EVOLVING NEURAL NETWORKS WITH HYPERNEAT

AND ONLINE TRAINING

by

Shaun M. Lusk, B.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
May 2014

Committee Members:

Wuxu Peng, Chair

Moonis Ali

Mina Guirguis

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

ACKNOWLEDGMENTS

Firstly, I would like to thank Dr. Kaikhah for his guidance throughout my research. He always challenged me to think about things in different ways. It is my hope that he would have been pleased with the final results of my work.

I extend my thanks to Dr. Peng who graciously stepped in to serve as my advisor during the final stages of my work and to Dr. Ali and Dr. Guirguis of my thesis committee. A special thanks also goes to Dr. Ali for taking extra time to assist me in completing this work.

Last, but certainly not least, I would like to thank my beautiful wife, Jessica “Rayven”, for being at my side through the ups and downs of my journey though education.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

ABSTRACT

Artificial neural network research of the past decade has seen significant growth with the advent of genetic algorithms such as NSGA and NEAT to develop neural networks through evolution. Another more recent advance in this technology is the HyperNEAT algorithm, an extension to the highly successful NEAT algorithm, which is capable of capturing the symmetry of a domain. HyperNEAT has been very successful for evolving agent controllers, and as such it seems a good platform for exploring hybrid techniques.

Our research focuses on augmenting HyperNEAT technology for use in agent controllers through strategic application of online learning. Several methods are proposed and explored.

All methodologies are tested using a team gathering task. A simulated environment is setup with gathering robots that must locate resources and work together to carry the resources back to a central base location. The robots are controlled by the networks produced by the HyperNEAT algorithm (referred to as "substrates").

In the first set of experiments, several types of online learning are combined with HyperNEAT. In all cases, evolution proceeds as normal until the evaluation phase; at this point the HyperNEAT substrate is trained in an online fashion using a given training technique. The learning methods explored are: supervised backpropagation,

reinforcement backpropagation, Hebbian learning, and temporal difference learning. These are compared against the baseline HyperNEAT algorithm with no online learning.

Next, the methodology of applying online learning is extended in an attempt to find optimal learning rate parameters for each of the learning techniques; this shall be referred to as parameter selection. The HyperNEAT algorithm uses Compositional Pattern Producing Networks (CPPNs) to generate the connection weight values for its substrates. The CPPN is augmented to also generate learning parameters for each of the other training algorithms. The initial set of experiments is repeated using the learning parameter selection approach. One additional training technique is added, the ABC variant of Hebbian learning, which uses additional parameters to control neural plasticity.

These two sets of experiments are repeated with an additional enhancement, to treat learning ability as a fitness measure. Each substrate is evaluated multiple times, with the agent environment reset between evaluations. The performance of each is recorded, and then the factor of improvement between evaluations (due to the online learning) is measured, and subsequently incorporated into the fitness score for the chromosome that produced the CPPN and substrate. Thus, individuals that demonstrate responsiveness to online learning will be favored, and will be more likely to produce offspring for future generations.

A different set of experiments is also performed examining a few other approaches.

These approaches focus on combining HyperNEAT with a couple of variants of heuristically supervised backpropagation for online learning.

The main variant that is tested involves performing geometric translations (in this case, rotations) to training samples during backpropagation, in attempt to take advantage of the substrate's symmetry. This is compared with baseline HyperNEAT; with basic backpropagation; and with repeated backpropagation, where each training sample is issued multiple times. The latter approach is introduced in order to account for the possibility that the performance of rotational backpropagation is enhanced purely due to the number of training iterations performed per sample.

Based on the results from this set of experiments, and the different strengths of the original HyperNEAT algorithm versus the addition of online learning, a another set of experiments is performed using a technique we call bootstrapping that uses online learning during the early stages of evolution, but switches it off when a certain average level of fitness is achieved. The results of these experiments suggest that some initial online training may produce more optimal results than with constant online training, or with none.

One final approach we explore is to attempt to reinforce useful behaviors performed by the agents during evaluations. This approach, referred to as HyperNEAT with training banks, identifies when the agent arrives in a state that should be rewarded, and collects

the inputs and outputs that resulted in that state in a repository (the training bank). Then, between HyperNEAT evaluations, the inputs and outputs from the training states are used as training samples, and the network is trained using backpropagation, repeated backpropagation, or rotational backpropagation. The results from these experiments show that networks evolved with HyperNEAT using rotational backpropagation applied via training banks exhibit a a higher degree of generalizability than HyperNEAT alone.

1. INTRODUCTION

Artificial intelligence is a relatively new field in the computer sciences, having existed for less than a century, but is evolving rapidly in the modern era of computing. What is artificial intelligence (AI)? A better question with which to start might be, "what is intelligence?