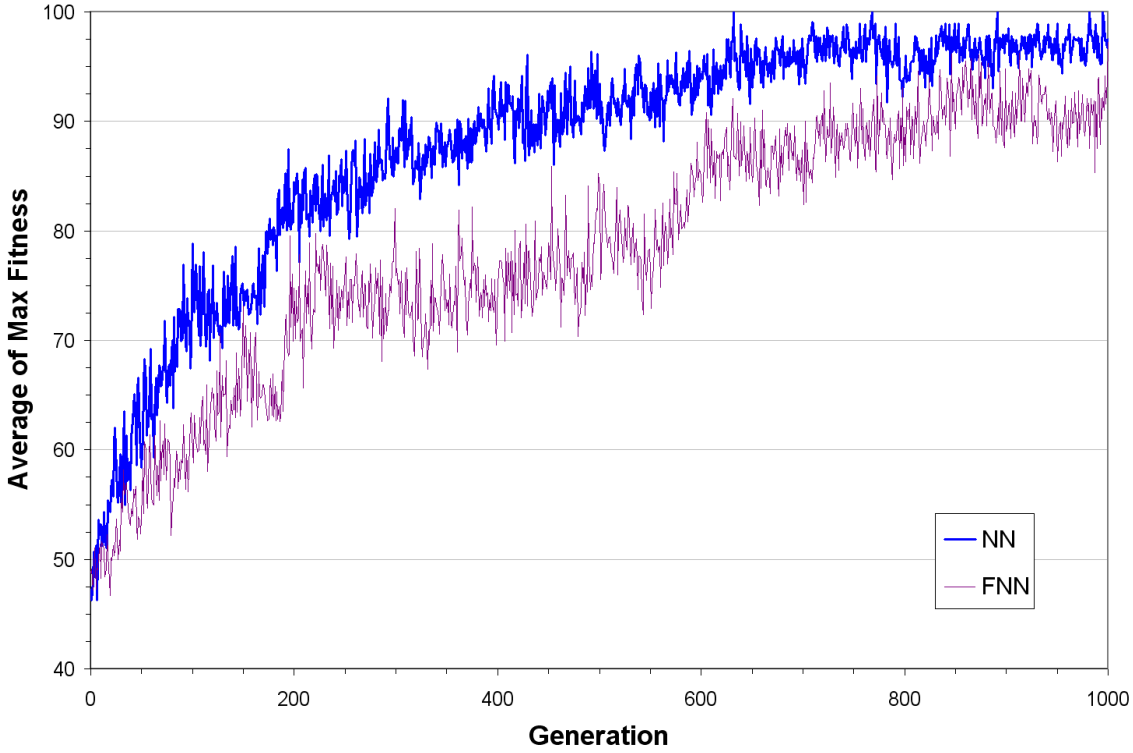


Figure 18: Track – Average Maximum Fitness vs Generation

All these findings indicate that the track experimental results match the baseline results. This is not surprising. The fundamental difference between the two experiments was that the distance between the fox and the rabbit was variable at the beginning of each simulation. That additional variable alone seems unlikely to account for any significant changes. Still the suspicion remains from plot comparison that the NN may perform slightly better under these conditions.

Intercept

The next variation on the baseline simulation was somewhat more dramatic. The field size was changed to 100 x 60. The allowed starting locations for the fox were anywhere on the horizontal line five units from the South end of the field. Similarly

the rabbit was allowed to start anywhere on the horizontal line five units from the North end of the field. The rabbit was allowed to move parallel to the X-axis (East and West) and its acceleration was determined stochastically at each time step. The rabbit's acceleration magnitude was limited to a maximum value of 0.2, along with a maximum speed limit of 1.0, to give the fox a sporting chance. Due to the resulting characteristics of the simulation, it is referred to as *intercept*.

The average maximum fitness values for the intercept NN and FNN experiments are shown in Figure 19. Similarly to the sensed walls experiment, the plots indicate

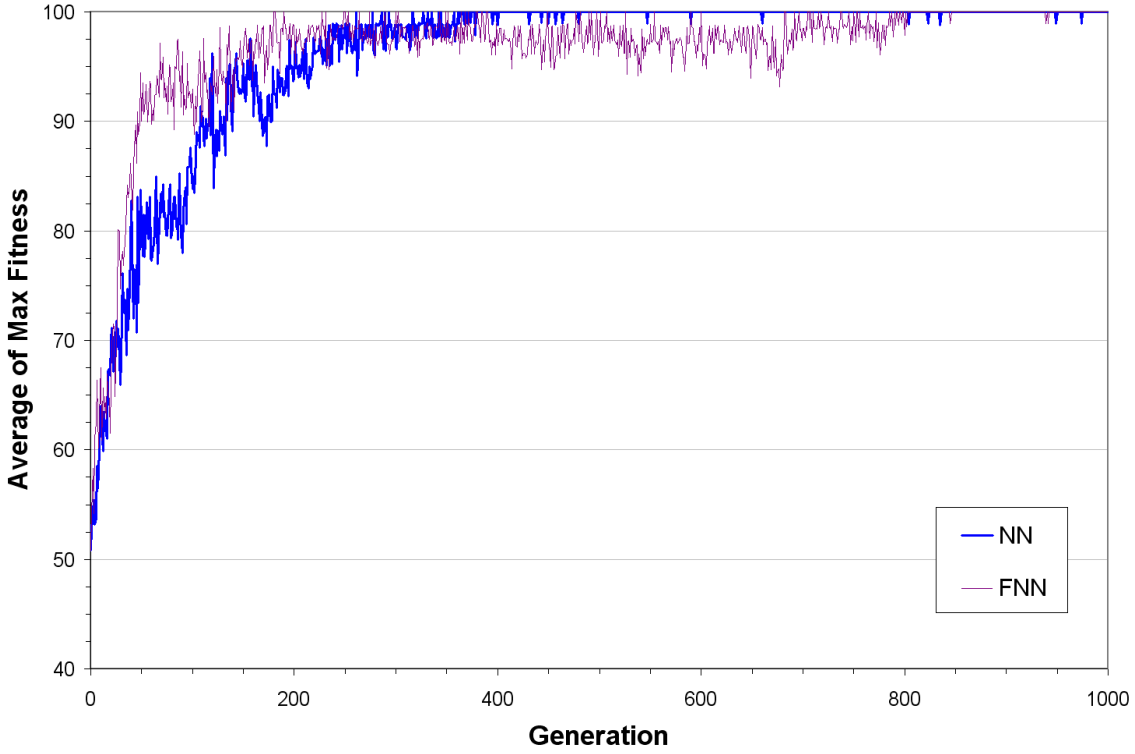


Figure 19: Intercept – Average Maximum Fitness vs Generation

an improvement in performance over the baseline experiment. In this case the NN fitness ramps up to the maximum within 400 generations. The FNN fitness levels

off near 200 generations, only rising to match the NN performance in the last 150 generations.

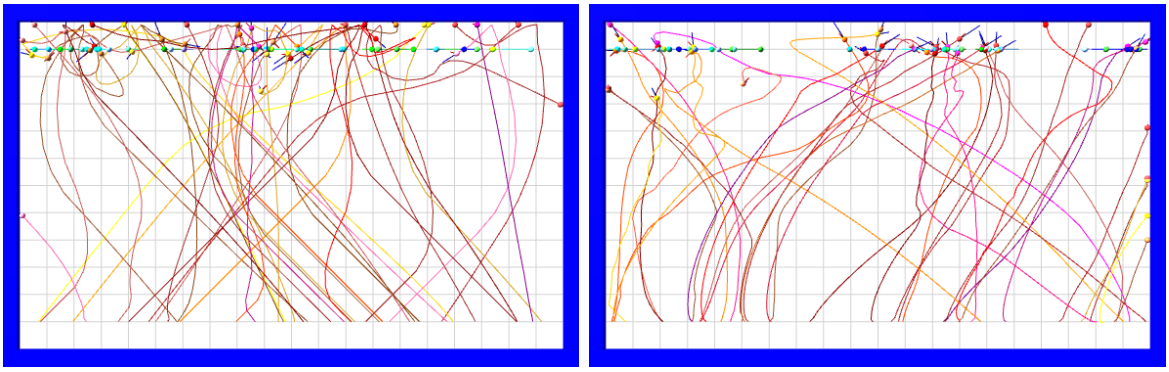
Peak and final maximum fitness values for both network types in the intercept and baseline experiments are given as 95% confidence intervals in Table 16, for direct

Experiment	Final Max Fitness	Peak Max Fitness
Intercept NN	100.0 ± 0.00	100.0 ± 0.00
Intercept FNN	100.0 ± 0.00	100.0 ± 0.00
Baseline NN	91.9 ± 26.84	98.7 ± 7.35
Baseline FNN	94.8 ± 11.72	100.0 ± 0.00

Table 16: Intercept – 95% Confidence Intervals for Maximum Fitness

comparison. Due to the limit of 100.0 imposed artificially on the fitness values, all the intercept results indicate 100% confidence in achieving the maximum. The t-tests using $\alpha = 0.05$ conclude however, that only 4% of the NN data was significantly different from the baseline case, and 18% of the FNN data varied significantly. Further t-tests indicate with 95% certainty that only 1% of the data varied between the NN and FNN for this experiment. Therefore, with only four experiments of each type to work with, it must be concluded that the intercept and baseline experimental results essentially do not differ. Allowing the rabbit to move intuitively suggests that it would become more difficult to capture, so it is somewhat surprising to see no change in the results.

Evolved behaviors from the intercept experiment are illustrated by the path diagrams in Figure 20. As with the baseline experiment, these were produced by running ten simulations for each of the four peak fitness individuals saved from evolutions of each network type. The resulting 40 NN simulations are pictured in Figure 20a and the resulting 40 FNN simulations are shown in Figure 20b. The rabbits are represented by cool colored dots moving horizontally along the tops of



(a) NN (b) FNN

Figure 20: Intercept – Behavior Visualization

the figures, and the foxes are represented by warm colored dots moving toward the rabbits from the bottom of the figures. The NN foxes seem to miss and bounce off the back wall, or circle the rabbit more than the FNN foxes. The latter seem to adjust their incoming trajectory frequently and more often capture the rabbit on their first pass. Again some instances of both network types fail to even approach their targets.

Hide and Seek

Yet another variation on the baseline simulation was to introduce obstacles. A single obstacle was added in the center of the field with a radius of 25.0, resulting in over 60% of the space on the field being inaccessible. The time step limit was increased to 80 to allow plenty of time for the foxes to navigate the obstacle. The rabbit was allowed to start anywhere on the field, just as in the track experiment, though the simulation logic prevented starting positions from falling inside the obstacle. The fox was allowed to start anywhere on a circle of radius 49.5 centered at the origin, just as in the baseline experiment. The rabbit was not allowed to move. The fox often could not perceive the rabbit from its starting location. In some sense then, the rabbit

was hiding, and the fox's task was to seek it out and capture it. This simulation is referred to as *hide and seek*.

Faced with these odds, it may come as no surprise that the hide and seek evolution was unsuccessful. The average maximum fitness values for both the NN and FNN remained relatively constant throughout the evolution as illustrated in Figure 21.

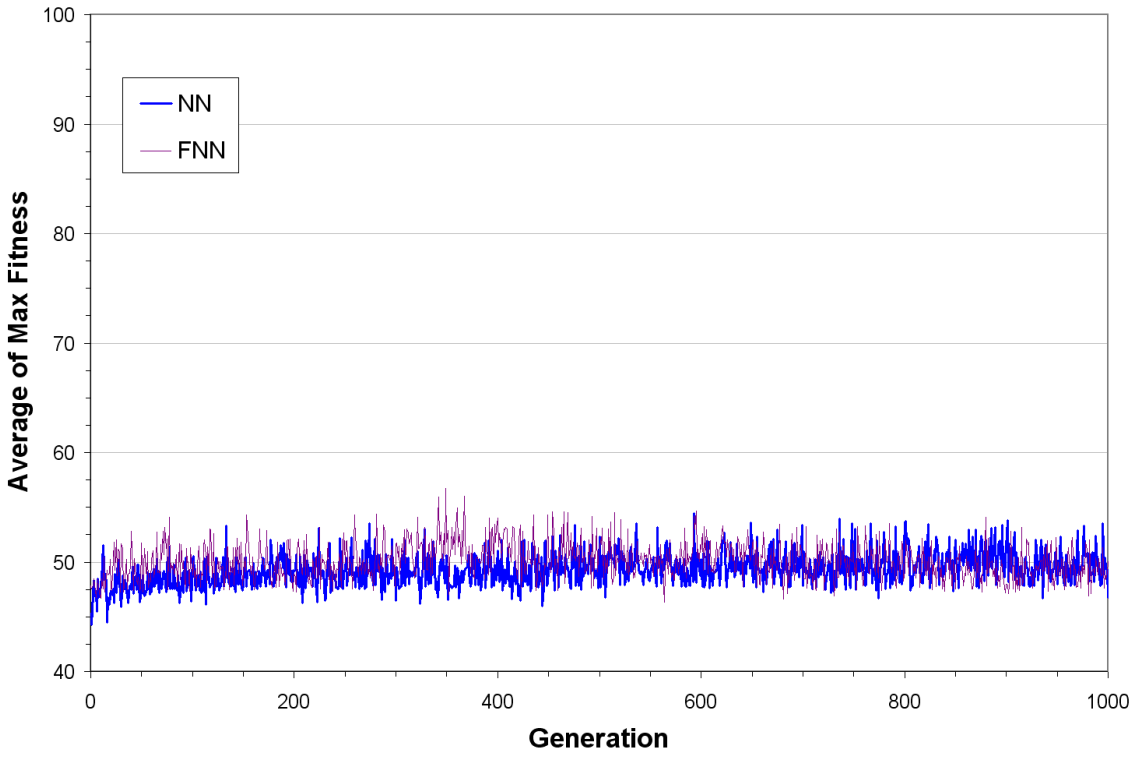
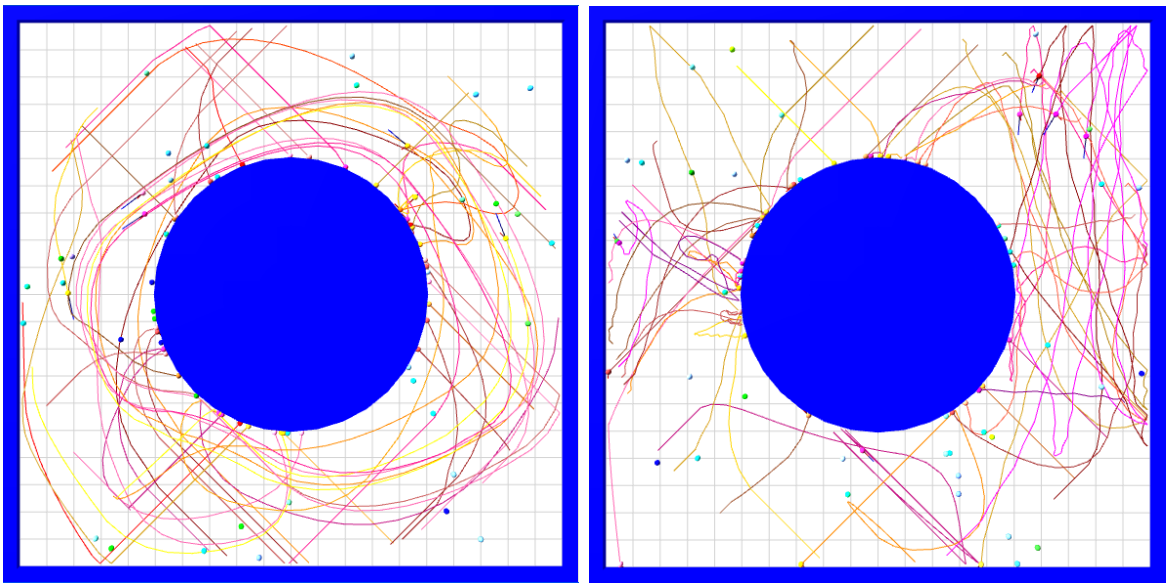


Figure 21: Hide and Seek – Average Maximum Fitness vs Generation

Although the flat fitness curve indicates a failure to evolve, the path diagrams shown in Figure 22 provide some insight into behaviors that did develop. These were generated using the same method as in previous experiments. Ten simulations per peak fitness individual were run resulting in 40 for each network type. The NNs in Figure 22a exhibited an "orbiting" behavior which served to capture the rabbit by way of covering lots of area quickly. The FNNs in Figure 22b had adopted more of a



(a) NN (b) FNN

Figure 22: Hide and Seek – Behavior Visualization

"scribbling" motion which likely served the same function. The capture rates for both sets of simulations was 2.5%, indicating that these techniques did not work reliably. Both had average scores just above 42.0.

For completeness, peak and final maximum fitness values for both network types in the hide-and-seek and baseline experiments are given as 95% confidence intervals in Table 17. These confidence intervals clearly indicate significant differences between

Experiment	Final Max Fitness	Peak Max Fitness
Hide and Seek NN	46.7 ± 8.80	62.7 ± 8.67
Hide and Seek FNN	50.1 ± 10.01	64.9 ± 1.45
Baseline NN	91.9 ± 26.84	98.7 ± 7.35
Baseline FNN	94.8 ± 11.72	100.0 ± 0.00

Table 17: Hide and Seek – 95% Confidence Intervals for Maximum Fitness

the results from the two simulation types. Unsurprisingly the t-tests agree with 95% confidence that these differences are significant even with so few samples available.

This is true for 80% of the NN data and 35% of the FNN data. Further t-tests with $\alpha = 0.05$ indicate that less than 1% of the data varies significantly between the NN and FNN for the hide and seek experiment.

Recall that the portion of the fitness function that applied before capture was based entirely on the derivative of the distance between the two creatures. The obstacle was more than 50% likely to be between the fox and the rabbit, but even in such a case, approaching the obstacle was not usually of benefit to the fox. Regardless, the fox was rewarded for approaching the obstacle so long as it was also approaching the rabbit. In this way, the fitness function was inappropriate for this simulation. Had a fitness function been used which addressed that shortcoming, better results would seem likely.

Structural Evolution

In all previously discussed experiments for this work, only parametric evolution was allowed while the network structures remained static. The experiments discussed in this section did allow structural evolution. Making structural evolution possible was the primary motivation in adopting the GNARL algorithm as the basis for the software implementation.

Each of these *structural evolution* experiments allowed nodes to be added or removed and links to be created or destroyed during each mutation. The number of each that were added and removed during each mutation was computed using Equation 12, from the "GNARL" section of Chapter 2. For both addition and deletion of nodes $\Delta_{max} = 3$ and $\Delta_{min} = 1$. For link addition $\Delta_{max} = 20$ and $\Delta_{min} = 4$, and for link deletion $\Delta_{max} = 5$ and $\Delta_{min} = 1$. These values were chosen due to their ability to keep the network size relatively constant during rapid undirected mutations

with $\hat{T} = 1.0$. The parameter used to bias link end points to input/output nodes was set to 0.2, matching the value used by Angeline, Saunders, and Pollack in [17]. The networks were all initialized from the same settings as in the baseline experiment, and the baseline simulation was utilized again.

Feed-Forward

The distinguishing characteristic of the *feed-forward structural evolution* experiment was that link creation was constrained to allow only feed-forward links, so that loops could not be generated in the computation path.

The average maximum fitness values for the NN and FNN feed-forward structural evolution experiments are shown in Figure 23. The plotted values were smoothed

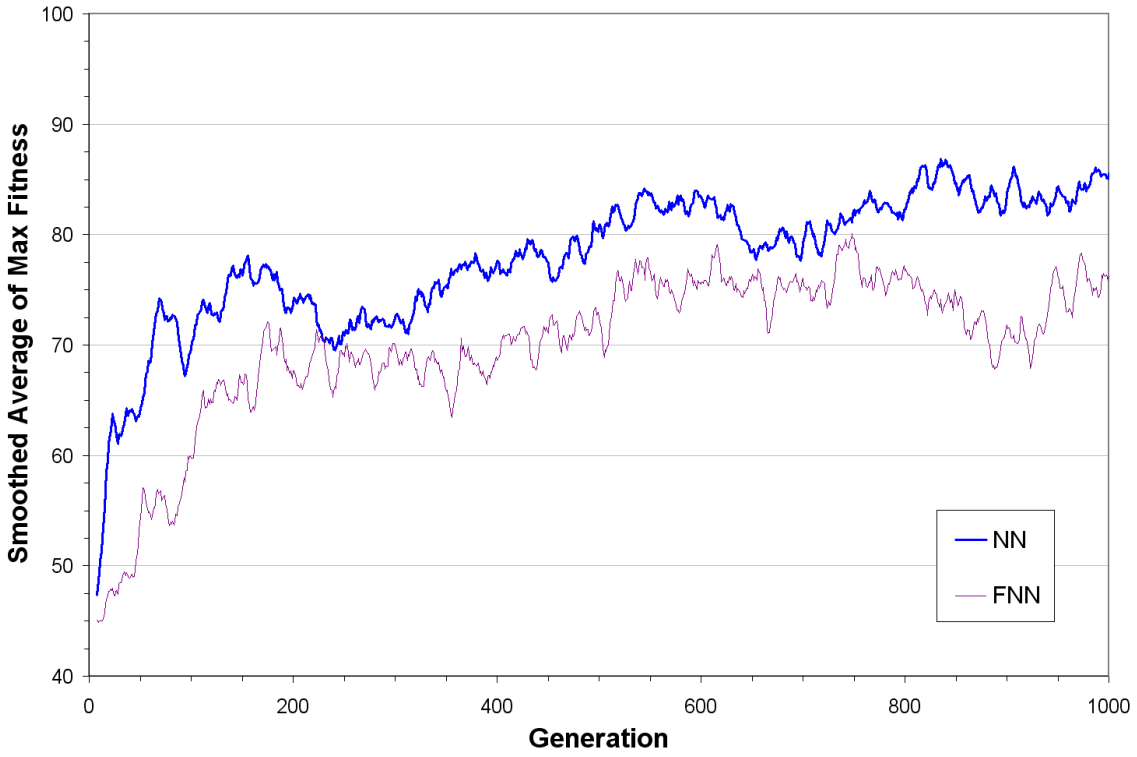


Figure 23: Feed-Forward Structural Evolution – Average Maximum Fitness vs Generation

using a moving average with a window size of nine. Visually the FNN fitness appeared to trail the NN fitness by approximately ten points throughout the evolution and both curves showed less fit final results after 1000 generations than the baseline experiment.

Peak and final maximum fitness values for both network types in the feed-forward structural and baseline experiments are given as 95% confidence intervals in Table 18. The confidence intervals seem to indicate an improvement over the baseline results

Experiment	Final Max Fitness	Peak Max Fitness
Feed-Forward Structural NN	90.3 ± 15.55	100.0 ± 0.00
Feed-Forward Structural FNN	86.9 ± 28.00	95.7 ± 15.41
Baseline NN	91.9 ± 26.84	98.7 ± 7.35
Baseline FNN	94.8 ± 11.72	100.0 ± 0.00

Table 18: Feed-Forward Structural – 95% Confidence Intervals for Maximum Fitness

for the NN, with a decrease in fitness results for the FNN. However, the t-tests with $\alpha = 0.05$ show essentially no significant difference between the feed-forward structural data and the baseline data. They also show absolutely no verifiable difference between the NN and FNN for this experiment.

The average number of nodes and links in the four peak fitness networks saved from both the NN and FNN evolutions were computed. For both network types, the average number of nodes was 37.25. The average number of links were 62.75 and 54.25 for the NN and FNN respectively. By comparison, the feed-forward topology used in the baseline experiment included 37 total nodes and 272 links. The structural evolution maintained roughly the same number of nodes (though not necessarily in the computation path). The average number of links in peak fitness individuals however, was reduced to roughly 20% of the original number.

In the feed-forward structural evolution experiments, few possibilities for additional links existed for the initial networks. Most of the allowed connections already

existed. Randomized initial topologies seem likely to improve results.

Recurrent

For the *recurrent structural evolution* experiment, link placement limitations were relaxed to allow any link except a duplicate link. These settings are consistent with those of the original GNARL algorithm, and allow loops to occur in the computation path.

The average maximum fitness values for the NN and FNN recurrent structural evolution experiments are shown in Figure 24. Again the results are smoothed using

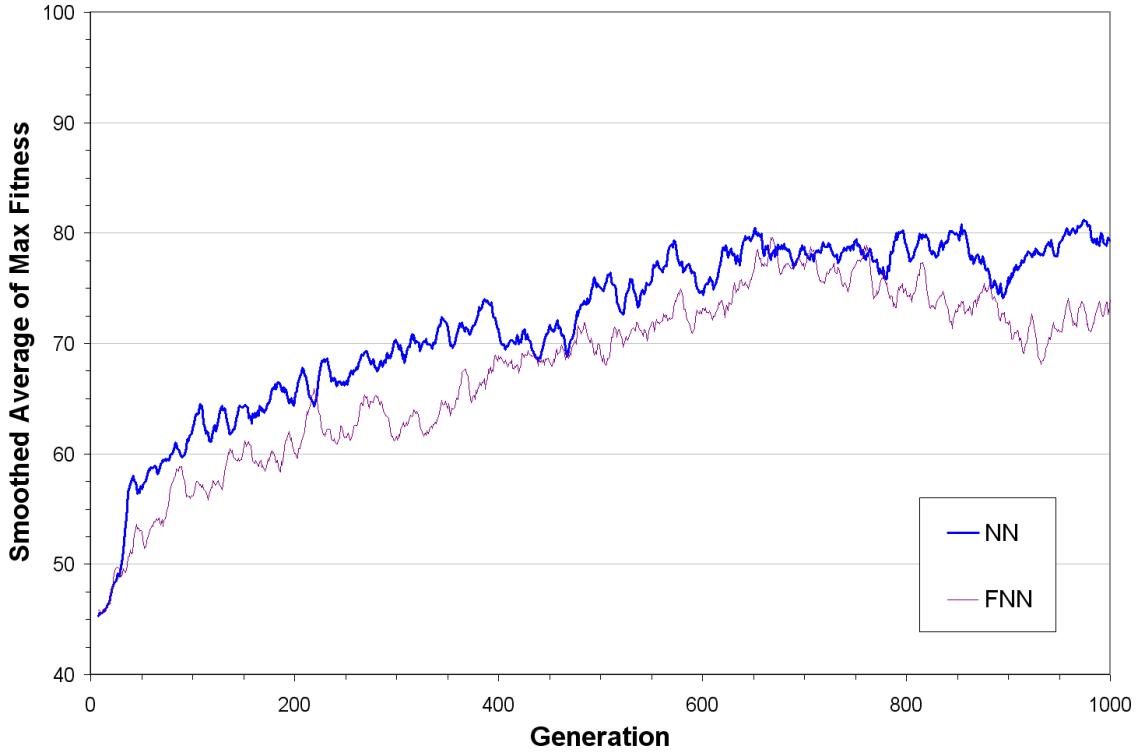


Figure 24: Recurrent Structural Evolution – Average Maximum Fitness vs Generation

a moving average with window size of nine. Again the plots indicate that the FNN consistently held a lower fitness value than the NN throughout the evolution. This

time the difference appeared to be approximately five points. The FNN curve closely followed that of the feed-forward experiment shown in Figure 23.

Peak and final maximum fitness values for both network types in the recurrent and feed-forward structural experiments along side those of the baseline experiment are given as 95% confidence intervals in Table 19. These appear to indicate a decrease

Experiment	Final Max Fitness	Peak Max Fitness
Recurrent Structural NN	82.5 ± 12.67	95.8 ± 7.82
Recurrent Structural FNN	75.4 ± 18.22	94.3 ± 18.07
Feed-Forward Structural NN	90.3 ± 15.55	100.0 ± 0.00
Feed-Forward Structural FNN	86.9 ± 28.00	95.7 ± 15.41
Baseline NN	91.9 ± 26.84	98.7 ± 7.35
Baseline FNN	94.8 ± 11.72	100.0 ± 0.00

Table 19: Recurrent Structural – 95% Confidence Intervals for Maximum Fitness

in performance compared to the baseline experiment, as well as the feed-forward experiment, especially in the case of the FNN. T-tests show that these differences are again insignificant with 95% confidence, though the FNN comparison indicated 12% of the data varied with the baseline FNN. They also show differences in less than 1% of the data comparing the NN and FNN results for the recurrent experiment.

The NN and FNN average link values were 49.75 and 47.50 respectively. The average numbers of nodes remained at approximately 37 for both network types.

Feed-Forward GA

The original feed-forward structural evolution experiment was modified to use the same evolutionary parameters used in the GA parametric-only experiment described in the "GA" section. Roulette selection was applied as before, along with the same multi-point crossover algorithm and 1% chance of non-temperature based mutation.

The average of the maximum fitness values for both the NN and FNN in this experiment are plotted over the length of the evolution in Figure 25. The results in

Figure 25: Feed-Forward Structural GA Evolution – Average Maximum Fitness vs Generation

the figure were smoothed using a moving average with window size of nine. Visually, the initial rise in the NN fitness value plot was gentler than that of the feed-forward structural evolutions shown in Figure 23. They were closely matched however, from roughly 300 generations to the end of the evolution. The FNN plot was lower throughout the evolution than for any other structural experiment. It appeared to exhibit a linear increase with a small slope.

For quantitative comparison, peak and final maximum fitness values for both network types in the feed-forward structural GA and GNARL experiments are given as 95% confidence intervals in Table 20. The 95% confidence intervals for the GA

Experiment	Final Max Fitness	Peak Max Fitness
Feed-Forward Structural GA NN	79.8 ± 49.06	95.8 ± 22.95
Feed-Forward Structural GA FNN	64.9 ± 56.58	78.5 ± 50.18
Feed-Forward Structural NN	90.3 ± 15.55	100.0 ± 0.00
Feed-Forward Structural FNN	86.9 ± 28.00	95.7 ± 15.41

Table 20: Feed-Forward Structural GA – 95% Confidence Intervals for Maximum Fitness

experiments cover noticeably wider ranges than those of the GNARL experiments, though there is still a great deal of overlap between them. T-tests with $\alpha = 0.05$ indicated that less than 7% of the data varied significantly between the structural GA and GNARL experiments for both the NN and FNN. They revealed essentially no difference between the NN and FNN within the structural GA experiment.

The average number of links in peak fitness members for the GA structural evolution NN and FNN were 103.25 and 96.0 respectively, while the average number of nodes remained at 37.25. For structural evolution, it is again reasonable to conclude that the GNARL algorithm produces results at least as good as those of the GA.

Summary of Results

Baseline fitness results showed no significant difference between the NN and the FNN. Similarly no verifiable differences were found between them in any other experiment discussed in this work. Leaving network states intact between simulations surprisingly revealed no significant change in results.

A possible trend was identified relating an increased number of nodes in the NN hidden layer directly to improved fitness. No such trend was apparent in parallel FNN experiments. The best FNN results produced corresponded to the zero and ten hidden node configurations.

Tests with fixed Q values indicated that fitness results are negatively impacted when using FNNs composed entirely of integrating units or 1^{st} order derivative units. The best results were produced using $Q = 0.25$. Evolving networks with variable Q values also returned high fitness results.

GNARL evolution produced results at least as good as the GA results for both fixed topology and evolved topology experiments with 95% confidence. No differences between the NN and FNN fitness data were established which related to the choice of evolutionary algorithm.

Behaviorally, the NN and FNN tended to produce subtly different solutions to each problem, but statistically significant differences between these behaviors were not identified.

T-tests revealed with 95% confidence that no significant difference existed between the results using the baseline simulation and those of any other simulation discussed in this work, except for hide and seek. Both network types showed similar difficulty in fitness evolution for the hide and seek experiment. Similarly, t-tests for the three types of structural evolution tested also showed with 95% confidence no significant differences in fitness results.

CHAPTER 5

FUTURE WORK

More Experimental Data

All the experimental data discussed in Chapter 4 is statistically limited in its usefulness due to the small number of evolutions (4) run for each experiment. The processing time required to run these experiments was prohibitive, which is why their number was so limited. Running more, and longer experiments would improve upon the results from this work.

More experiments and analysis regarding the differences in performance for varying numbers of nodes in the hidden layer are desired. These would qualify the existence of the identified trend in the NN experiments, and perhaps discover a relationship between the number of nodes and the fitness results for the FNN. Experiments could also be run on different topologies to detect further interesting relationships.

Results for the fixed Q value results indicated that Q values in the approximate range $[0.0, 0.5]$ may correlate with improved fitness results. Informal observations also indicated that evolved Q values tend to fall within that range. Not enough statistical data was available to say whether these observations were meaningful. More experimental data and the collection of network Q value composition statistics in FNN experiments with variable Q values would correct the problem.

Verifying consistent differences in behavior between the NN and FNN, or the lack thereof, would improve on one of the more interesting facets of this work. Behavioral differences were observed, but whether or not they can be attributed to the difference

in network type remains to be shown.

Though no statistically relevant differences were found between the network types on any discussed simulation, visual interpretation of the plots still suggests that such differences may be verified with more available data. This seems particularly likely for the sensed walls, track and structural GA experiments. As discussed, additional data also seems likely to reveal quantifiable differences among hidden node size and fixed q value experiments.

Structural Evolution

The GNARL algorithm was adopted as the basis for software implementation in this work and a generalized multi-point crossover algorithm capable of handling varying parent network topologies was developed. These and related efforts were targeted at enabling automated topology design through evolution while simultaneously adjusting network parameters for best fitness. Initial exploration of structural evolution showed promising results, but due to time constraints evolution parameters for the formal experiments discussed in this work were only minimally tuned. Continuation of the work on structural evolution is still expected to produce interesting results.

Backpropagation

In Chapter 2, the possibility of using backpropagation as a means of training FNNs was mentioned. This still remains an interesting possibility. If successful, it would allow more direct comparison of FNN learning characteristics with the many existing systems which use backpropagation as a training mechanism. The concept remains

sound and adaptation of backpropagation to FNNs may be as simple as adding the Q value of each node to the list of parameters that is adjusted using the propagated error. The Q value of a node changes the relative effect of all the outgoing link weights at once however, and may therefore require some special handling.

Fractional calculus may also be applied directly to the backpropagation algorithm used in training un-augmented neural networks. Such an attempt was made in [29]. In that system, the weight update rule was augmented with proportional integral and derivative (PID) control parameters acting on the inverse error gradient. The fractional differintegral was then applied to both the integer and derivative PID control terms resulting in $PI^{\lambda}D^{\mu}$ control parameters. More useful behavior might result from applying a differintegral operation to the gradient calculation, or to a dynamic learning rate η. Should such a system be tuned to achieve a learning performance gain, it would be equally well adapted to training FNNs as the basic backpropagation algorithm.

Simulation Complexity

The vision held throughout this work, though perhaps naive, was to evolve foxes with much richer behavioral characteristics. The original simulation involved complicated energy use characteristics which would require the foxes to manage their energy carefully so they would still have enough available when necessary for a short but extreme speed burst in making a capture. The imagined behaviors that might develop from these rules were many and varied. The foxes were expected to learn enough to catch an evasive rabbit. Once that level of complexity was reached, the rabbits could be evolved to improve their escape tactics. The foxes could then be

evolved to outperform the rabbits again, and so on. The system was designed to allow simultaneous co-evolution to occur with this objective in mind.

A simpler sensor and movement model was proposed but not implemented. In this model, the fox would face in a specific direction and would be allowed to rotate or move forward or backward. Its sensor array would be pointed in the direction it was facing so that it would have to turn to "see" more of its surroundings. Because it would cover less area, the array could be composed of fewer sensors while still providing the same quality perception of the environment. Much less redundant learning would be required with sensors tightly coupled to movement.

The mechanism of evaluating each individual using a single value at the end of several lengthy simulations creates a credit assignment problem which is difficult to alleviate by adjusting the fitness function. A more direct approach would be to implement online learning during each simulation. If weights and Q values were adjusted at small intervals throughout the simulation the creatures would receive more direct feed-back for their actions and more complex behaviors might become accessible.

The fox/rabbit simulation implementation did not appear to exhibit the characteristics necessary to make power-law modeling applicable. A simulation which directly simulated power-law dynamics might produce better results.

CHAPTER 6

CONCLUSIONS

Fitness data for the NN and FNN showed with 95% confidence that no significant difference existed between them for any simulation discussed in Chapter 4. The most obvious conclusion that can be drawn from this is that more experimental data is desired.

It was shown with 95% confidence that failing to reset FNN states between simulations in the baseline experiment did not impact the fitness results. This result was unexpected and may indicate a robustness to inconsistency in the FNN.

The hidden layer size experiment showed a possible trend relating improved fitness results directly to an increased number of nodes in the NN's hidden layer. No such trend was apparent in the FNN hidden layer size experiment. Increasing the number of nodes in the FNN's hidden layer more often resulted in lower fitness values. Findings showed that among the hidden node sizes tested, the zero and ten node configurations produced the highest fitness results with 95% confidence.

Tests with fixed Q values indicated that fitness results are negatively impacted when using FNNs composed entirely of integrating units or 1^{st} order derivative units. The best results among the fixed Q value experiments were produced using $Q = 0.25$ on the baseline simulation. The available data indicated that a correlation may exist between Q values in the approximate range $[0.0, 0.5]$ and higher fitness values. The necessary data to test this correlation with statistical relevance was not available. Evolving networks with variable Q values was also found to produce better results than many of the fixed Q value experiments. If an optimal configuration of homogeneous Q values existed, the evolution of variable Q values might be

a reasonable way to discover it. Further, if the optimal Q value were problem dependant, its automatic discovery would seem valuable.

GNARL based evolution produced results at least as good as the GA results for both fixed topology and evolved topology experiments with 95% confidence. This shows the choice of evolution to be an insignificant factor in the results.

Differences in fitness results among the five simulations tested in this work appeared plausible based on visual analysis of plotted data. T-tests revealed with 95% confidence that very little of the data varied significantly. The obvious exception was the hide and seek simulation, which clearly produced different fitness results than the baseline simulation. With more experimental data, some of these plausible differences are expected to become verifiable.

The introduction of an obstacle to the simulation environment significantly decreased fitness results in the hide and seek experiment. Adaptation of the fitness evaluation function to account for the obstacle is expected to improve results. For the evaluation to be effective, the fox must not be rewarded for approaching the rabbit unless it can sense the rabbit.

Three types of structural evolution were tested on the baseline simulation. As for other experiments, t-tests showed with 95% confidence that the results did not vary significantly from those of the baseline, or from each other. Based on visual interpretation of the fitness plots, a verifiable difference between the NN and FNN fitness results appeared plausible for all three experiments. This was particularly true in the case of the feed-forward structural GA experiments. With only four evolutions for each network type and experiment, these differences were not statistically significant but are expected to become so with slightly more data.

Differences in behavior were observed between the NN and FNN for each simulation. Whether the differences could be attributed to the change in network

type was not determined, nor were any behavioral differences identified as statistically significant. This remains as a point of future research.

The FNN tested in this work has shown fitness results comparable to those of the NN in one artificial life simulation. However, no significant advantages of the intended benefits added by the fractional augmentation were observed. The simulation tested may not have exhibited the necessary characteristics to benefit from the hypothesized natural modeling capability or lossy long-term memory available in the FNN. Differences in fitness results and behaviors between the NN and FNN were implicated by the results but not statistically significant. Whether the augmentation provides any significant advantages remains an open question.

Some exploration of fractional calculus as an augmentation to artificial neural networks was achieved in this work. As with most exploration, many more questions were raised than were answered and countless possibilities for continued exploration remain.

REFERENCES

[1] W. Duch, "Towards comprehensive foundations of computational intelligence," *Challenges for Computational Intelligence*, vol. 63, pp. 261–316, 2007.

[2] B. Hammer and T. Villmann, "Mathematical aspects of neural networks," in *European Symposium of Artificial Neural Networks 2003*, M. Verleysen, Ed., 2003, pp. 59–72.

[3] R. Setiono and W. K. Leow, "Pruned neural networks for regression," in *Proceedings of the 6th Pacific Rim Conference on Artificial Intelligence, PRICAI 2000, Lecture Notes in AI 1886*, 2000, pp. 500–509.

[4] K.-C. Jim, C. L. Giles, and B. G. Horne, "An analysis of noise in recurrent neural networks: Convergence and generalization," *IEEE Transactions on Neural Networks*, vol. 7, pp. 1424–1439, 1996.

[5] M. J. L. Orr, "Introduction to radial basis function networks," Centre for Cognitive Science, University of Edinburgh, Tech. Rep., 1996.

[6] T. Kohonen, "Automatic formation of topological maps of patterns in a self-organizing system," in *Proceedings of 2nd Scandinavian Conference on Image Analysis, Espoo, Finland*, 1981, pp. 214–220.

[7] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilites," *Proceedings of National Academy of Science, USA*, vol. 79, pp. 2554–2558, 1982.

[8] L. O. Chua, "Cellular neural networks: Theory," *IEEE Transactions On Circuits And Systems*, vol. 35, no. 10, pp. 1257–1272, 1988.

[9] J. Baxter, "The evolution of learning algorithms for artificial neural networks," in *Complex Systems*, G. D. and B. T., Eds. IOS Press, 1992, pp. 313–326.

[10] T. D. Sanger, "Optimal unsupervised learning in a single-layer linear feedforward neural network," *Neural Networks*, vol. 2, pp. 459–473, 1989.

[11] G. W. Bohannan, "Application of fractional calculus to polarization dynamics in solid dielectric materials," Ph.D. dissertation, Montana State University, Bozeman, Montana, November 2000.

[12] Y. Chen, B. M. Vinagre, and I. Podlubny, "Fractional order disturbance observer for robust vibration suppression," *Nonlinear Dynamics*, vol. 38, no. 1–2, pp. 355–367, 2004.

[13] Y. Chen, "Ubiquitous fractional order controls?" in *The Second IFAC Symposium on Fractional Derivatives and Applications (IFAC FDA06)*, JUL 2006.

[14] J. A. T. Machado, "Analysis and design of fractional-order digital control systems," in *SAMS - Journal Systems Analysis-Modeling-Simulation*. Edinburgh: OPA (Overseas Publishers Association), 1997, vol. 27, pp. 107–122.

[15] X. Yao, "Evolving artificial neural networks," in *Proceedings of the IEEE*, vol. 87, no. 9, 1999, pp. 1423–1447.

[16] X. Yao and Y. Liu, "Towards designing artificial neural networks by evolution," *Applied Mathematics and Computation*, vol. 91, pp. 83–90, 1998.

[17] P. J. Angeline, G. M. Saunders, and J. P. Pollack, "An evolutionary algorithm that constructs recurrent neural networks," *IEEE Transactions on Neural Networks*, vol. 5, no. 1, pp. 54–65, JAN 1994.

[18] B. Krose and P. Van Der Smagt, *An Introduction to Neural Networks*, 8th ed. The University of Amsterdam, 2006, pp. 15–31, 33–46.

[19] P. J. Werbos, *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. Wiley, John & Sons, Inc., 1994.

[20] M. Bodén, "A guide to recurrent neural networks and backpropagation," in *The Dallas project, SICS Technical Report T2002:03, SICS*, 2002.

[21] T. M. Mitchell, *Machine Learning*. WCB/McGraw-Hill, 2007, pp. 81–124.

[22] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, MAY 1983.

[23] R. Storn and K. Price, "Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[24] K. A. D. Jong, "An analysis of the behavior of a class of genetic adaptive systems," Ph.D. dissertation, University of Michigan, Ann Arbor, 1975.

[25] M. Mitchell, S. Forrest, and J. H. Holland, "The royal road for genetic algorithms: Fitness landscapes and ga performance," in *Proceedings of the First European Conference on Artificial Life*. MIT Press, 1992, pp. 245–254.

[26] F. Carpenteri and F. Mainardi, Eds., *Fractals and Fractional Calculus in Continuum Mechanics*, ser. CISM Courses and Lectures. New York: Springer-Verlag New York, Inc., 1997, vol. 378, p. 224.

[27] K. B. Oldham and J. Spanier, *The Fractional Calculus Theory and Applications of Differentiation and Integration to Arbitrary Order*, ser. Mathematics in Science and Engineering, R. Bellman, Ed. Academic Press, Inc., 1974, vol. 111.

[28] C. A. Monje, A. J. Calderon, B. M. Vinagre, Y. Chen, and V. Felíu, "On fractional pi^{λ} controllers: Some tuning rules for robustness to plant uncertainties," *Nonlinear Dynamics*, vol. 38, no. 1–2, pp. 369–381, 2004.

[29] S. Gardner, R. Lamb, and J. Paxton, "An initial investigation on the use of fractional calculus with neural networks," in *Proceedings of the IASTED International Conference COMPUTATIONAL INTELLIGENCE*, NOV 2006, pp. 182–187.

APPENDICES

Appendix A

SOFTWARE

The software solution implemented for this work was considerably more feature rich than required to run the experiments discussed in Chapter 4. This appendix describes the software implementation in detail and may be of use in continued exploration of fractional calculus as applied to neural networks.

The software implementation is embodied in a C# program designed to be compiled and run under the Microsoft Windows® operating system. It is called *Evolver*. Evolver is a graphical user interface (GUI) to a generalized evolutionary algorithm. One of the primary motivations for the GUI was to make program operations easily observable for verification of its proper operation.

Evolution Monitor

Evolver was designed to execute multiple instances of a given experiment using a thread pool which could easily saturate the computational resources of the host computer. The main application window is pictured in Figure 26. It allows for

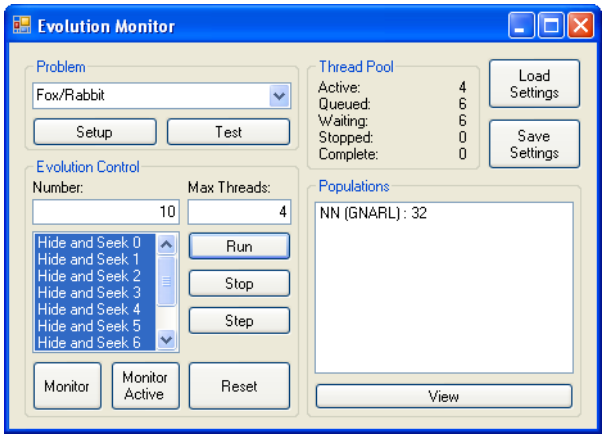


Figure 26: Evolution Monitor – Evolver Main Window

problem selection, management of the evolution thread pool, and saving or loading

of evolution settings. Interfaces to other parts of the program can also be spawned from the main window.

The problem selector in the upper left allows the problem definition to be changed at runtime. The original plan for this work was to study FNNs in multiple problem domains, but the work required to get high quality results from one problem turned out to be more than enough. Other problems toyed with during development include spatial classification functions, sine wave generation, and predator/prey co-evolution. Only the fox/rabbit problem definition is fully functional in the current version of Evolver.

The "Evolution Control" section of the main window allows the user to specify how many evolution instances to run, and how many threads to make available in the thread pool. The list view displays all evolutions in the experiment. Any subset of these may be selected for manipulation as a group. They may be started, stopped, or stepped one generation at a time. Monitor windows may be spawned for progress overviews and current status of each evolution. The "Reset" button clears all current evolutions so that new ones can be started with the current settings. A red message appears above this button to indicate when settings have been changed that require a reset to take effect.

The thread pool information displayed in the upper right section of the window provides a quick overview of what's going on in the thread pool. This indicates the current number of running or active threads. It also shows the number of evolutions in the queue, the number which are waiting and ready to run as soon as a thread becomes available, and the number which have been stopped by the user. The number of completed evolutions is also provided.

The population view lists all the populations that exist in each of the running evolutions. Any set of populations in the running evolutions may be viewed by

selecting the relevant evolution(s) and population(s) before activating the "View" button.

The settings for the current evolution can be saved to or loaded from disk using the buttons in the upper right. These provide common Windows dialog boxes for file selection.

Monitor Windows

A monitor window may be displayed for each separate evolution with shortcuts for displaying just the active ones or hiding all of them. These may be redefined in the problem definition if display of more information is desired for a particular problem. The default monitor window is shown in Figure 27, giving a quick summary

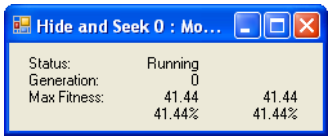


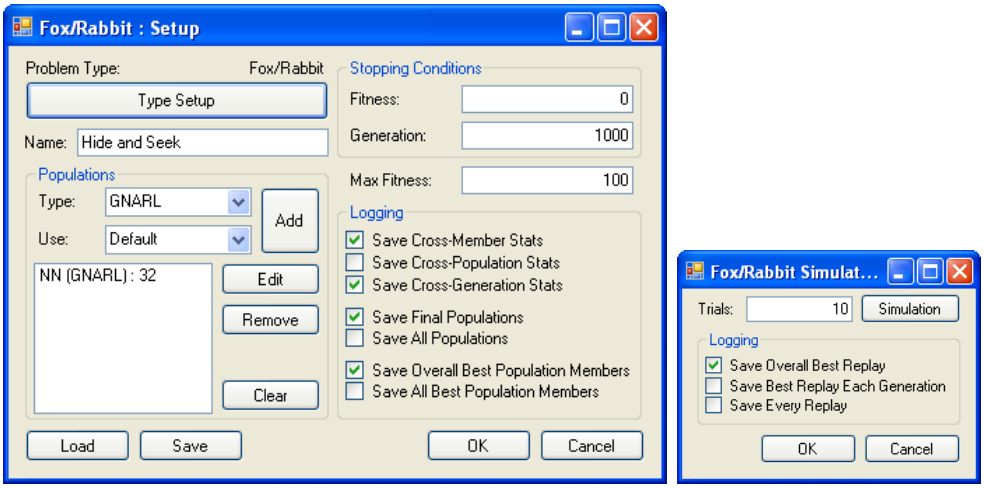
Figure 27: Monitor Window

of the maximum and peak fitness as raw values and percentages (these often matched due to selected fitness ranges $[0, 100]$). It is possible for the system to display animated simulation results in the monitor window as they occur. This was done while developing the predator/prey co-evolution problem.

Evolution Setup

The "Setup" button in the main window provides access to all the evolution settings. Many of these are specific to the selected problem. Access to all these settings is provided through the GUI by way of cascading non-modal dialog boxes.

Figure 28 shows the general evolution settings dialog alongside the fox/rabbit problem



(a) General (b) Fox/Rabbit

Figure 28: Evolution Setting Setting Dialogs

specific evolution settings dialog. The general setup includes an evolution name specification and stopping conditions. Logging options are also present to toggle the generation of various output files. These range from the aggregated statistics discussed in the "Statistic Collection" section at the end of Chapter 3, to peak fitness individuals and whole populations.

A modifiable list of populations is also provided. Each population in the list will be present in every evolution instance. The problem specific settings are accessed by clicking the "Type Setup" button from the general settings dialog. Figure 28b shows the result for the fox/rabbit problem used in this work. Here, the number of simulation trials run for member evaluation can be specified. Some additional output file selections for saving certain simulation replays to disk are also available. The "Simulation" button provides access to all the fox/rabbit simulation settings which will be discussed in the "Simulation Settings" subsection that follows.

When the user chooses to edit a population in the list, a dialog similar to the one

in Figure 29 is presented. This is the general population editing dialog, but member

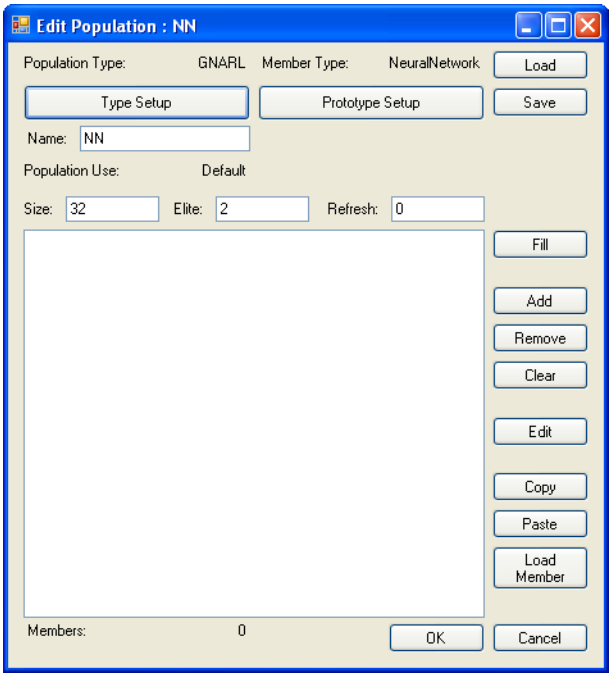
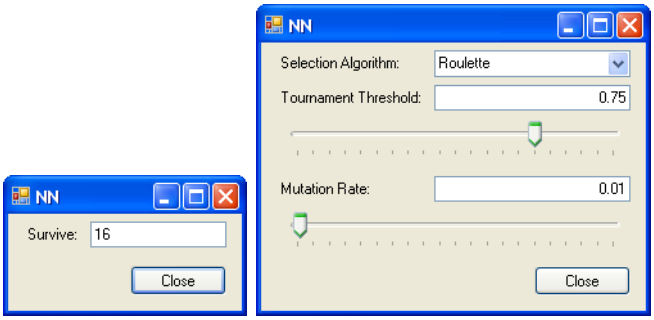


Figure 29: Population Edit Dialog

types are controlled by the problem description. If the number of members added here does not match the population size, each evolution instance fills the population with randomly initialized members before the evolution begins. Some population type specific settings may also exist. These are accessed by clicking the "Type Setup" button.

The population type specific dialogs for the two available population types are shown in Figure 30. The only setting provided in the GNARL population edit dialog is the number of population members that survive to form each following generation through mutation. This value is maintained as a percentage of the size when it is changed in the main population edit dialog. The GA population edit dialog allows a choice of selection algorithm, modification of some of the selection algorithms'



(a) GNARL (b) GA

Figure 30: Population Type Specific Setting Dialogs

parameters, and a fixed member mutation rate adjustment. Selection algorithms include "roulette", "tournament", and "uniform" selection.

For the fox/rabbit problem used in this work, the only allowed population members are of the neural network type. The "Add Member" dialog shown in Figure 31 is specific to neural network members. The text representation of the

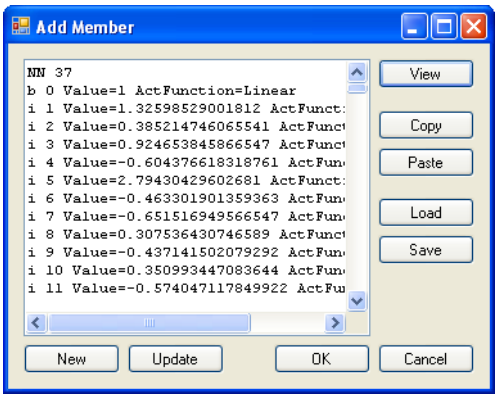


Figure 31: Add Neural Network Member Dialog

neural network is directly modifiable from this dialog, and a graphical interactive view can be spawned using the "View" button. This will be illustrated in a later section.

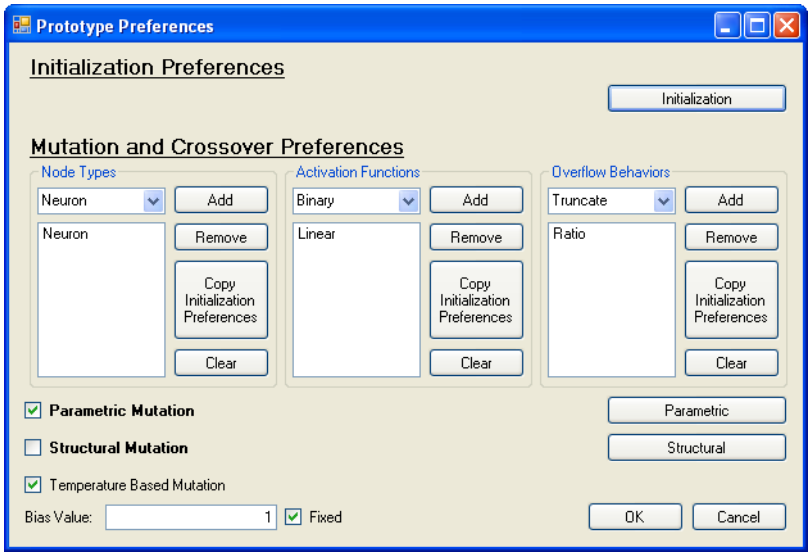
The population member prototypes can be configured by clicking on the "Prototype Setup" button in the main population edit dialog from Figure 29. This

spawns the "Prototype Preferences" dialog shown in Figure 32a. Again, this dialog is specific to the neural network members used in the fox/rabbit problem. The prototype controls the initialization and mutation for all members of the corresponding population. The main prototype editing dialog provides the user with a means of defining the pool of allowed node types, activation functions, and fractional history overflow behaviors. There are also options to force all bias nodes to a specified value or toggle temperature based mutation. The initialization preferences, parametric mutation, and structural mutation options are accessed by clicking their respective buttons.

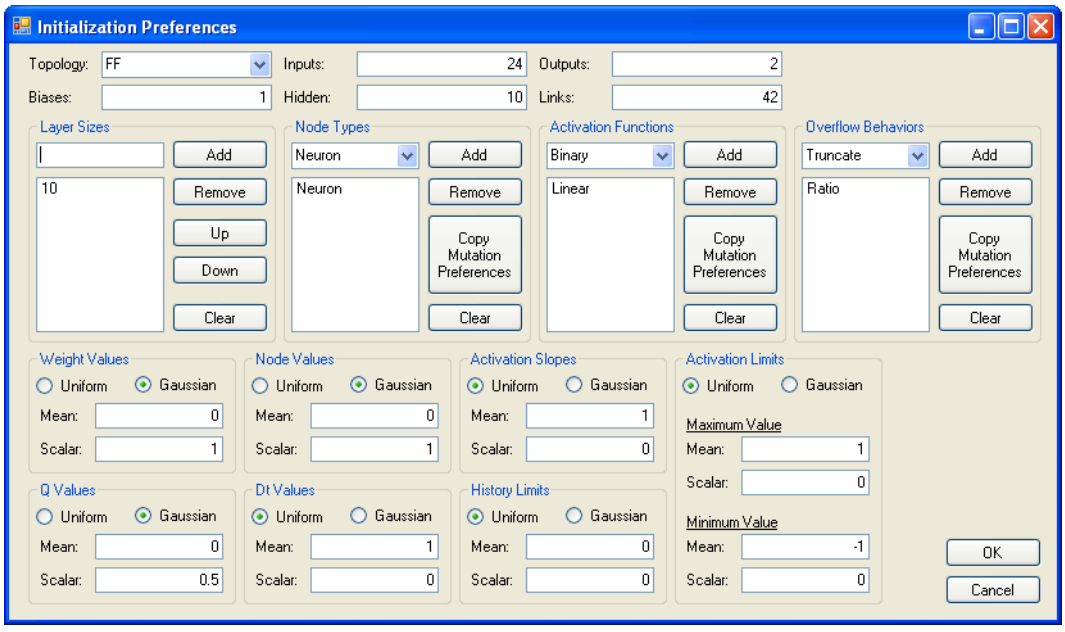
Figure 32b shows the "Initialization Preferences" dialog which allows the user to select from several predefined topologies for initial networks. This includes a "Random" topology setting which generates the appropriate numbers of each node type and connects them randomly using the specified number of links and specified link placement options. The layers for more rigid topologies can be defined in the list to the left, and the pools of allowed node types, activation functions, and fractional history overflow behaviors can be specified separately for initial networks. The bottom half of the "Initialization Preferences" dialog allows the user to specify random distribution types and ranges for all the initial node and link parameters.

The "Parametric Preferences" dialog pictured in Figure 33a allows parametric mutation settings to be modified. These include the scale of link weight and node value adjustments. Several activation function adjustment settings are provided, along with the differintegral parameter related options.

The "Structural Preferences" dialog shown in Figure 33b allows several settings that affect structural mutation to be modified. Some of these settings also apply to neural network initialization when the "Random" topology is selected. The adjustable structural mutation preferences include minimums and maximums for

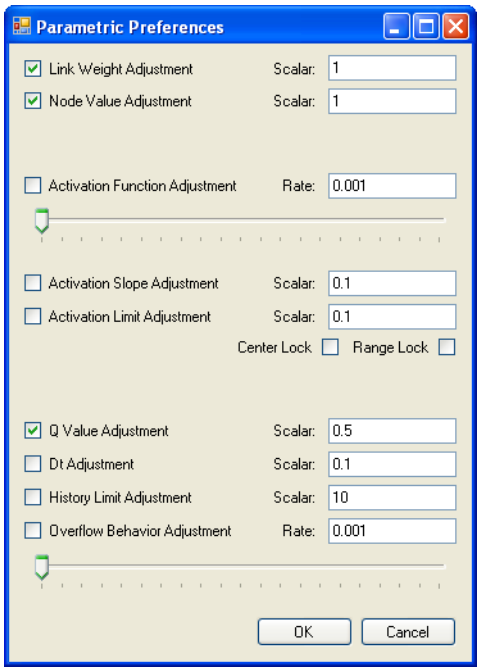


(a) Main Prototype Settings

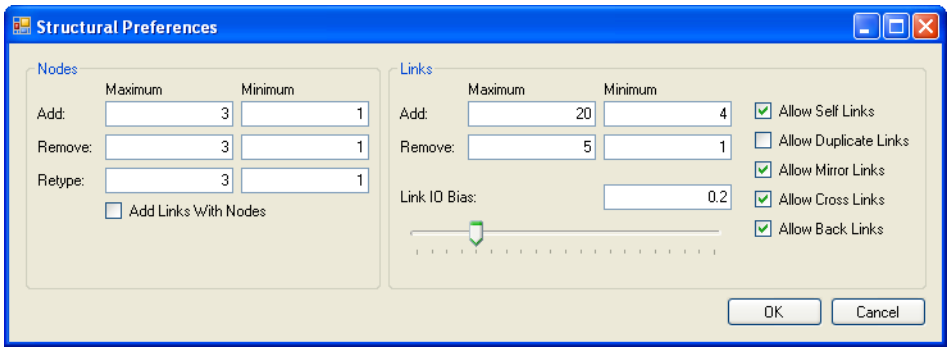


(b) Initialization Settings

Figure 32: Neural Network Prototype Preference Dialogs



(a) Parametric Settings



(b) Structural Settings

Figure 33: More Neural Network Prototype Preference Dialogs

adding, removing and type changing of nodes. Link addition and removal count ranges are also specified here, as well as the IO bias parameter. Allowed link types are toggled on the right side.

Simulation Settings

The fox/rabbit simulation settings are accessible via the "Settings" button in the fox/rabbit problem specific evolution settings dialog pictured in Figure 28b. The resulting main simulation settings dialog is shown in Figure 34. It provides choices

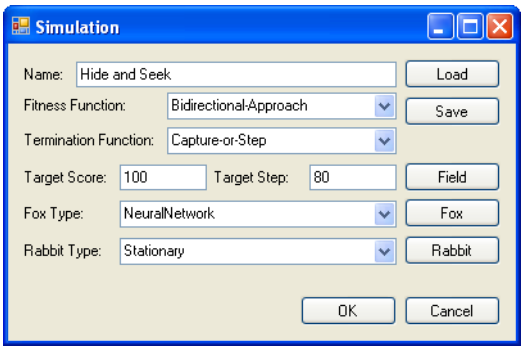


Figure 34: Fox/Rabbit Simulation Settings Dialog

for the applied fitness function and options for simulation stopping conditions. The type of both the fox and rabbit can be selected, although the problem is designed to evolve the fox as a neural network, so other options should not be chosen in this scope. Configuration dialogs for the field, and for both the fox and rabbit can be spawned using their respective buttons.

The "Field" settings dialog pictured in Figure 35 allows the size of the simulation field to be specified, along with a list of obstacles and their dimensions. An option to toggle field boundaries, and one to toggle the creatures' perception of them is also provided here.

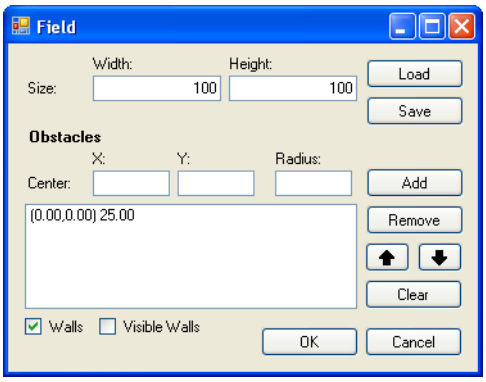
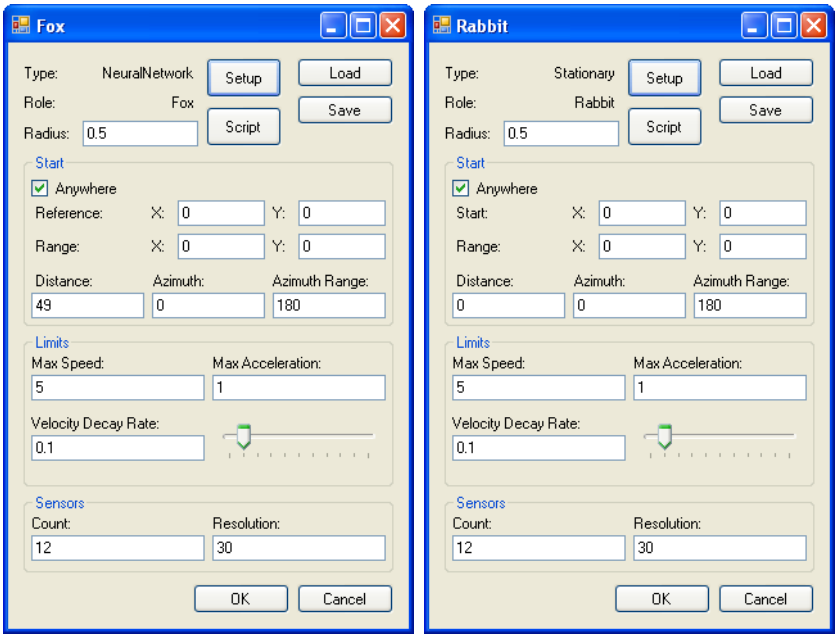


Figure 35: Simulation Field Settings Dialog

Both the fox and rabbit are configured independently but have many common settings provided by a general creature settings dialog. This is shown for both the fox and rabbit in Figure 36. This dialog allows the size of the creature to be

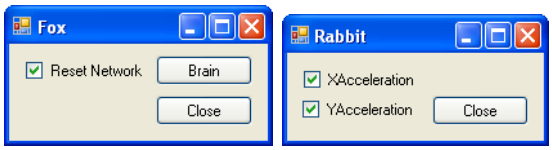


(a) Fox (b) Rabbit

Figure 36: Simulation Creature Setting Dialogs

specified, along with an acceptable range of starting positions. The starting positions can be allowed to fall anywhere on the field, constrained to a rectangular area, or

constrained to an arc by radius azimuth range. The maximum speed and acceleration magnitudes can also be set here, as well as the velocity decay factor applied at each time step. The number of sensors in the sensor array and their resolution can be set separately for each creature as well. The "Setup" button spawns creature type specific settings. Figure 37 shows the specific configuration option dialogs for the two



(a) Neural Network (b) Random

Figure 37: Type Specific Creature Setting Dialogs

creature types that implement them. The neural network creature type allows the resetting of network state information between simulations to be toggled, along with specification of a particular "Brain" or neural network. The "Brain" button opens the same member editing dialog shown in Figure 31. In the case of the fox, this is always overridden by the evolved network being evaluated. The random creature type allows movement on either axis to be toggled independently. A movement script can be specified by spawning a "Script" dialog from either creature setup dialog. It is shown in Figure 38 If a non-empty script is entered into the script configuration

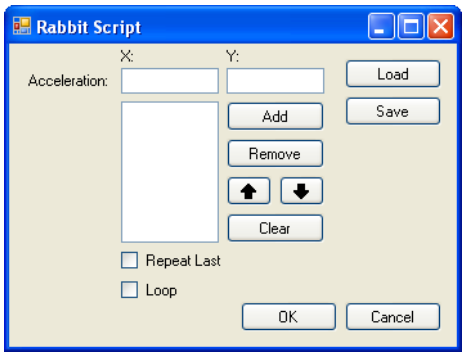


Figure 38: Creature Script Dialog

dialog, the list of acceleration vectors will be applied to the creature in order to control the creature's movement at each time step. The script may loop, or continue to use the final value indefinitely.

Setting Storage

All the settings configurable in the GUI, and a few others, are maintained as a tree in memory. Any branch can be written to disk in plain text format which can be easily modified by the user if desired. The experiment settings as saved or loaded from the Evolver main window are in this format. The baseline FNN experiment settings are provided in Listing A.1, as an example.

```
// Name: Evolver
// FNN.set
// 2/15/2009 11:53 PM
// Evolver version: 0.3
Evolutions = 2
MaxThreads = 2
EA | Fox/Rabbit {
  name = Baseline
  maxPossibleFitness = 100
  targetFitness = 0
  targetGeneration = 1000
  saveGenerationalBestMembers = False
  saveBestMembers = True
  saveAllPopulations = False
  saveFinalPopulations = True
  saveMemberStats = True
  savePopulationStats = False
  saveGenerationStats = True
  trials = 10
  saveEveryReplay = False
  saveGenerationalBestReplays = False
  saveBestReplays = True
  Population | GNARL {
    seed = -1
    name = FNN
    use = Default
    size = 32
    elite = 2
    refresh = 0
    survive = 16
    Prototype | NeuralNetwork {
      seed = -1
```

```
maxPossibleFitness = 100
initialTopology = FF
initialLayerSize = 10
initialNodeTypeValue = Fractional
initialActivationFunctionValue = Linear
initialOverflowBehaviorValue = Ratio
initialInputs = 24
initialOutputs = 2
initialBiases = 1
initialHiddenNodes = 10
initialLinks = 42
initialGaussianWeights = True
initialGaussianValues = True
initialGaussianActivationSlopes = False
initialGaussianActivationLimits = False
initialGaussianQs = True
initialGaussianDts = False
initialGaussianHistoryLimits = False
initialWeight = 0
initialWeightRange = 1
initialValue = 0
initialValueRange = 1
initialActivationSlope = 1
initialActivationSlopeRange = 0
initialActivationMax = 1
initialActivationMaxRange = 0
initialActivationMin = -1
initialActivationMinRange = 0
initialQ = 0
initialQRange = 0.5
initialDt = 1
initialDtRange = 0
initialHistoryLimit = 0
initialHistoryLimitRange = 0
nodeTypeValue = Fractional
activationFunctionValue = Linear
overflowBehaviorValue = Ratio
parametricMutation = True
structuralMutation = False
temperatureBasedMutation = True
weightMutation = True
valueMutation = True
activationFunctionMutation = False
activationSlopeMutation = False
activationLimitMutation = False
activationCenterLock = False
activationRangeLock = False
qMutation = True
dtMutation = False
historyLimitMutation = False
overflowBehaviorMutation = False
```

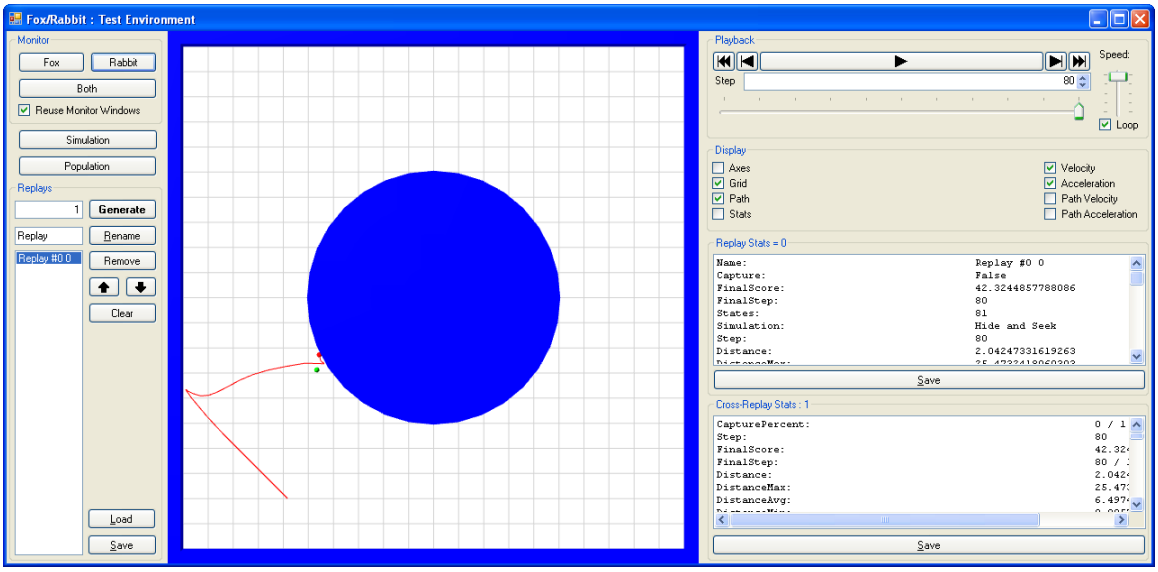
```
    biasValue = 1
    weightScale = 1
    valueScale = 1
    activationFunctionChangeRate = 0.001
    activationSlopeScale = 0.1
    activationLimitScale = 0.1
    qScale = 0.5
    dtScale = 0.1
    historyLimitScale = 10
    overflowBehaviorChangeRate = 0.001
    maxNodeAdd = 3
    minNodeAdd = 1
    maxNodeDel = 3
    minNodeDel = 1
    maxNodeRetype = 3
    minNodeRetype = 1
    addLinksWithNodes = False
    maxLinkAdd = 20
    minLinkAdd = 4
    maxLinkDel = 5
    minLinkDel = 1
    linkIOBias = 0.2
    allowSelfLinks = True
    allowDuplicateLinks = False
    allowMirrorLinks = True
    allowCrossLinks = True
    allowBackLinks = True
  }
}
Simulation {
  seed = -1
  name = Baseline
  fitnessFunction = Bidirectional-Approach
  terminationFunction = Capture-or-Step
  funnelDistanceScalar = 0.5
  funnelAccelScalar = 0.002
  funnelEpsilon = 0.001
  proximityThreshold = 2
  graduatedProximityPercent = 0, 1
  graduatedProximityPercent = 0.707106781186548, 0.95
  graduatedProximityPercent = 1.4142135623731, 0.9
  graduatedProximityPercent = 2.12132034355964, 0.85
  graduatedProximityPercent = 2.82842712474619, 0.8
  graduatedProximityPercent = 3.53553390593274, 0.75
  targetScore = 100
  targetStep = 40
  Field {
    width = 100
    height = 100
    walls = True
    visibleWalls = False
```

```
    }
    Fox | NeuralNetwork {
      seed = -1
      radius = 0.5
      startAnywhere = False
      start = 0, 0
      startRange = 0, 0
      startDistance = 49
      startAzimuth = 0
      startAzimuthRange = 180
      maxSpeed = 5
      maxAccel = 1
      velocityDecayRate = 0.1
      role = Fox
      sensorCount = 12
      sensorResolution = 30
      resetNetwork = True
      Brain {
        NN 0
      } data
    }
    Rabbit | Stationary {
      seed = -1
      radius = 0.5
      startAnywhere = False
      start = 0, 0
      startRange = 0, 0
      startDistance = 0
      startAzimuth = 0
      startAzimuthRange = 180
      maxSpeed = 5
      maxAccel = 1
      velocityDecayRate = 0.1
      role = Rabbit
      sensorCount = 12
      sensorResolution = 30
    }
  }
}
```

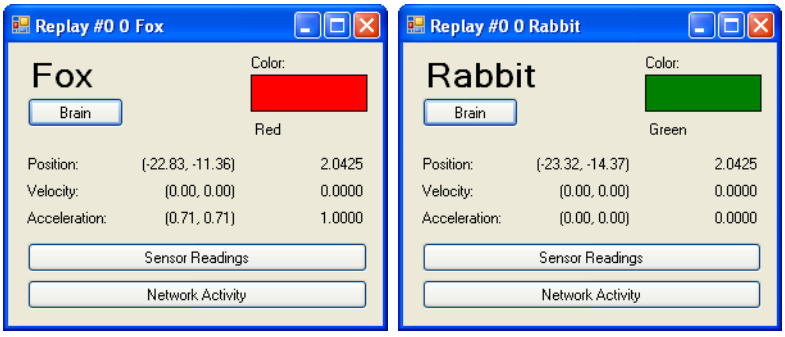
Listing A.1: Baseline FNN.set

Test Environment

The "Test" button in the main window provides access to the problem specific test environment. The idea behind the test environment was to provide a sort of sandbox in which populations and individual members could be tested under a variety of conditions with instant visual feedback. The fox/rabbit test environment pictured in Figure 39 provides a simulation replay viewer and generator. A large variety of



(a) Main Window



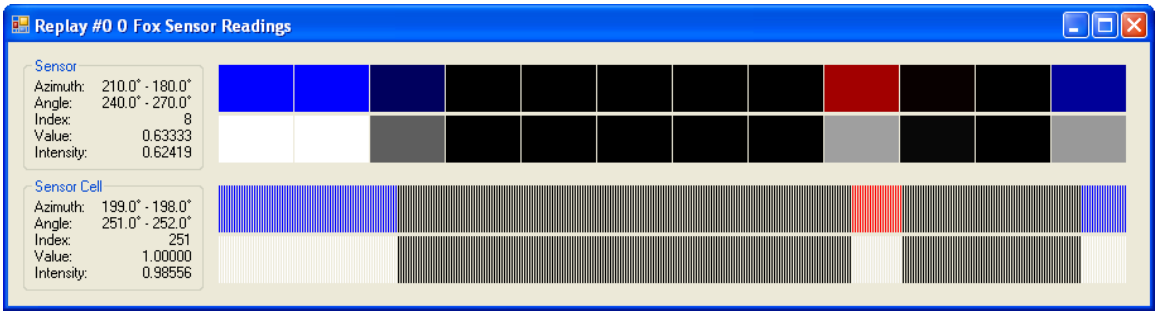
(b) Fox Monitor (c) Rabbit Monitor

Figure 39: Fox/Rabbit Test Environment

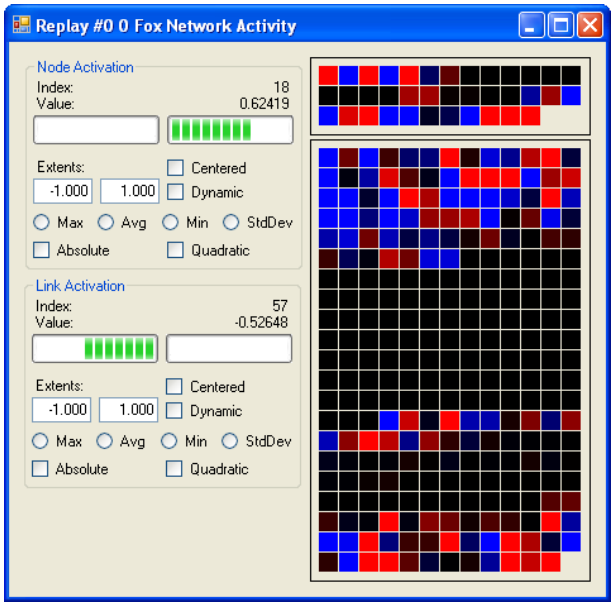
statistics are generated from each individual replay and for the currently selected

group of replays in the two panes to the lower right. Step by step traversal of replays is provided as well as automated playback at adjustable speed. The "Simulation" button opens the simulation settings dialog from Figure 34 as it applies to the local test environment simulation. All settings provided by its subtree of dialogs are also adjustable. The "Population" button similarly opens the population edit dialog from Figure 29. This controls a population of members local to the test environment which can be used to generate simulation replays for playback and observation of statistics. The field to the left of the "Generate" button controls the number of replays generated for each member of the population when the button is activated.

Each of the creature monitors shown in Figures 39b and 39c update with the corresponding creature's movement as the main test environment window plays back the selected replay. These monitor windows also provide access to "Sensor Readings" and "Network Activity" for the creature they represent. Figure 40a shows the sensor readings corresponding to the state of the test environment in Figure 39a. The information in the sensor readings window updates according to mouse-over input and as the replay state changes in the main test environment window. Each pair of vertically stacked colored blocks in the top half of the window represents one sensor. It is shaded according to the value reported by the sensor. Blue represents walls and obstacles, while red represents other creatures. The intensity values are greyscale. Similarly, each pair of much narrower stacked color bars in the bottom half of the window represents a sensor cell. Figure 40b displays the values of the nodes and links in the creature's "Brain". This data also updates according to mouse-over input and to track replay state changes in the main test environment window. The colored squares arranged in a grid in the top section of the window represent the nodes of the network in internally indexed order. These are shaded to match their *activation values*. The activation value of an input or bias node is simply its value, while the



(a) Sensor Readings



(b) Network Activity

Figure 40: Creature Detail Monitors

activation value of a hidden or output node is its propagation value. The grid in the lower section represents all the links of the network in internally indexed order. They too are shaded to match their activation values. For links, the activation value is the output value of its input node multiplied by the link's weight. Red represents positive values and blue represents negative values. Control panels are available on the left side of the window for both nodes and links. These display quantitative values for the last selected square, and offer several modifiers that affect the color display.

Also accessible from the creature monitors pictured in Figures 39b and 39c, are

visual representations of each creature's "Brain" through the neural network viewer. This is the same neural network view available by clicking "View" on any of the various member edit dialogs from Figure 31 throughout the program. It is shown in Figure 41. This window provides an interactive 3-dimensional (3D) representation

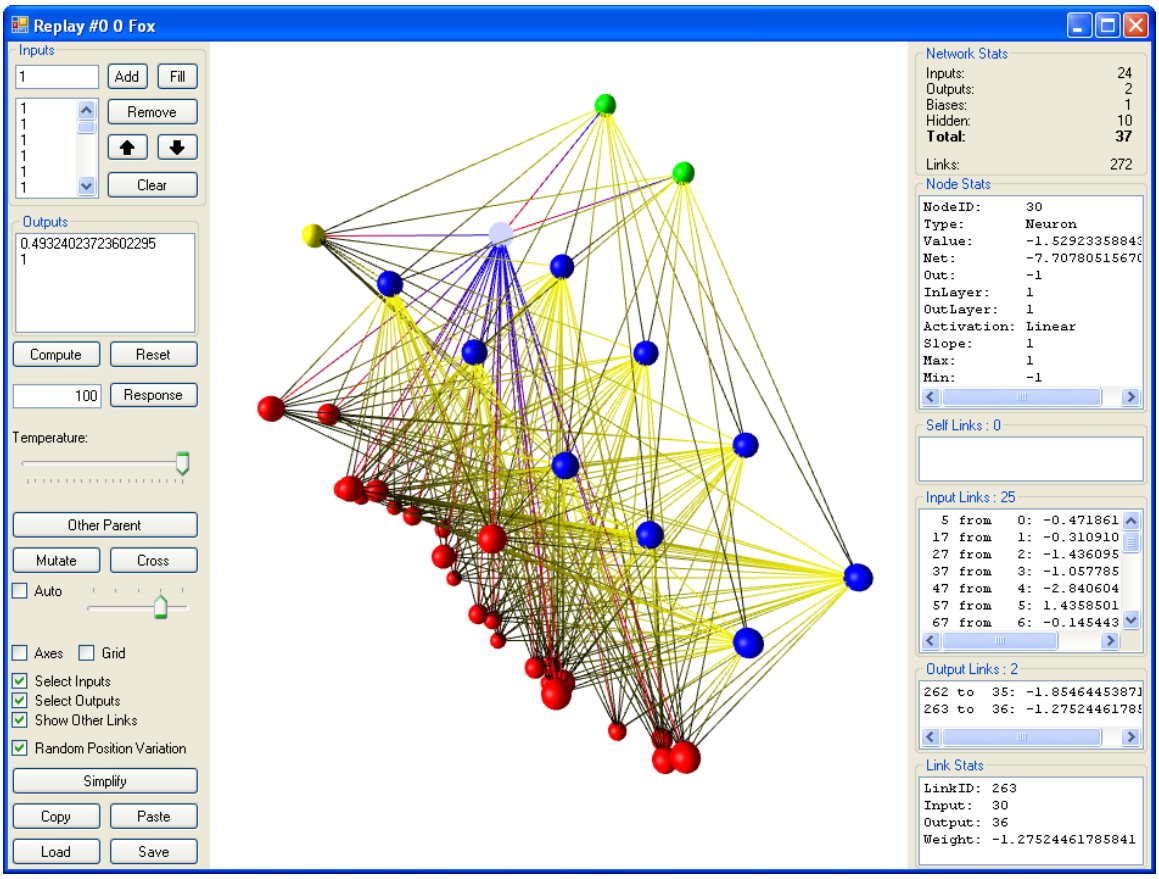


Figure 41: Neural Network Viewer

of the loaded neural network. In the 3D model, colored spheres represent network nodes, and links are colored lines drawn between them. The links are drawn with a color gradient to indicate direction, with black at their input and yellow at their output. Red spheres represent input nodes, yellow represents a bias node, and green an output. Hidden nodes are blue. Fractional nodes always appear as purple spheres. Note that the fractional augmentation has no effect on an input or a bias node, so

those node types cannot be changed. When a node is selected, it is lightened in color and enlarged. Its incident links may also be selected depending on user specified options. When a link is selected it changes to a red–blue gradient with blue at the output.

The right side of the window provides basic network component counts and state information for the currently selected node and its links. The primary purpose of the neural network viewer was to allow a neural network to be inspected visually, to ensure that the user's concept of the network matched the program's model.

The upper left section of the window allows inputs to be specified. The resulting outputs are then displayed when computed. The "Reset" button clears any state information in the network to allow for fresh computation. The field to the left of the "Response" button specifies how many compute cycles are run and stored to disk when the button is clicked. Below these controls is a set of controls for testing mutation and crossover operations. The "Other Parent" button opens a dialog similar to the one in Figure 31 which allows another network to be loaded for crossover testing. The temperature slider naturally controls the instantaneous temperature value for mutation. The "Mutate" and "Cross" buttons apply the mutation or crossover operations respectively. If the "Auto" box is checked, these will toggle the automatic application of the corresponding operator at a rate controlled by the slider to its right. The "Simplify" button prunes nodes and links from the network which are not in the computation path.

Statistic Aggregation

The layered approach to statistic aggregation described in the "Statistic Collection" section at the end of Chapter 3 was done with the help of self merging lists

of statistic objects. Each relevant quantity was assigned a name and added to the statistic list at the appropriate level. Transfer of these statistics to higher levels was done using functions that automatically expanded each statistic object in the list into a group of statistic objects where each represented a component of the previous level statistic. A long list of named quantities (115,415 for the current version of Evolver) resulted at the final level of aggregation. Logging options were used to select which levels of statistics were written to disk.

A separate program was written to aggregate the statistic data output from multiple evolutions. This provided the convenience of running discontinuous evolutions by parts, possibly even on multiple host computers while still allowing for easily acquired experiment level statistics.

Appendix B

CODE LISTINGS

The source code for Evolver is much too lengthy to include here. Instead some partial source code listings are given in order to provide detailed records of Evolver's implementation. Though they are presented as complete class definitions, much of their functionality has been omitted. The removed code was either obvious in implementation, or was deemed distracting from the primary purpose of its containing class. These are grouped by function in the sections that follow.

Differintegral Code

```
/// <summary>
/// FractionalDifferIntegrator class
/// Fractional Differentiator / Integrator
/// By: Sam Gardner
/// 10-18-2008
/// </summary>
public class FractionalDifferIntegrator
{
  public enum OverflowBehavior
              { Truncate, Coopmans, Exponential, Ratio }


  private double q;            //Order of derivative(+) / integral(-)
  private double dt;           //Sample period (time step)

  private int historyLimit;           //History size limit
  private OverflowBehavior behavior; //History overflow behavior

  private List<double> history; //History list history[0] = oldest
  private List<double> weight;  //Weight list weight[0] = newest

  private bool retainHistory; //Keep history on resize or change in q?


  /// <summary>
  /// Create a new fractional differintegrator
  /// </summary>
  public FractionalDifferIntegrator()
  {
    q = 0.5;
    dt = 1.0;

    historyLimit = 0;
    behavior = OverflowBehavior.Ratio;
```

```
    history = new List<double>();
    weight = new List<double>();

    retainHistory = true;

    CalcWeights();
  }

  /// <summary>
  /// Provide the next sample to the differintegral
  /// <para>The resulting differintegral is returned</para>
  /// <para>
  ///   Providing inputs at intervals other than Dt
  ///   may invalidate results
  /// </para>
  /// </summary>
  /// <param name="value">next input sample</param>
  public double Compute(double value)
  {
    Insert(value);
    return Compute(history);
  }

  /// <summary>
  /// Return the differintegral for a provided history
  /// The provided values are not added to the internal history
  /// </summary>
  /// <param name="curve">list of history values</param>
  public double Compute(List<double> curve)
  {
    double sum = 0.0;
    int idx;
    int i;

    for(i = 0; i < curve.Count; i++)
    {
      idx = curve.Count - 1 - i;
      if(i > weight.Count - 1) CalcWeights();
      sum += curve[idx] * weight[i];
    }

    return sum / Math.Pow(dt, q);
  }

  /// <summary>
  /// Add a sample value to the history
  /// </summary>
  /// <param name="value">next input sample</param>
  public void Insert(double value)
  {
```

```
    if(historyLimit != 0 && history.Count >= historyLimit)
    {
      switch(behavior)
      {
        case OverflowBehavior.Ratio:
          double r;

          if(weight.Count < history.Count) CalcWeights();
          if(history.Count < 2)
            r = 0.99;
          else
            r = weight[history.Count - 1] / weight[history.Count - 2];

          history[1] += history[0] * r;
          break;

        case OverflowBehavior.Exponential:
          history[1] += history[0] * 0.99;
          break;

        case OverflowBehavior.Coopmans:
          if(weight.Count < history.Count) CalcWeights();
          value += history[0] * weight[history.Count - 1];
          break;

        default:  // OverflowBehavior.Truncate
          break;
      }

      history.RemoveAt(0);
    }

    history.Add(value);
  }

  /// <summary>
  /// Calculate and store the next uncomputed weight
  /// </summary>
  private void NextWeight()
  {
    int k = weight.Count;

    if(k == 0)
      weight.Add(1.0);
    else
      weight.Add(weight[k - 1] * (k - 1 - q) / k);
  }

  /// <summary>
  /// Calculate enough weights to differintegrate
  /// over the entire internal history
```

```
  /// </summary>
  public void CalcWeights()
  {
    CalcWeights(Math.Max(history.Count, historyLimit));
  }

  /// <summary>
  /// Calculate enough weights to differintegrate over
  /// a history of the specified length
  /// </summary>
  /// <param name="length">history length</param>
  public void CalcWeights(int length)
  {
    while(weight.Count < length)
      NextWeight();
  }
}
```

Listing B.1: Selections from FractionalDifferIntegrator.cs

Neural Network Code

```
/// <summary>
/// Neuron class
/// By: Sam Gardner
/// 10-31-2007
/// Modified: 02-15-2009
/// </summary>
public class Neuron : IEnumerable<Neuron>
{
  public const string DEFAULT_TYPE = "Neuron";
  public const double DEFAULT_VALUE = 0.0;
  public const string DEFAULT_ACTIVATION_FUNCTION = "Tanh";
  public const double DEFAULT_ACTIVATION_SLOPE = 1.0;
  public const double DEFAULT_MAX_ACTIVATION_VALUE = 1.0;
  public const double DEFAULT_MIN_ACTIVATION_VALUE = -1.0;


  protected List<double> inWeight; // Weights of incoming links
  protected List<Neuron> inLink;   // Incoming links from other nodes
  protected List<Neuron> outLink;  // Outgoing links to other nodes

  protected string actFunc;        // Activation function type name
  protected double actSlope;       // Activation slope
  protected double maxActValue;    // Maximum activation output value
  protected double minActValue;    // Minimum activation output value

  protected double value;          // Value of neuron
```

```
protected double propValue;       // Propagation value
protected double auxValue;        // Auxillary computation value
protected double output;          // Output value

protected int inLayer;           // Layer index from input
protected int outLayer;          // Layer index from output

protected bool computed;          // Computed flag
protected bool visited;           // Visited flag


/// <summary>
/// Create a new neuron
/// </summary>
public Neuron()
{
  inWeight = new List<double>();
  inLink = new List<Neuron>();
  outLink = new List<Neuron>();

  actFunc = DEFAULT_ACTIVATION_FUNCTION;
  actSlope = DEFAULT_ACTIVATION_SLOPE;
  maxActValue = DEFAULT_MAX_ACTIVATION_VALUE;
  minActValue = DEFAULT_MIN_ACTIVATION_VALUE;

  value = DEFAULT_VALUE;
  propValue = 0.0;
  auxValue = 0.0;
  output = double.NaN;

  inLayer = -1;
  outLayer = -1;

  computed = false;
  visited = false;
}

/// <summary>
/// Traverse this node setting visited flags
/// </summary>
public virtual void Traverse()
{
  if(!visited)
  {
    visited = true;

    foreach(Neuron i in inLink)
      i.Traverse();
  }
}
```

```
/// <summary>
/// Initialize this neuron between uses
/// <para>
///   Intended to clear state memory and other features which
///   should not be carried over between problems
/// </para>
/// </summary>
public virtual void Initialize()
{
  propValue = 0.0;
  auxValue = 0.0;
  output = double.NaN;

  Reset();
}

/// <summary>
/// Reset this neuron for another computation
/// </summary>
public void Reset()
{
  computed = false;
  visited = false;
}

/// <summary>
/// Return or compute the output value of this neuron
/// <para>
///   Designed for recursive computation starting at network outputs
/// </para>
/// </summary>
/// <returns>neuron output</returns>
public virtual double Compute()
{
  if(!computed)
  {
    if(visited)
      return PrematureOutput();

    visited = true;

    // Don't adjust the value of input and bias nodes
    if(inLink.Count <= 0)
    {
      propValue = value;
      auxValue = value;
      output = value;
    }
    else
    {
      propValue = Propagation();
```

```
      auxValue = Auxillary(propValue);
      output = Activation(auxValue);
    }

    computed = true;
  }

  return output;
}

/// <summary>
/// Return an output value prematurely
/// <para>
///   This is required when a cycle is encountered in the network
///   Some initial value is required to unwrap the cycle
/// </para>
/// </summary>
/// <returns>a premature output for this neuron</returns>
public virtual double PrematureOutput()
{
  if(inLink.Count <= 0)
  {
    propValue = value;
    auxValue = value;
    output = value;
  }
  else if(double.IsNaN(output))
  {
    propValue = value;
    auxValue = Auxillary(propValue);
    output = Activation(auxValue);
  }

  return output;
}

/// <summary>
/// Compute the propagation value for this neuron
/// <para>Input and Bias nodes return 0.0</para>
/// </summary>
public virtual double Propagation()
{
  // Compute weighted sum of inputs
  double net = 0.0;
  for(int n = 0; n < inLink.Count; n++)
    net += inLink[n].Compute() * inWeight[n];

  return net;
}

/// <summary>
```

```
  /// Run a value through this neuron's auxillary computation
  /// <para>Default is an identity function (pass-through)</para>
  /// </summary>
  /// <param name="input">computation input</param>
  /// <returns>computation output</returns>
  public virtual double Auxillary(double input)
  {
    return input;
  }

  /// <summary>
  /// Run a value through this neuron's activation function
  /// </summary>
  /// <param name="input">activation function input</param>
  /// <returns>activation function output</returns>
  public virtual double Activation(double input)
  {
    double midpoint, range, halfRange, output;
    switch(actFunc.Trim().ToLower())
    {
      case "binary":  // actSlope selects threshold value
        return (input >= actSlope) ? maxActValue : minActValue;
      case "linear":
        midpoint = (maxActValue + minActValue) / 2.0;
        output = (actSlope * input) + midpoint;
        if(output < minActValue) output = minActValue;
        if(output > maxActValue) output = maxActValue;
        return output;
      case "sigmoid":
        range = maxActValue - minActValue;
        return (range / (1.0 + Math.Exp(-actSlope * input)))
               + minActValue;
      case "tanh":
        halfRange = (maxActValue - minActValue) / 2.0;
        midpoint = (maxActValue + minActValue) / 2.0;
        return (halfRange * Math.Tanh(actSlope * input)) + midpoint;
    }

    // unrecognized activation function results in identity function
    return input;
  }
}
```

Listing B.2: Selections from Neuron.cs

```
/// <summary>
/// FractionalNeuron class
/// By: Sam Gardner
/// 10-31-2007
/// Modified: 10-22-2008
/// </summary>
public class FractionalNeuron : Neuron
{
  public const double DEFAULT_Q = 0.0;
  public const double DEFAULT_DT = 1.0;
  public const int DEFAULT_HISTORY_LIMIT = 0;
  public const FractionalDifferIntegrator.OverflowBehavior
               DEFAULT_OVERFLOW_BEHAVIOR
                = FractionalDifferIntegrator.OverflowBehavior.Ratio;

  protected FractionalDifferIntegrator frac;

  /// <summary>
  /// Create a new fractional neuron
  /// </summary>
  public FractionalNeuron() : base()
  {
    frac = new FractionalDifferIntegrator();

    frac.Q = DEFAULT_Q;
    frac.Dt = DEFAULT_DT;
    frac.HistoryLimit = DEFAULT_HISTORY_LIMIT;
    frac.Behavior = DEFAULT_OVERFLOW_BEHAVIOR;
  }

  /// <summary>
  /// Initialize the fractional neuron by clearing its history
  /// </summary>
  public override void Initialize()
  {
    frac.ClearHistory();
  }

  /// <summary>
  /// Return result of fractional differintegration on input value
  /// </summary>
  /// <param name="input">computation input</param>
  public override double Auxillary(double input)
  {
    return frac.Compute(input);
  }
}
```

Listing B.3: Selections from FractionalNeuron.cs

```
/// <summary>
/// NeuralNetwork class
/// By: Sam Gardner
/// 10-31-2007
/// Modified: 12-16-2008
/// </summary>
public class NeuralNetwork : IEnumerable<Neuron>
{
  protected RandomGenerator rand;

  protected List<Neuron> node;          // All neurons in network
  protected List<Neuron> inputNode;     // All input neurons in network
  protected List<Neuron> outputNode;    // All output neurons in network


  /// <summary>
  /// Create a new neural network
  /// </summary>
  public NeuralNetwork()
  {
    rand = new RandomGenerator();

    node = new List<Neuron>();
    inputNode = new List<Neuron>();
    outputNode = new List<Neuron>();
  }

  /// <summary>
  /// Create a neural network by copying another
  /// </summary>
  /// <param name="nn">neural network to copy</param>
  public NeuralNetwork(NeuralNetwork nn)
  {
    int index;

    rand = nn.rand.Copy();

    node = new List<Neuron>();
    inputNode = new List<Neuron>();
    outputNode = new List<Neuron>();


    foreach(Neuron n in nn.node)
      node.Add(n.Copy());

    for(int n = 0; n < nn.node.Count; n++)
      for(int i = 0; i < nn.node[n].InputCount; i++)
      {
        index = nn.node.IndexOf(nn.node[n].Input(i));
        node[n].AddInput(node[index], nn.node[n].Weight(i));
```

```
    }

  foreach(Neuron n in nn.inputNode)
  {
    index = nn.node.IndexOf(n);
    inputNode.Add(node[index]);
  }
  foreach(Neuron n in nn.outputNode)
  {
    index = nn.node.IndexOf(n);
    outputNode.Add(node[index]);
  }
}

/// <summary>
/// Traverse the network recursively setting visited flag for each
/// node in the computation path
/// </summary>
public virtual void Traverse()
{
  Reset();
  foreach(Neuron o in outputNode)
    o.Traverse();
}

/// <summary>
/// Traverse the network setting input layer values on all nodes
/// <para>
///   layers are numbered from 0 starting at the input layer
/// </para>
/// </summary>
public virtual void ForwardLayerId()
{
  List<Neuron> searched = new List<Neuron>();
  List<Neuron> discovered = new List<Neuron>();
  Neuron n;

  foreach(Neuron i in inputNode)
  {
    i.InLayer = 0;
    discovered.Add(i);
  }

  while(discovered.Count > 0)
  {
    n = discovered[0];
    discovered.RemoveAt(0);

    foreach(Neuron o in n.Outputs)
    {
      o.InLayer = n.InLayer + 1;
```

```
        if(!searched.Contains(o))
          discovered.Add(o);
      }

      searched.Add(n);
    }
  }

  /// <summary>
  /// Simplify neural network by removing unused nodes
  /// </summary>
  public virtual void Simplify()
  {
    int i = 0;

    Traverse();

    // Prune unused nodes
    while(i < node.Count)
    {
      if(!node[i].Visited && !inputNode.Contains(node[i]) &&
                             !outputNode.Contains(node[i]))
        RemoveAt(i);
      else
        i++;
    }
  }

  /// <summary>
  /// Initialize entire neural network to birth condition
  /// <para>
  ///   Intended to reset state memory and other stored values
  ///   between problems
  /// </para>
  /// </summary>
  public virtual void Initialize()
  {
    foreach(Neuron n in node)
      n.Initialize();
  }

  /// <summary>
  /// Reset neural network node status for another computation
  /// </summary>
  protected virtual void Reset()
  {
    foreach(Neuron n in node)
      n.Reset();
  }

  /// <summary>
```

```
/// Set input nodes to provided input values
/// </summary>
/// <param name="input">list of input nodes</param>
public virtual void SetInputs(List<double> input)
{
  if(input.Count < inputNode.Count)
    throw new Exception("Not enough inputs provided");
  if(input.Count > inputNode.Count)
    throw new Exception("Too many inputs provided");

  for(int i = 0; i < inputNode.Count; i++)
    inputNode[i].Value = input[i];
}

/// <summary>
/// Compute outputs of neural network based on provided inputs
/// </summary>
/// <param name="input">list of input values</param>
/// <return>list of output values</return>
public virtual List<double> Compute(List<double> input)
{
  List<double> output = new List<double>();

  SetInputs(input);
  Reset();

  // Compute output
  foreach(Neuron o in outputNode)
    output.Add(o.Compute());

  return output;
}

/// <summary>
/// Create a feed forward multi layer neural network
/// using specified node type
/// <para>Builds layers with the given node counts in order</para>
/// <para>The first layer is all inputs</para>
/// <para>The last layer is all outputs</para>
/// <para>Bias nodes connect to all layers but the input</para>
/// <para>All weights are initialized to 0.0</para>
/// </summary>
/// <param name="prototype">node to copy as template</param>
/// <param name="layer">list of layer sizes in order</param>
/// <param name="multiBias">selects one bias node per layer</param>
public void BuildFFMultiLayer(Neuron prototype,
                              List<int> layer, bool multiBias)
{
  if(layer.Count < 2)
    throw new Exception("Not enough layers specified");
```

```
    List<List<Neuron>> layerSet = new List<List<Neuron>>();
    Neuron newNode;

    Clear();

    for(int l = 0; l < layer.Count; l++)
    {
      layerSet.Add(new List<Neuron>());
      for(int i = 0; i < layer[l]; i++)
      {
        // Bias nodes
        if(multiBias || l == 0)
          if(i == 0 && l < layer.Count - 1)
          {
            newNode = prototype.Copy();
            newNode.Value = 1.0;
            layerSet[l].Add(newNode);
          }

        newNode = prototype.Copy();
        layerSet[l].Add(newNode);

        // Links
        if(l > 0)
        {
          if(!multiBias && l > 1)
            newNode.AddInput(layerSet[0][0], 0.0);

          foreach(Neuron prevNode in layerSet[l - 1])
            newNode.AddInput(prevNode, 0.0);
        }
      }
    }

    // Add all nodes to network structure
    int n = 0;
    for(int l = 0; l < layerSet.Count; l++)
      for(int i = 0; i < layerSet[l].Count; i++)
      {
        node.Add(layerSet[l][i]);
        if(l == 0 && i != 0) SetInput(n);
        if(l == layerSet.Count - 1) SetOutput(n);
        n++;
      }
  }

  /// <summary>
  /// Mirror all links in network
  /// <para>New links copy the weight of their mirror</para>
  /// <para>Pre-existing mirrored link weights are not changed</para>
  /// </summary>
```

```
public void MirrorLinks()
{
  foreach(Neuron n in node)
    for(int o = 0; o < n.OutputCount; o++)
      if(!n.Output(o).ConnectedTo(n))
        Connect(n.Output(o), n, n.OutputWeight(o));
}

/// <summary>
/// Set all weights in the network to random values with specified
/// uniform distribution
/// </summary>
/// <param name="min">minimum extent of distribution</param>
/// <param name="max">maximum extent of distribution</param>
public void RandomUniformWeights(double min, double max)
{
  foreach(Neuron n in node)
    for(int o = 0; o < n.OutputCount; o++)
      n.SetOutputWeight(o, rand.NextDouble(min, max));
}

/// <summary>
/// Set all weights in the network to random values with specified
/// gaussian distribution
/// </summary>
/// <param name="mean">mean of gaussian</param>
/// <param name="stdDev">standard deviation of gaussian</param>
public void RandomGaussianWeights(double mean, double stdDev)
{
  foreach(Neuron n in node)
    for(int o = 0; o < n.OutputCount; o++)
      n.SetOutputWeight(o, rand.NextGaussianDouble(mean, stdDev));
}

/// <summary>
/// Add an existing neuron to the network
/// </summary>
/// <param name="n">the neuron to add</param>
public void Add(Neuron n)
{
  node.Add(n);
}

/// <summary>
/// Add an existing neuron to the network as an input node
/// </summary>
public void AddInput(Neuron n)
{
  inputNode.Add(n);
  node.Add(n);
}
```

```
  /// <summary>
  /// Add an existing neuron to the network as an output node
  /// </summary>
  public void AddOutput(Neuron n)
  {
    outputNode.Add(n);
    node.Add(n);
  }

  /// <summary>
  /// Completely removes a neuron from the network
  /// <para>any links to or from the neuron are also destroyed</para>
  /// </summary>
  /// <param name="n">the neuron to remove</param>
  public void Remove(Neuron n)
  {
    n.Clear();
    if(ContainsInput(n)) inputNode.Remove(n);
    if(ContainsOutput(n)) outputNode.Remove(n);
    node.Remove(n);
  }

  /// <summary>
  /// Connnect two neurons in the network by index
  /// with specified weight
  /// </summary>
  /// <param name="one">index of neuron to link from</param>
  /// <param name="two">index of neuron to link to</param>
  /// <param name="weight">link weight</param>
  public void Connect(int one, int two, double weight)
  {
    if(one >= 0 && one < node.Count)
      if(two >= 0 && two < node.Count)
        node[one].AddOutput(node[two], weight);
  }

  /// <summary>
  /// Disconnect two neurons in the network by index
  /// <para>all links from neuron one to neuron two are removed</para>
  /// </summary>
  /// <param name="one">index of source neuron</param>
  /// <param name="two">index of destination neuron</param>
  public void Disconnect(int one, int two)
  {
    node[one].RemoveOutput(node[two]);
  }
}
```

Listing B.4: Selections from NeuralNetwork.cs

Evolutionary Algorithm Code

```
/// <summary>
/// Member class
/// By: Sam Gardner
/// 11-26-2007
/// Modified: 12-16-2008
/// </summary>
public class Member : IComparable<Member>
{
  protected RandomGenerator rand;

  protected List<string> comment;

  protected double maxPossibleFitness;
  protected double fitness;


  /// <summary>
  /// Create a new member
  /// </summary>
  public Member()
  {
    rand = new RandomGenerator();

    comment = new List<string>();

    maxPossibleFitness = 100.0;
    fitness = 0.0;
  }

  /// <summary>
  /// Get member's temperature (inverse of fitness)
  /// </summary>
  public virtual double Temperature
  {
    get { return 1.0 - (fitness / maxPossibleFitness); }
  }

  /// <summary>
  /// Get member's instantaneous temperature (random variation)
  /// </summary>
  public virtual double InstantaneousTemperature
  {
    get
    {
      double frac = rand.NextDouble();
      return frac * Temperature;
    }
```

```
  }

  /// <summary>
  /// Return a mutated copy of this member
  /// </summary>
  public virtual Member Mutate()
  {
    return this.Copy();
  }

  /// <summary>
  /// Return a copy of this member crossed with another
  /// </summary>
  /// <param name="other">father member</param>
  public virtual Member Crossover(Member other)
  {
    return this.Copy();
  }

  /// <summary>
  /// Return fitness score of this member
  /// </summary>
  public virtual double Evaluate()
  {
    return 0.0;
  }


  /// <summary>
  /// Return a random number in the range [min, max]
  /// tempered by the instantaneous temperature
  /// </summary>
  /// <param name="min">minimum number</param>
  /// <param name="max">maximum number</param>
  /// <param name="temp">instantaneous temperature</param>
  public int GNARLModifications(int min, int max, double temp)
  {
    double u = rand.NextDouble();
    return min + (int)Math.Floor(u * temp * (double)(max - min));
  }
}
```

Listing B.5: Selections from Member.cs

```
/// <summary>
/// Population class
/// By: Sam Gardner
/// 11-26-2007
/// Modified: 12-02-2008
/// </summary>
public class Population : IEnumerable<Member>
{
  protected RandomGenerator rand;

  protected EvolutionaryAlgorithm ea;  // reference to parent EA

  protected List<string> comment;      // comment for disk files

  protected string name;               // friendly name of population
  protected string use;                // population use type name

  protected Member prototype;          // prototype population member
  protected List<Member> member;       // population member list

  protected int size;                  // population size
  protected int elite;                 // number of elite members
  protected int refresh;               // number of fresh members


  /// <summary>
  /// Create a new population
  /// </summary>
  public Population()
  {
    rand = new RandomGenerator();

    ea = null;

    comment = new List<string>();

    name = "";
    use = "Default";

    prototype = null;
    member = new List<Member>();

    size = 0;
    elite = 0;
    refresh = 0;
  }

  /// <summary>
  /// Add a member to the population
  /// <para>type must match the current prototype</para>
```

```
  /// </summary>
  public virtual void Add(Member m)
  {
    if(m.Type == prototype.Type)
      member.Add(m.Copy());
  }

  /// <summary>
  /// Return a freshly initialized population member
  /// <para>null if prototype is null</para>
  /// </summary>
  public virtual Member Initialized()
  {
    if(prototype != null)
      if(ea != null)
        return ea.InitializeMember(prototype,
                                   ea.IndexOfPopulation(this), -1);
      else
        return prototype.Copy().Initialize();

    return null;
  }

  /// <summary>
  /// Return the next generation's population
  /// </summary>
  public virtual void Next()
  {
    List<Member> next = new List<Member>();

    next.AddRange(EliteList());
    next.AddRange(InitializedList(refresh));

    while(next.Count < size)
    {
      next.Add(Initialized());
    }

    member = next;
  }
}
```

Listing B.6: Selections from Population.cs

```
/// <summary>
/// EvolutionaryAlgorithm class
/// By: Sam Gardner
/// 11-24-2007
/// Modified: 12-16-2008
/// </summary>
public class EvolutionaryAlgorithm
{
  protected string name;

  protected List<string> comment;

  protected List<Population> population;

  protected int generation;
  protected bool initialized;
  protected bool finished;

  protected double maxPossibleFitness;

  // Stopping Conditions
  protected double targetFitness;
  protected int targetGeneration;

  // Reference to log and current state (not copied)
  protected EvolutionaryLog log;
  protected EvolutionaryState state;

  // Logging preferences
  protected bool saveGenerationalBestMembers;
  protected bool saveBestMembers;

  protected bool saveAllPopulations;
  protected bool saveFinalPopulations;

  protected bool saveMemberStats;
  protected bool savePopulationStats;
  protected bool saveGenerationStats;


  /// <summary>
  /// Create a new evolutionary algorithm with no populations
  /// </summary>
  public EvolutionaryAlgorithm()
  {
    name = "Evolution";

    comment = new List<string>();

    population = new List<Population>();
```

```
    generation = 0;
    initialized = false;
    finished = false;

    maxPossibleFitness = 100.0;

    targetFitness = 0.0;
    targetGeneration = 0;

    log = null;
    state = null;

    saveGenerationalBestMembers = false;
    saveBestMembers = true;

    saveAllPopulations = false;
    saveFinalPopulations = true;

    saveMemberStats = true;
    savePopulationStats = true;
    saveGenerationStats = true;
  }

  /// <summary>
  /// Initialize the ea
  /// </summary>
  public virtual void Initialize()
  {
    for(int p = 0; p < population.Count; p++)
      InitializePopulation(population[p], p);
  }

  /// <summary>
  /// Evaluate a population member
  /// </summary>
  /// <param name="m">member to evaluate</param>
  /// <param name="popIndex">population index</param>
  /// <param name="memberIndex">member index</param>
  /// <returns>fitness</returns>
  public virtual double EvaluateMember(Member m, int popIndex,
                                       int memberIndex)
  {
    return m.Evaluate();
  }

  /// <summary>
  /// Evaluate a population
  /// </summary>
  /// <param name="p">population to evaluate</param>
  /// <param name="popIndex">population index</param>
```

```
public virtual void EvaluatePopulation(Population p, int popIndex)
{
  for(int m = 0; m < p.Count; m++)
    p[m].Fitness = EvaluateMember(p[m], popIndex, m);
}

/// <summary>
/// Evaluate current generation
/// </summary>
public virtual void Evaluate()
{
  for(int p = 0; p < population.Count; p++)
    EvaluatePopulation(population[p], p);
}

/// <summary>
/// Adjust a population member to obtain a different one
/// </summary>
/// <param name="m">member to mutate</param>
public virtual Member Mutate(Member m)
{
  return m.Mutate();
}

/// <summary>
/// Cross two population members to obtain a new one related to both
/// </summary>
/// <param name="father">father member</param>
/// <param name="mother">mother member</param>
public virtual Member Crossover(Member father, Member mother)
{
  return mother.Crossover(father);
}

/// <summary>
/// Generate next generation
/// </summary>
protected virtual void Next()
{
  foreach(Population p in population)
    p.Next();
}

/// <summary>
/// Reset ea for a new evolution
/// </summary>
public void Reset()
{
  if(log != null) CreateNextState();

  generation = 0;
```

```
    finished = false;

    Initialize();

    if(log != null) log.Initialize();

    Evaluate();
    if(log != null) log.LogCurrentState(this);

    initialized = true;
  }

  /// <summary>
  /// Run the ea for one generation
  /// </summary>
  public void RunStep()
  {
    if(!finished)
    {
      if(!initialized)
      {
        Reset();
      }
      else
      {
        if(log != null) CreateNextState();
        generation++;
        Next();
        Evaluate();
        if(log != null) log.LogCurrentState(this);
      }

      // Check Stopping Conditions
      if(targetFitness > 0.0 && CurrentMaxFitness() >= targetFitness)
        finished = true;

      if(targetGeneration > 0 && generation >= targetGeneration)
        finished = true;
    }
  }
}
```

Listing B.7: Selections from EvolutionaryAlgorithm.cs

Population Code

```
/// <summary>
/// GNARLPopulation class
/// By: Sam Gardner
/// 10-19-2008
/// Modified: 12-02-2008
/// </summary>
public class GNARLPopulation : Population
{
  protected int survive; //Number of surviving members (top fitness)


  /// <summary>
  /// Create a new GNARL population
  /// </summary>
  public GNARLPopulation() : base()
  {
    survive = 0;
  }

  /// <summary>
  /// Replace the population with the next generation
  /// </summary>
  public override void Next()
  {
    List<Member> next = new List<Member>();
    int index;

    next.AddRange(EliteList());
    next.AddRange(InitializedList(refresh));

    while(next.Count < size)
    {
      if(survive > 0)
      {
        index = rand.NextInt(survive);
        next.Add(Mutate(member[index]));
      }
      else
      {
        next.Add(Initialized());
      }
    }

    member = next;
  }
}
```

Listing B.8: Selections from GNARLPopulation.cs

```
/// <summary>
/// GAPopulation class
/// By: Sam Gardner
/// 10-22-2008
/// Modified: 12-16-2008
/// </summary>
public class GAPopulation : Population
{
  protected string selector;              //Selection algorithm type
  protected double tournamentThreshold; //Probability for fit select
  protected double mutationRate;          //Probability of mutation


  /// <summary>
  /// Create a new GA population
  /// </summary>
  public GAPopulation() : base()
  {
    selector = "Uniform";
    tournamentThreshold = 0.75;
    mutationRate = 0.10;
  }


  /// <summary>
  /// Select a population member to populate the next generation
  /// <para>Selection method depends on selector type name</para>
  /// <para>"Uniform" if unrecognized</para>
  /// </summary>
  protected virtual Member Select()
  {
    switch(selector.Trim().ToLower())
    {
      case "roulette": return SelectRoulette();
      case "tournament": return SelectTournament();
      default: return SelectUniform();
    }
  }

  /// <summary>
  /// Select a population member using roulette selection
  /// <para>null if selection logic is flawed</para>
  /// </summary>
  protected virtual Member SelectRoulette()
  {
    double total = 0.0;
    double mark, sum;

    Sort();
```

```
    foreach(Member m in member)
      total += m.Fitness;

    mark = rand.NextDouble(total);

    sum = 0.0;
    foreach(Member m in member)
    {
      sum += m.Fitness;

      if(sum >= mark)
        return m;
    }

    return null;
  }

  /// <summary>
  /// Select a population member using tournament selection
  /// </summary>
  protected virtual Member SelectTournament()
  {
    int one = rand.NextInt(member.Count);
    int two = rand.NextInt(member.Count);
    if(rand.Chance(tournamentThreshold))
      return (member[one].Fitness > member[two].Fitness)
              ? member[one] : member[two];

    return (member[one].Fitness > member[two].Fitness)
            ? member[two] : member[one];
  }

  /// <summary>
  /// Select a population member using uniform selection
  /// </summary>
  protected virtual Member SelectUniform()
  {
    int index = rand.NextInt(member.Count);

    return member[index];
  }


  /// <summary>
  /// Replace the population with the next generation
  /// </summary>
  public override void Next()
  {
    List<Member> next = new List<Member>();
    Member f, m, c;
```

```
    next.AddRange(EliteList());
    next.AddRange(InitializedList(refresh));

    while(next.Count < size)
    {
      f = Select();
      m = Select();
      c = Crossover(f, m);
      if(rand.Chance(mutationRate)) c = Mutate(c);
      next.Add(c);
    }

    member = next;
  }
}
```

Listing B.9: Selections from GAPopulation.cs

Fox/Rabbit Evolutionary Algorithm Code

```
/// <summary>
/// NeuralNetworkMember class
/// Population Member
/// By: Sam Gardner
/// 03-13-2008
/// Modified 02-15-2008
/// </summary>
public class NeuralNetworkMember : Member
{
  protected NeuralNetwork nn;


  // Initialization Preferences
  protected string initialTopology;
  protected List<int> initialLayerSizes;

  protected List<string> initialNodeTypeValues;
  protected List<string> initialActivationFunctionValues;
  protected List<string> initialBehaviorValues;

  protected int initialInputs;
  protected int initialOutputs;
  protected int initialBiases;
  protected int initialHiddenNodes;
  protected int initialLinks;

  protected bool initialGaussianWeights;
  protected bool initialGaussianValues;
```

```
protected bool initialGaussianActivationSlopes;
protected bool initialGaussianActivationLimits;
protected bool initialGaussianQs;
protected bool initialGaussianDts;
protected bool initialGaussianHLimits;

protected double initialWeight;
protected double initialWeightRange;
protected double initialValue;
protected double initialValueRange;
protected double initialActivationSlope;
protected double initialActivationSlopeRange;
protected double initialActivationMax;
protected double initialActivationMaxRange;
protected double initialActivationMin;
protected double initialActivationMinRange;

protected double initialQ;
protected double initialQRange;
protected double initialDt;
protected double initialDtRange;
protected int initialHLimit;
protected int initialHLimitRange;


// Mutation and Crossover Preferences
protected List<string> nodeTypeValues;
protected List<string> activationFunctionValues;
protected List<string> behaviorValues;

protected bool parametricMutation;
protected bool structuralMutation;
protected bool temperatureBasedMutation;


// Parametric Preferences
protected bool weightMutation;
protected bool valueMutation;
protected bool actFuncMutation;
protected bool actSlopeMutation;
protected bool actLimitMutation;
protected bool actCenterLock;
protected bool actRangeLock;
protected bool qMutation;
protected bool dtMutation;
protected bool hLimitMutation;
protected bool behaviorMutation;

protected double biasValue;

protected double weightScale;
```

```
protected double valueScale;
protected double actFuncChangeRate;
protected double actSlopeScale;
protected double actLimitScale;
protected double qScale;
protected double dtScale;
protected int hLimitScale;
protected double behaviorChangeRate;


// Structural Preferences
protected int maxNodeAdd;
protected int minNodeAdd;
protected int maxNodeDel;
protected int minNodeDel;
protected int maxNodeRetype;
protected int minNodeRetype;

protected bool addLinksWithNodes;

protected int maxLinkAdd;
protected int minLinkAdd;
protected int maxLinkDel;
protected int minLinkDel;

protected double linkIOBias;

protected bool allowSelfLinks;
protected bool allowDuplicateLinks;
protected bool allowMirrorLinks;
protected bool allowCrossLinks;
protected bool allowBackLinks;


/// <summary>
/// Create a new neural network member
/// </summary>
public NeuralNetworkMember() : base()
{
  nn = new NeuralNetwork();

  initialLayerSizes = new List<int>();

  initialNodeTypeValues = new List<string>();
  initialActivationFunctionValues = new List<string>();
  initialBehaviorValues = new List<string>();

  nodeTypeValues = new List<string>();
  activationFunctionValues = new List<string>();
  behaviorValues = new List<string>();
```

```
// Initialization Preferences
initialTopology = "FF";
initialLayerSizes.Add(10);

initialNodeTypeValues.AddRange(NodeTypeValueList);
initialActivationFunctionValues.Add("Tanh");
initialBehaviorValues.Add("Ratio");

initialInputs = 2;
initialOutputs = 2;
initialBiases = 1;
initialHiddenNodes = 10;
initialLinks = 42;

initialGaussianWeights = false;
initialGaussianValues = false;
initialGaussianActivationSlopes = false;
initialGaussianActivationLimits = false;
initialGaussianQs = false;
initialGaussianDts = false;
initialGaussianHLimits = false;

initialWeight = 0.0;
initialWeightRange = 0.5;
initialValue = 0.0;
initialValueRange = 3.0;
initialActivationSlope = 1.0;
initialActivationSlopeRange = 0.0;
initialActivationMax = 1.0;
initialActivationMaxRange = 0.0;
initialActivationMin = -1.0;
initialActivationMinRange = 0.0;

initialQ = 0.0;
initialQRange = 1.0;
initialDt = 1.0;
initialDtRange = 0.0;
initialHLimit = 0;
initialHLimitRange = 0;


// Mutation and Crossover Preferences
nodeTypeValues.AddRange(initialNodeTypeValues);
activationFunctionValues.AddRange(initialActivationFunctionValues);
behaviorValues.AddRange(initialBehaviorValues);

parametricMutation = true;
structuralMutation = true;
temperatureBasedMutation = true;

// Parametric Preferences
```

```
    weightMutation = true;
    valueMutation = true;
    actFuncMutation = false;
    actSlopeMutation = false;
    actLimitMutation = false;
    actCenterLock = false;
    actRangeLock = false;
    qMutation = true;
    dtMutation = false;
    hLimitMutation = false;
    behaviorMutation = false;

    biasValue = 1.0;

    weightScale = 1.0;
    valueScale = 10.0;
    actFuncChangeRate = 0.001;
    actSlopeScale = 0.1;
    actLimitScale = 0.1;
    qScale = 0.1;
    dtScale = 0.1;
    hLimitScale = 10;
    behaviorChangeRate = 0.001;


    // Structural Preferences
    maxNodeAdd = 3;
    minNodeAdd = 1;
    maxNodeDel = 3;
    minNodeDel = 1;
    maxNodeRetype = 3;
    minNodeRetype = 1;

    addLinksWithNodes = false;

    maxLinkAdd = 20;
    minLinkAdd = 4;
    maxLinkDel = 5;
    minLinkDel = 1;

    linkIOBias = 0.2;

    allowSelfLinks = false;
    allowDuplicateLinks = false;
    allowMirrorLinks = false;
    allowCrossLinks = false;
    allowBackLinks = false;
  }

  /// <summary>
  /// Initialize this neural network member
```

```
/// </summary>
public override Member Initialize()
{
  nn.Clear();

  switch(initialTopology.ToLower())
  {
    case "ff": InitializeFeedForward(); break;
    case "cc-ff": InitializeCrossConnectedFeedForward(); break;
    case "recurrent": InitializeRecurrent(); break;
    case "fc-ff": InitializeFullyConnectedFeedForward(); break;
    case "fc": InitializeFullyConnected(); break;
    default: /* random */ InitializeRandom(); break;
  }

  if(!double.IsNaN(biasValue)) SetAllBiasValues(biasValue);

  return this;
}

protected void InitializeRandom()
{
  Neuron node;
  List<int> from, to;
  int f, t;
  bool valid;

  for(int b = 0; b < initialBiases; b++)
  {
    node = InitialRandomNode();
    nn.Add(node);
  }

  for(int i = 0; i < initialInputs; i++)
  {
    node = InitialRandomNode();
    nn.AddInput(node);
  }

  for(int h = 0; h < initialHiddenNodes; h++)
  {
    node = InitialRandomNode();
    nn.Add(node);
  }

  for(int o = 0; o < initialOutputs; o++)
  {
    node = InitialRandomNode();
    nn.AddOutput(node);
  }
```

```
    // Links
    for(int l = 0; l < initialLinks; l++)
    {
      from = rand.Chance(linkIOBias)
             ? nn.InputNodeIndices() : nn.NonIONodeIndices();
      to = rand.Chance(linkIOBias)
           ? nn.OutputNodeIndices() : nn.NonIONodeIndices();

      if(from.Count > 0 && to.Count > 0)
      {
        from = RandomGenerator.Shuffle(from);
        to = RandomGenerator.Shuffle(to);

        valid = false;
        f = 0;
        t = -1;
        while(!valid)
        {
          t++; if(t >= to.Count) { t = 0; f++; }
          if(f >= from.Count) break;

          valid = LinkIsValid(from[f], to[t]);
        }

        if(valid)
          nn.Connect(from[f], to[t],
                     InitialRandomValue(initialGaussianWeights,
                                        initialWeight,
                                        initialWeightRange));
      }
    }
  }

  /// <summary>
  /// Return a mutated copy of this neural network member
  /// </summary>
  public override Member Mutate()
  {
    NeuralNetworkMember m = this.Copy() as NeuralNetworkMember;
    double temp = temperatureBasedMutation
                  ? m.InstantaneousTemperature : 1.0;

    if(m.parametricMutation) m.MutateParametric(temp);
    if(m.structuralMutation) m.MutateStructural(temp);

    if(!double.IsNaN(biasValue)) m.SetAllBiasValues(biasValue);

    return m;
  }

  /// <summary>
```

```
/// Parametric mutation
/// </summary>
/// <param name="temp">instantaneous temperature of member</param>
protected void MutateParametric(double temp)
{
  if(weightMutation) MutateWeights(temp);
  if(valueMutation) MutateValues(temp);
  if(actFuncMutation) MutateActivationFunctions(temp);
  if(actSlopeMutation) MutateActivationSlopes(temp);
  if(actLimitMutation) MutateActivationLimits(temp);

  if(qMutation) MutateQValues(temp);
  if(dtMutation) MutateDtValues(temp);
  if(hLimitMutation) MutateHistoryLimits(temp);
  if(behaviorMutation) MutateBehaviors(temp);
}

protected void MutateWeights(double temp)
{
  double value;

  for(int l = 0; l < nn.LinkCount; l++)
  {
    value = nn.Weight(l);
    value += GNARLPreturbation(weightScale, temp);
    nn.SetWeight(l, value);
  }
}

/// <summary>
/// Return true if link between specified indices would be valid
/// according to current settings
/// </summary>
/// <param name="from">source node index</param>
/// <param name="to">destination node index</param>
protected bool LinkIsValid(int from, int to)
{
  if(!allowSelfLinks && from == to) return false;
  if(!allowDuplicateLinks && nn.ConnectedTo(from, to)) return false;
  if(!allowMirrorLinks && nn.ConnectedTo(to, from)) return false;
  if(!allowCrossLinks && nn.IsCrossLink(from, to)) return false;
  if(!allowBackLinks && nn.IsBackLink(from, to)) return false;

  return true;
}

/// <summary>
/// Structural mutation
/// </summary>
/// <param name="temp">instantaneous temperature of member</param>
protected void MutateStructural(double temp)
```

```
{
  MutateNodeAdd(temp);
  MutateNodeDel(temp);
  MutateNodeRetype(temp);

  MutateLinkAdd(temp);
  MutateLinkDel(temp);
}

protected void MutateNodeDel(double temp)
{
  int num = GNARLModifications(minNodeDel, maxNodeDel, temp);
  List<int> indices;

  for(int i = 0; i < num; i++)
  {
    indices = nn.NonIONodeIndices();
    if(indices.Count > 0)
      nn.RemoveAt(indices[rand.NextInt(indices.Count)]);
  }
}

/// <summary>
/// Return a copy of this neural network member crossed with another
/// </summary>
/// <param name="other">father member</param>
public override Member Crossover(Member other)
{
  NeuralNetworkMember c = this.ShallowCopy() as NeuralNetworkMember;
  if(other is NeuralNetworkMember)
  {
    NeuralNetworkMember father = other as NeuralNetworkMember;
    NeuralNetworkMember mother = this;
    List<int> ChildToMother = new List<int>();
    List<int> ChildToFather = new List<int>();
    List<int> MotherToChild = new List<int>();
    List<int> FatherToChild = new List<int>();

    List<int> mHidden, fHidden, cHidden;
    bool inheritFromMother;
    int nodes, links;
    int mlink, flink;
    int mi, fi;
    int mdst, fdst;
    int i;


    // Inputs
    nodes = Math.Max(mother.nn.InputCount, father.nn.InputCount);
    for(i = 0; i < nodes; i++)
      if(rand.Chance(0.5))
```

```
    {
      if(i < mother.nn.InputCount)
        c.nn.AddInput(mother.nn.InputNode(i).Copy());
    }
    else
    {
      if(i < father.nn.InputCount)
        c.nn.AddInput(father.nn.InputNode(i).Copy());
    }

  // Outputs
  nodes = Math.Max(mother.nn.OutputCount, father.nn.OutputCount);
  for(i = 0; i < nodes; i++)
    if(rand.Chance(0.5))
    {
      if(i < mother.nn.OutputCount)
        c.nn.AddOutput(mother.nn.OutputNode(i).Copy());
    }
    else
    {
      if(i < father.nn.OutputCount)
        c.nn.AddOutput(father.nn.OutputNode(i).Copy());
    }

  // Non-IO Nodes
  mHidden = mother.nn.NonIONodeIndices();
  fHidden = father.nn.NonIONodeIndices();
  nodes = Math.Max(mHidden.Count, fHidden.Count);
  for(i = 0; i < nodes; i++)
    if(rand.Chance(0.5))
    {
      if(i < mHidden.Count)
        c.nn.Add(mother.nn.Node(mHidden[i]).Copy());
    }
    else
    {
      if(i < fHidden.Count)
        c.nn.Add(father.nn.Node(fHidden[i]).Copy());
    }


  // Build index maps
  for(i = 0; i < c.nn.InputCount; i++)
  {
    ChildToMother.Add((i < mother.nn.InputCount)
        ? mother.nn.IndexOf(mother.nn.InputNode(i)) : -1);
    ChildToFather.Add((i < father.nn.InputCount)
        ? father.nn.IndexOf(father.nn.InputNode(i)) : -1);
  }

  for(i = 0; i < c.nn.OutputCount; i++)
```

```
{
  ChildToMother.Add((i < mother.nn.OutputCount)
       ? mother.nn.IndexOf(mother.nn.OutputNode(i)) : -1);
  ChildToFather.Add((i < father.nn.OutputCount)
       ? father.nn.IndexOf(father.nn.OutputNode(i)) : -1);
}

cHidden = c.nn.NonIONodeIndices();
for(i = 0; i < cHidden.Count; i++)
{
  ChildToMother.Add((i < mHidden.Count) ? mHidden[i] : -1);
  ChildToFather.Add((i < fHidden.Count) ? fHidden[i] : -1);
}

for(i = 0; i < mother.nn.NodeCount; i++)
  MotherToChild.Add(ChildToMother.IndexOf(i));

for(i = 0; i < father.nn.NodeCount; i++)
  FatherToChild.Add(ChildToFather.IndexOf(i));


// Links
for(int n = 0; n < c.nn.NodeCount; n++)
{
  mi = ChildToMother[n];
  fi = ChildToFather[n];
  mlink = (mi != -1) ? mother.nn.Node(mi).OutputCount : 0;
  flink = (fi != -1) ? father.nn.Node(fi).OutputCount : 0;
  links = Math.Max(mlink, flink);
  for(i = 0; i < links; i++)
  {
    mdst = (i < mlink)
           ? MotherToChild[mother.nn.IndexOf(
                           mother.nn.Node(mi).Output(i))]
           : int.MinValue;
    fdst = (i < flink)
           ? FatherToChild[father.nn.IndexOf(
                           father.nn.Node(fi).Output(i))]
           : int.MinValue;

    inheritFromMother = false;
    if(rand.Chance(0.5)) inheritFromMother = true;
    if(mdst == -1) inheritFromMother = false;
    if(fdst == -1) inheritFromMother = true;

    if(inheritFromMother)
    {
      if(i < mlink)
        c.nn.Connect(n, mdst, mother.nn.Node(mi).OutputWeight(i));
    }
    else
```

```
            {
              if(i < flink)
                c.nn.Connect(n, fdst, father.nn.Node(fi).OutputWeight(i));
            }
          }
        }
      }

      if(!double.IsNaN(biasValue)) c.SetAllBiasValues(biasValue);

      return c;
    }
}
```

Listing B.10: Selections from NeuralNetworkMember.cs

```
/// <summary>
/// FoxRabbitEA class
/// Evolutionary Algorithm
/// By: Sam Gardner
/// 05-23-2008
/// Modified: 12-04-2008
/// </summary>
public class FoxRabbitEA : EvolutionaryAlgorithm
{
  protected Simulation sim;

  protected int trials;

  protected bool saveEveryReplay;
  protected bool saveGenerationalBestReplays;
  protected bool saveBestReplays;


  /// <summary>
  /// Create a new fox rabbit EA
  /// </summary>
  public FoxRabbitEA() : base()
  {
    sim = new Simulation();

    trials = 30;

    saveEveryReplay = false;
    saveGenerationalBestReplays = false;
    saveBestReplays = true;
  }

  /// <summary>
```

```
/// Return a new population suitable for use with the ea
/// </summary>
/// <param name="type">population type name</param>
/// <param name="useId">population use type name</param>
public override Population NewPopulation(string type, string use)
{
  int index = population.Count;
  Population pop = MakePopulation(type);
  NeuralNetworkMember prototype = new NeuralNetworkMember();
  prototype.MaxPossibleFitness = maxPossibleFitness;

  prototype.InitialBiases = 1;
  prototype.InitialInputs = 24;
  prototype.InitialOutputs = 2;
  prototype.InitialActivationFunctionValues.Clear();
  prototype.InitialActivationFunctionValues.Add("Tanh");
  prototype.InitialGaussianActivationSlopes = false;
  prototype.InitialActivationSlope = 1;
  prototype.InitialActivationSlopeRange = 0;
  prototype.InitialGaussianActivationLimits = false;
  prototype.InitialActivationMaxRange = 0;
  prototype.InitialActivationMinRange = 0;

  prototype.InitialBehaviorValues.Clear();
  prototype.InitialBehaviorValues.Add("Ratio");

  prototype.ActivationFunctionValues.Clear();
  prototype.ActivationFunctionValues.AddRange(
                          prototype.InitialActivationFunctionValues);

  prototype.BehaviorValues.Clear();
  prototype.BehaviorValues.AddRange(prototype.InitialBehaviorValues);

  prototype.StructuralMutation = false;
  prototype.AllowBackLinks = true;
  prototype.AllowDuplicateLinks = false;
  prototype.AllowCrossLinks = true;
  prototype.AllowMirrorLinks = true;
  prototype.AllowSelfLinks = true;


  pop.Prototype = prototype;
  pop.Name = string.Format("Fox {0}", index);
  pop.Use = "Default";
  pop.Size = 64;
  pop.Elite = 3;

  if(pop is GNARLPopulation)
  {
    (pop as GNARLPopulation).SurvivalRate = 0.5;
  }
```

```
    if(pop is GAPopulation)
    {
      (pop as GAPopulation).MutationRate = 0.01;
      (pop as GAPopulation).Selector = "Roulette";
      (pop as GAPopulation).TournamentThreshold = 0.75;
    }

    return pop;
  }

  /// <summary>
  /// Initialize the fox rabbit ea
  /// </summary>
  public override void Initialize()
  {
    base.Initialize();
    sim.Reset();
  }

  /// <summary>
  /// Evaluate a population member
  /// </summary>
  /// <param name="popIndex">population index</param>
  /// <param name="memberIndex">member index</param>
  /// <returns>fitness value</returns>
  public override double EvaluateMember(Member individual,
                                         int popIndex,
                                         int memberIndex)
  {
    if(individual is NeuralNetworkMember)
    {
      NeuralNetworkMember m = individual as NeuralNetworkMember;
      double fitness = 0.0;

      sim.FoxNN = m.NN.Copy();

      for(int t = 0; t < trials; t++)
      {
        sim.Reset();
        sim.Run();
        if(log != null)
        {
          sim.Replay.Generation = generation;
          sim.Replay.Population = population[popIndex].Name;
          sim.Replay.MemberIndex = memberIndex;
          sim.Replay.Trial = t;

          (state as FoxRabbitState).UpdateTrialLevelStatistics(
                              popIndex, memberIndex, sim.Replay);
```

```
            if(saveEveryReplay)
              sim.Replay.Save(
                string.Format("Replay-{0}-G{1}-{2}-M{3}-T{4}.replay",
                name,
                generation,
                population[popIndex].Name,
                memberIndex,
                t));

            if(saveGenerationalBestReplays)
              if((state as FoxRabbitState).IsGenerationalBestReplay(
                                                    popIndex, sim.Replay))
                sim.Replay.Save(
                  string.Format("BestReplay-{0}-{1}-G{2}.replay",
                  name,
                  population[popIndex].Name,
                  generation));

            if(saveBestReplays)
              if((log as FoxRabbitLog).IsBestReplay(
                                popIndex, sim.Replay))
                sim.Replay.Save(
                  string.Format("BestReplay-{0}-{1}.replay",
                  name,
                  population[popIndex].Name));
          }
          fitness += sim.Score();
        }
        fitness /= (double)trials;

        return fitness;
      }

      return 0.0;
    }
}
```

Listing B.11: Selections from FoxRabbitEA.cs

Fox/Rabbit Simulation Code

```
/// <summary>
/// Entity class
/// By: Sam Gardner
/// 05-25-2008
/// Modified: 12-16-2008
/// </summary>
public abstract class Entity
{
  protected RandomGenerator rand;

  protected List<string> comment;

  protected double radius;            // All entities are circular
  protected Vector2 position;
  protected Vector2 velocity;
  protected Vector2 acceleration;

  protected double translationScalar;
  protected Vector2 translation;
  protected bool solidCollision;

  protected Script script;

  protected bool startAnywhere;
  protected Vector2 start;
  protected Vector2 startRange;
  protected double startDistance;
  protected double startAzimuth;
  protected double startAzimuthRange;

  protected double maxSpeed;
  protected double maxAccel;
  protected double velocityDecayRate;


  /// <summary>
  /// Base constructor
  /// </summary>
  public Entity()
  {
    rand = new RandomGenerator();

    comment = new List<string>();

    radius = 0.5;
    position = new Vector2(0.0, 0.0);
    velocity = new Vector2(0.0, 0.0);
```

```
    acceleration = new Vector2(0.0, 0.0);

    translationScalar = 1.0;
    translation = new Vector2(0.0, 0.0);
    solidCollision = false;

    script = null;

    startAnywhere = true;
    start = new Vector2(0.0, 0.0);
    startRange = new Vector2(0.0, 0.0);
    startDistance = 0.0;
    startAzimuth = 0.0;
    startAzimuthRange = 180.0;

    maxSpeed = 8.0;
    maxAccel = 1.0;
    velocityDecayRate = 0.1;
  }

  /// <summary>
  /// Return a starting position that meets start preferences
  /// </summary>
  public Vector2 StartPosition()
  {
    Vector2 s = new Vector2();

    if(StartRelative)
    {
      double azimuth = startAzimuth;
      if(startAzimuthRange > 0.0)
        azimuth += rand.NextCenteredDouble(0.0, startAzimuthRange);
      azimuth = Angle.Rad(Angle.Azimuth2Deg(azimuth));

      s.X = start.X + ((startDistance + radius) * Math.Cos(azimuth));
      s.Y = start.Y + ((startDistance + radius) * Math.Sin(azimuth));
    }
    else
    {
      s.X = start.X;
      s.Y = start.Y;
      if(startRange.X > 0.0)
        s.X += rand.NextCenteredDouble(0.0, startRange.X);
      if(startRange.Y > 0.0)
        s.Y += rand.NextCenteredDouble(0.0, startRange.Y);
    }

    return s;
  }

  /// <summary>
```

```
/// Compute next action
/// <para>usually the next acceleration vector</para>
/// </summary>
/// <param name="sim">simulation to compute for</param>
public virtual void ComputeNextAction(Simulation sim)
{
  if(script != null)
    acceleration = script.Next();
  else
    Compute(sim);

  EnforceLimits();
}

/// <summary>
/// Perform any computations and adjustments necessary for movement
/// </summary>
/// <param name="sim">simulation to compute for</param>
protected virtual void Compute(Simulation sim)
{
  acceleration.Zero();
}

/// <summary>
/// Adjust acceleration and velocity to satisfy limits
/// </summary>
protected virtual void EnforceLimits()
{
  // Enforce maximum acceleration
  if(acceleration.Length() > maxAccel)
    acceleration = acceleration.Unit() * maxAccel;

  velocity += acceleration;

  // Enforce maximum velocity
  if(velocity.Length() > maxSpeed)
     velocity = velocity.Unit() * maxSpeed;

  translationScalar = 1.0;
}

/// <summary>
/// Return a vector from this entity's center to that of another
/// </summary>
public Vector2 CenterSeparation(Entity other)
{
  return other.position - position;
}

/// <summary>
/// Return a vector from the edge of this entity
```

```
/// to the edge of another
/// <para>takes radii into account</para>
/// </summary>
public Vector2 Separation(Entity other)
{
  Vector2 cs = CenterSeparation(other);
  return cs - (cs.Unit() * (radius + other.radius));
}

/// <summary>
/// Return fraction of translation before
/// this entity collides with another
/// <para>1.0 indicates no collision</para>
/// <para>
///   Uses single frame of reference technique
///   Described by: Joe van den Heuvel and Miles Jackson
///   For: Gamasutra
///   January 18, 2002
/// </para>
/// </summary>
public double Collision(Entity other)
{
  double t = 1.0;

  // Compute relative velocity using the frame of reference
  // where other is stationary
  Vector2 rv = velocity - other.velocity;
  double rvmag = rv.Length();

  Vector2 rp = position.To(other.position);
  double radii = radius + other.radius;

  double rp_dot_rv = rp.Dot(rv);
  if(rp_dot_rv > 0.0)          //Short circuit for wrong direction
    if(rvmag >= rp.Length() - radii) //sc for not enough distance
    {
      //length of rp projected onto rv
      double p = rp_dot_rv / rvmag;
      // square of closest distance entities get to each other
      double d = rp.LengthSq() - (p * p);
      if(d <= radii * radii)   //sc for doesn't pass close enough
      {
        double travel = p - Math.Sqrt(radii * radii - d);
        if(travel <= rvmag)    //sc for doesn't get there
          t = travel / rvmag;
      }
    }

  return t;
}
```

```
/// <summary>
/// Return fraction of translation before this entity
/// collides with a wall
/// <para>1.0 indicates no collision</para>
/// </summary>
/// <param name="sim">simulation to collide in</param>
public double CollisionWalls(Simulation sim)
{
  double collision = 1.0;

  if(sim.Field.Walls)
  {
    double xWall = sim.Field.Width / 2.0 - radius;
    double yWall = sim.Field.Height / 2.0 - radius;
    double t;

    if(velocity.X < 0.0)
    {
      t = (-xWall - position.X) / velocity.X;     // west wall
      if(t >= 0.0 && t < 1.0) collision = Math.Min(collision, t);
    }
    else if(velocity.X > 0.0)
    {
      t = (xWall - position.X) / velocity.X;      // east wall
      if(t >= 0.0 && t < 1.0) collision = Math.Min(collision, t);
    }

    if(velocity.Y < 0.0)
    {
      t = (-yWall - position.Y) / velocity.Y;     // north wall
      if(t >= 0.0 && t < 1.0) collision = Math.Min(collision, t);
    }
    else if(velocity.Y > 0.0)
    {
      t = (yWall - position.Y) / velocity.Y;      // south wall
      if(t >= 0.0 && t < 1.0) collision = Math.Min(collision, t);
    }
  }

  return collision;
}

/// <summary>
/// Check for entity-to-entity collision and set translation scalar
/// </summary>
/// <param name="other">entity to check collision with</param>
public void Collide(Entity other)
{
  double t = Collision(other);
  translationScalar = Math.Min(translationScalar, t);
  if(t < 1.0 && !(this is Critter && other is Critter))
```

```
      solidCollision = true;
    }

    /// <summary>
    /// Check for entity-to-wall collisions and set translation scalar
    /// </summary>
    /// <param name="sim">simulation to collide in</param>
    public void CollideWalls(Simulation sim)
    {
      double t = CollisionWalls(sim);
      translationScalar = Math.Min(translationScalar, t);
      if(t < 1.0) solidCollision = true;
    }

    /// <summary>
    /// Apply translation scalar and adjust motion vectors for collision
    /// </summary>
    public void CollisionAdjust()
    {
      translation = velocity * translationScalar;
      if(solidCollision) velocity.Zero();

      translationScalar = 1.0;
      solidCollision = false;
    }

    /// <summary>
    /// Apply next translation to entity
    /// </summary>
    public virtual void Move()
    {
      position += translation;

      // Velocity decay
      velocity -= velocity * velocityDecayRate;
    }
}
```

Listing B.12: Selections from Entity.cs

```
/// <summary>
/// Critter entity
/// By: Sam Gardner
/// 05-23-2008
/// Modified: 12-16-2008
/// </summary>
public class Critter : Entity
{
  public class CritterSensor
  {
    public double value;
    public double intensity;
  }

  protected string role;

  protected int sensorCount;
  protected int sensorResolution;

  protected List<CritterSensor> sensor;
  protected List<CritterSensor> sensorCell;


  /// <summary>
  /// Create a new critter
  /// </summary>
  public Critter() : base()
  {
    rand = new RandomGenerator();

    sensor = new List<CritterSensor>();
    sensorCell = new List<CritterSensor>();

    role = "Critter";

    sensorCount = 12;
    sensorResolution = 30;
  }

  /// <summary>
  /// Return sensed intensity of entity at specified distance
  /// <para>[0.0, 1.0]</para>
  /// </summary>
  /// <param name="distance">distance</param>
  /// <param name="sim">simulation to sense</param>
  protected double Intensity(double distance, Simulation sim)
  {
    double max = sim.Field.MaxSeparationDistance;
    if(distance <= 0.0)
      return 1.0;
```

```
  return (max - distance) / max;
}

/// <summary>
/// Imprint sensor array with perception of field boundaries
/// </summary>
/// <param name="value">sensor signature value for walls</param>
/// <param name="sim">simulation to sense</param>
protected void SenseWalls(double value, Simulation sim)
{
  double halfWidth = sim.Field.Width / 2.0;
  double halfHeight = sim.Field.Height / 2.0;

  List<Vector2> corner = new List<Vector2>();
  corner.Add(new Vector2(halfWidth, halfHeight));    // ur
  corner.Add(new Vector2(position.X, halfHeight));   // top
  corner.Add(new Vector2(-halfWidth, halfHeight));   // ul
  corner.Add(new Vector2(-halfWidth, position.Y));   // left
  corner.Add(new Vector2(-halfWidth, -halfHeight)); // ll
  corner.Add(new Vector2(position.X, -halfHeight)); // bottom
  corner.Add(new Vector2(halfWidth, -halfHeight));   // lr
  corner.Add(new Vector2(halfWidth, position.Y));    // right

  List<int> index = new List<int>();
  List<double> intensity = new List<double>();

  // Calculate angles and indices
  foreach(Vector2 v in corner)
  {
    Vector2 sep = position.To(v);
    double angle = sep.Angle();

    index.Add(CellIndex(sep.Angle()));
    intensity.Add(Intensity(sep.Length() - radius, sim));
  }

  // Fill sensor array cells
  for(int rc = 0; rc < corner.Count; rc++)
  {
    int lc = (rc + 1) % corner.Count;
    int cells = (index[lc] >= index[rc])
                ? (index[lc] - index[rc] + 1)
                : (sensorCell.Count - index[rc] + index[lc] + 1);
    double slope = (intensity[lc] - intensity[rc]) / (cells - 1);
    int i = index[rc];
    for(int c = 0; c < cells; c++)
    {
      double z = intensity[rc] + (slope * c);
      if(z > sensorCell[i].intensity)
      {
```

```
        sensorCell[i].value = value;
        sensorCell[i].intensity = z;
      }

      i++;
      i %= sensorCell.Count;
    }
  }
}

/// <summary>
/// Imprint sensor array with perception of specified entity
/// </summary>
/// <param name="e">entity to sense</param>
/// <param name="sign">value sign adjustment</param>
/// <param name="sim">simulation to sense</param>
protected void SenseEntity(Entity e, double sign, Simulation sim)
{
  if(e == this) return;

  // Calculate angles
  Vector2 sep = CenterSeparation(e);
  // angular direction to obstacle center from critter position
  double angle = sep.Angle();
  // angular radius of entity from sensing critter's position
  double theta = Math.Atan2(e.Radius, sep.Length());

  // Calculate indices
  int ri = CellIndex(angle - theta);
  int li = CellIndex(angle + theta);
  int cells = (li >= ri)
            ? (li - ri + 1)
            : (sensorCell.Count - ri + li + 1);

  // Calculate intensity
  double intensity = Intensity(SeparationDistance(e), sim);

  // Fill sensor cells
  int index = ri;
  for(int c = 0; c < cells; c++)
  {
    if(intensity >= sensorCell[index].intensity)
    {
      sensorCell[index].value = e.SensorSignature * Math.Sign(sign);
      sensorCell[index].intensity = intensity;
    }

    index++;
    index %= sensorCell.Count;
  }
}
```

```
/// <summary>
/// Fill sensors with data from sensor cells
/// </summary>
protected void FillSensors()
{
  double value;
  double intensity;
  int c = 0;

  for(int s = 0; s < sensor.Count; s++)
  {
    value = 0;
    intensity = 0;

    for(int i = 0; i < sensorResolution; i++)
    {
      if(sensorCell[c].intensity != 0.0)
      {
        value += sensorCell[c].value;
        intensity += sensorCell[c].intensity;
      }

      c++;
    }

    sensor[s].value = value / (double)sensorResolution;
    sensor[s].intensity = intensity / (double)sensorResolution;
  }
}

/// <summary>
/// Setup for a new simulation
/// </summary>
public virtual void Initialize(Vector2 start)
{
  if(script != null) script.Reset();
  DestroySensors();
  CreateSensors();
  Reset();
  Stop();
  Position = start.Copy();
}

/// <summary>
/// Read sensors
/// </summary>
/// <param name="sim">simulation to sense</param>
public virtual void Sense(Simulation sim)
{
  ClearSensorCells();
```

```
    if(sim.Field.VisibleWalls) SenseWalls(-1.0, sim);

    foreach(Obstacle o in sim.Field.Obstacle)
      SenseEntity(o, 1.0, sim);

    SenseEntity(sim.Fox, 1.0, sim);
    SenseEntity(sim.Rabbit, 1.0, sim);

    FillSensors();
  }
}
```

Listing B.13: Selections from Critter.cs

```
/// <summary>
/// Simulation class
/// By: Sam Gardner
/// 05-23-2008
/// Modified: 12-16-2008
/// </summary>
public class Simulation
{
  private RandomGenerator rand;

  private List<string> comment;

  private SimulationReplay replay;

  private string name;

  private string fitnessFunction;
  private string terminationFunction;

  private double funnelDistanceScalar;
  private double funnelAccelScalar;
  private double funnelEpsilon;
  private double proximityThreshold;
  private List<ProximityField> graduatedProximityField;

  private double targetScore;
  private int targetStep;

  private Field field;
  private Critter fox;
  private Critter rabbit;

  private Stat distance;
  private Stat approach;
```

```
  private Stat speed;
  private Stat accel;

  private double scoreQuickCapture;
  private double scoreMinDistance;
  private double scoreDiscreteProximity;
  private double scoreApproach;
  private double scoreDiscreteApproach;

  private double scoreFunnelDistance;
  private double scoreFunnelAcceleration;

  private double scoreMutuallyExclusiveDiscreteProximity;
  private double scoreMutuallyExclusiveDiscreteApproach;

  private double scoreGraduatedProximity;
  private double scoreGraduatedApproach;

  private int step;
  private bool capture;
  private bool initialized;
  private bool finished;


  /// <summary>
  /// Create a new fox rabbit simulation
  /// </summary>
  public Simulation()
  {
    rand = new RandomGenerator();

    comment = new List<string>();

    replay = null;

    name = "Simulation";

    fitnessFunction = "";
    terminationFunction = "Step";

    funnelDistanceScalar = 0.5;
    funnelAccelScalar = 0.002;
    funnelEpsilon = 0.001;
    proximityThreshold = 2.0;
    graduatedProximityField = new List<ProximityField>();
    graduatedProximityField.Add(new ProximityField(0.0, 1.00));
    graduatedProximityField.Add(new ProximityField(1.0, 0.95));
    graduatedProximityField.Add(new ProximityField(2.0, 0.90));
    graduatedProximityField.Add(new ProximityField(3.0, 0.85));
    graduatedProximityField.Add(new ProximityField(4.0, 0.80));
    graduatedProximityField.Add(new ProximityField(5.0, 0.75));
```

```
    targetScore = 100.0;
    targetStep = 500;

    field = new Field();
    Fox = new NeuralNetworkCritter();
    Rabbit = new StationaryCritter();


    distance = new Stat();
    approach = new Stat();
    speed = new Stat();
    accel = new Stat();
  }

  /// <summary>
  /// Reset stats for a new simulation
  /// </summary>
  private void InitializeStats()
  {
    distance.Clear();
    approach.Clear();
    speed.Clear();
    accel.Clear();
  }

  /// <summary>
  /// Record stats from current simulation step
  /// </summary>
  private void UpdateStats()
  {
    distance.Add(fox.SeparationDistance(rabbit));
    approach.Add((step > 0) ? -distance.Derivative : 0.0);
    speed.Add(fox.Velocity.Length());
    accel.Add(fox.Acceleration.Length());
  }

  /// <summary>
  /// Get current fox score
  /// </summary>
  public double Score()
  {
    double score = 0.0;

    switch(fitnessFunction.ToLower())
    {
      // overly complex (06-03-2008)
      // [0.0, 100.0]
      // 50% for speed (awarded upon capture)
      // 25% for distance (recomputed each time step)
      // 25% for approach (awarded per time step)
```

```
case "original":
  score += 50.0 * scoreQuickCapture;
  score += 25.0 * scoreMinDistance;
  if(scoreApproach > 0.0) score += 25.0 * scoreApproach;
  break;

// approach awarded (06-09-2008)
// [0.0, 100.0]
// 100% for capture (no-adjustments)
// 50% for approach
//   (starting with 0% and increasing with each time step)
case "discrete-approach":
  if(capture)
    score = 100.0;
  else
    if(scoreDiscreteApproach > 0.0)
      score += 50.0 * scoreDiscreteApproach;
  break;

// approach awarded and negative approach punished
// [0.0, 100.0]
// 100 for capture
// 80 for approach
//   (starting at 40 and changing with each time step)
case "bidirectional-approach":
  score = capture ? 100.0 : (40.0 + (40.0 * scoreApproach));
  break;

// approach awarded and negative approach punished (06-14-2008)
// also used in wall-less and xgallery simulations
// [0.0, 100.0]
// 100% for capture (no-adjustments)
// 80% for approach
//   (starting with 40% and changing with each time step)
case "bidirectional-discrete-approach":
  score = capture ? 100.0
                  : (40.0 + (40.0 * scoreDiscreteApproach));
  break;

// incentive for approach leading to large award for
// maintaining proximity
// bonus given for capture (06-20-2008)
// used in locality simulation
// [0.0, 100.0]
// 10% for capture (awarded upon capture)
// 60% for proximity (accumulated with each time step)
// 30% for approach
//   (starting with 15% and changing with each time step)
case "discrete-proximity-and-approach":
  if(capture) score += 10.0;
  score += 60.0 * scoreDiscreteProximity;
```

```
    score += (15.0 + (15.0 * scoreDiscreteApproach));
    break;

  // accumulates high precision score rewarding
  // non-linear closeness (funnel) (7-18-2008)
  // [0.0, +inf]
  // due to epsilon, practical upper limit is
  // funnelDistanceScalar * field.MaxSeparationDistance / epsilon
  // total: about 70,711 for kd = 0.5, epsilon = 0.001
  //   and 100 x 100 field
  case "funnel-distance-only":
    score = scoreFunnelDistance;
    break;

  // accumulates high precision score rewarding
  // non-linear closeness (funnel) and low acceleration (7-18-2008)
  // [0.0, +inf]
  // due to epsilon, practical upper distance score limit is
  // funnelDistanceScalar * field.MaxSeparationDistance / epsilon
  // the practical upper acceleration score limit is
  //   funnelAccelScalar * fox.MaxAccel / epsilon
  // total: about 70,713 for ka = 0.002, epsilon = 0.001,
  //                         100 x 100 field, and maxAccel = 1.0
  case "funnel-distance-and-acceleration":
    score = scoreFunnelDistance + scoreFunnelAcceleration;
    break;

  // points accumulated each time step for either
  // discrete proximity or discrete approach (07-21-2008)
  // [0.0, 100.0]
  // 100% max for approach (accumulated with each time step)
  // 100% max for proximity (accumulated with each time step)
  // no more than 100% total
  case "discrete-approach-or-proximity":
    score = 100.0
            * (scoreMutuallyExclusiveDiscreteProximity
               + scoreMutuallyExclusiveDiscreteApproach);
    break;

  // points accumulated each time step for either
  // discrete proximity or discrete approach (07-30-2008)
  // [0.0, 100.0]
  // 6 levels of proximity
  // 100% max for overlap
  // 95% max for within 1.0
  // 90% max for within 2.0
  // 85% max for within 3.0
  // 80% max for within 4.0
  // 75% max for within 5.0
  // 50% max for proximity
  // no more than 100% total
```

```
      case "graduated-proximity-levels":
        score = 100.0
                * (scoreGraduatedProximity
                   + (0.5 * scoreGraduatedApproach));
        break;

      // minimum distance achieved
      // [0.0, 100.0]
      case "min-distance":
        score = 100.0 * scoreMinDistance;
        break;

      // capture only
      // [0.0, 100.0]
      // 100% for capture
      default:
        score = capture ? 100.0 : 0.0;
        break;
    }

    return score;
  }

  /// <summary>
  /// Reset scores for a new simulation
  /// </summary>
  private void InitializeScores()
  {
    scoreQuickCapture = 0.0;
    scoreMinDistance = 0.0;
    scoreDiscreteProximity = 0.0;
    scoreApproach = 0.0;
    scoreDiscreteApproach = 0.0;

    scoreFunnelDistance = 0.0;
    scoreFunnelAcceleration = 0.0;

    scoreMutuallyExclusiveDiscreteProximity = 0.0;
    scoreMutuallyExclusiveDiscreteApproach = 0.0;

    scoreGraduatedProximity = 0.0;
    scoreGraduatedApproach = 0.0;
  }

  /// <summary>
  /// Update score values from current simulation stats
  /// </summary>
  private void UpdateScores()
  {
    double point = Math.Round(1.0 / (double)targetStep, 10);
    double dist = (distance.Last > 0.0) ? distance.Last : 0.0;
```

```
scoreQuickCapture = 1.0 - ((double)step / (double)targetStep);
if(scoreQuickCapture > 1.0) scoreQuickCapture = 1.0;
if(scoreQuickCapture < 0.0) scoreQuickCapture = 0.0;

scoreMinDistance = 1.0
                 - (distance.Min / field.MaxSeparationDistance);
if(scoreMinDistance > 1.0) scoreMinDistance = 1.0;
if(scoreMinDistance < 0.0) scoreMinDistance = 0.0;

if(distance.Last <= proximityThreshold)
  scoreDiscreteProximity += point;

scoreApproach += approach.Last / fox.MaxSpeed * point;
if(approach.Last > 0.0) scoreDiscreteApproach += point;
if(approach.Last < 0.0) scoreDiscreteApproach -= point;
if(scoreDiscreteProximity > 1.0) scoreDiscreteProximity = 1.0;
if(scoreDiscreteApproach > 1.0) scoreDiscreteApproach = 1.0;
if(scoreDiscreteApproach < -1.0) scoreDiscreteApproach = -1.0;
if(scoreApproach > 1.0) scoreApproach = 1.0;
if(scoreApproach < -1.0) scoreApproach = -1.0;

scoreFunnelDistance += funnelDistanceScalar
                     * field.MaxSeparationDistance
                     / (dist + funnelEpsilon)
                     / (double)targetStep;

scoreFunnelAcceleration += funnelAccelScalar
                         * fox.MaxAccel
                         / (accel.Last + funnelEpsilon)
                         / (double)targetStep;

if(distance.Last <= proximityThreshold)
  scoreMutuallyExclusiveDiscreteProximity += point;
else if(approach.Last > 0.0)
  scoreMutuallyExclusiveDiscreteApproach += point;

if(scoreMutuallyExclusiveDiscreteProximity > 1.0)
  scoreMutuallyExclusiveDiscreteProximity = 1.0;
if(scoreMutuallyExclusiveDiscreteApproach > 1.0)
  scoreMutuallyExclusiveDiscreteApproach = 1.0;

double pscore = 0.0;
foreach(ProximityField p in graduatedProximityField)
  if(dist <= p.cutoff)
  {
    pscore += (p.pointScale * point);
    break;
  }
if(pscore == 0.0 && approach.Last > 0.0)
  scoreGraduatedApproach += point;
```

```
    scoreGraduatedProximity += pscore;

    if(scoreGraduatedProximity > 1.0) scoreGraduatedProximity = 1.0;
    if(scoreGraduatedApproach > 1.0) scoreGraduatedApproach = 1.0;
  }

  /// <summary>
  /// Detect and react to collisions between fox/rabbit and environment
  /// </summary>
  private void CollisionCorrection()
  {
    fox.CollideWalls(this);
    rabbit.CollideWalls(this);
    foreach(Obstacle o in field.Obstacle)
    {
      fox.Collide(o);
      rabbit.Collide(o);
    }

    fox.CollisionAdjust();
    rabbit.CollisionAdjust();

    if(fox.Collision(rabbit) < 1.0) capture = true;
  }

  /// <summary>
  /// Reset the simulation for another run
  /// </summary>
  public void Reset()
  {
    InitializeCritter(rabbit);
    if(fox.StartRelative) fox.Start = rabbit.Position.Copy();
    InitializeCritter(fox);

    fox.Sense(this);
    rabbit.Sense(this);

    step = 0;
    capture = false;
    finished = false;

    if(fox.Overlapping(rabbit)) capture = true;

    InitializeStats();
    InitializeScores();
    replay = new SimulationReplay();
    replay.Initialize(this);

    initialized = true;
  }
```

```
/// <summary>
/// Set up specified critter for a new simulation
/// </summary>
private void InitializeCritter(Critter c)
{
  // Find starting position
  if(c.StartAnywhere)
  {
    c.Start.Zero();
    c.StartRange = new Vector2(field.Width / 2.0 - c.Radius,
                               field.Height / 2.0 - c.Radius);
  }
  c.Initialize(c.StartPosition());

  // Correct for initial overlap
  c.OverlapCorrectWalls(this);
  foreach(Obstacle o in field.Obstacle)
    c.OverlapCorrect(o);
}

/// <summary>
/// Run the simulation for a single time step
/// </summary>
public void RunStep()
{
  if(!finished)
  {
    if(!initialized)
    {
      Reset();
    }
    else
    {
      fox.ComputeNextAction(this);
      rabbit.ComputeNextAction(this);

      UpdateStats();
      UpdateScores();
      replay.Update(this);

      CollisionCorrection();

      fox.Move();
      rabbit.Move();

      fox.Sense(this);
      rabbit.Sense(this);

      step++;
```

```
        if(Terminate())
        {
          finished = true;
          UpdateStats();
          UpdateScores();
          replay.Update(this);
        }
      }
    }
  }

  /// <summary>
  /// Return true if simulation should terminate
  /// </summary>
  public bool Terminate()
  {
    switch(terminationFunction.ToLower())
    {
      case "any":
        if(capture) return true;
        if(Score() >= targetScore) return true;
        if(step >= targetStep) return true;
        break;
      case "capture-or-score":
        if(capture) return true;
        if(step >= targetStep) return true;
        break;
      case "capture-or-step":
        if(capture) return true;
        if(step >= targetStep) return true;
        break;
      case "score-or-step":
        if(Score() >= targetScore) return true;
        if(step >= targetStep) return true;
        break;
      case "capture":
        if(capture) return true;
        break;
      case "score":
        if(Score() >= targetScore) return true;
        break;
      case "step":
        if(step >= targetStep) return true;
        break;
    }

    return false;
  }
}
```

Listing B.14: Selections from Simulation.cs

Statistic Code

```
/// <summary>
/// Stat class
/// By: Sam Gardner
/// 04-10-2008
/// Modified: 06-20-2008
/// </summary>
public class Stat : IComparable<Stat>
{
  protected string name;

  protected int size;
  protected List<double> history;

  protected int count;
  protected double integral;
  protected double derivative;
  protected double max;
  protected double avg;
  protected double min;
  protected double diffSqIntegral;

  protected int maxIndex;
  protected int minIndex;

  /// <summary>
  /// Create a new statistic
  /// </summary>
  public Stat()
  {
    name = "Stat";

    size = 5;
    history = new List<double>();

    ClearStat();
  }

  /// <summary>
  /// Set stat members to initial values
  /// </summary>
  private void ClearStat()
  {
    history.Clear();

    count = 0;
    integral = 0.0;
    derivative = 0.0;
```

```
    max = 0.0;
    avg = 0.0;
    min = 0.0;
    diffSqIntegral = 0.0;

    maxIndex = -1;
    minIndex = -1;
  }

  /// <summary>
  /// Get current variance of all values
  /// accounted for by this statistic
  /// </summary>
  public double Variance
  {
    get { return diffSqIntegral / (double)count; }
  }

  /// <summary>
  /// Get current standard deviation of all values
  /// accounted for by this statistic
  /// </summary>
  public virtual double StdDev
  {
    get { return Math.Sqrt(Variance); }
  }


  /// <summary>
  /// Calculate the instantaneous running average for the statistic
  /// accross the specified number of data points
  /// <para>points must be less than or equal to history size</para>
  /// </summary>
  /// <param name="points">number of values to average</param>
  public virtual double RunningAvg(int points)
  {
    double sum = 0.0;
    int p = 0;
    while(p < points && p < history.Count)
    {
      sum += history[history.Count - 1 - p];
      p++;
    }

    return sum / (double)points;
  }

  /// <summary>
  /// Add a new value to the statistic calculations
  /// </summary>
  /// <param name="value">value to record</param>
```

```
  public virtual void Add(double value)
  {
    double diff;
    double last = Last;

    history.Add(value);
    if(history.Count > size) history.RemoveAt(0);

    integral += value;
    derivative = value - last;

    if(value > max || count == 0)
    {
      max = value;
      maxIndex = count;
    }
    if(value < min || count == 0)
    {
      min = value;
      minIndex = count;
    }

    count++;

    avg = integral / (double)count;
    diff = value - avg;
    diffSqIntegral += diff * diff;
  }
}
```

Listing B.15: Selections from Stat.cs

```
/// <summary>
/// StatList class
/// By: Sam Gardner
/// 04-10-2008
/// Modified: 01-06-2009
/// </summary>
public class StatList : IEnumerable<Stat>
{
  private List<Stat> stat;


  /// <summary>
  /// Create a new stat list
  /// </summary>
  public StatList()
  {
    stat = new List<Stat>();
```

```
  }

  /// <summary>
  /// Add statistics for each value of a statistic
  /// with the given name
  /// <para>no data is recorded</para>
  /// </summary>
  /// <param name="name">base name</param>
  public void AddMeta(string name)
  {
    Add(name + "-max");
    Add(name + "-avg");
    Add(name + "-min");
    Add(name + "-stddev");
    Add(name + "-variance");
    Add(name + "-derivative");
  }

  /// <summary>
  /// Add statistics for each value of all statistics in list
  /// <para>no data is recorded</para>
  /// </summary>
  /// <param name="list">stat list</param>
  public void AddMeta(StatList list)
  {
    foreach(Stat s in list)
      AddMeta(s);
  }

  /// <summary>
  /// Record statistics for each value of a statistic
  /// with a different name
  /// <para>statistics are created if they don't exist</para>
  /// </summary>
  /// <param name="s">statistic to sample</param>
  /// <param name="name">new stat name</param>
  public void RecordMetaAs(Stat s, string name)
  {
    this[name + "-max"].Add(s.Max);
    this[name + "-avg"].Add(s.Avg);
    this[name + "-min"].Add(s.Min);
    this[name + "-stddev"].Add(s.StdDev);
    this[name + "-variance"].Add(s.Variance);
    this[name + "-derivative"].Add(s.Derivative);
  }
}
```

Listing B.16: Selections from StatList.cs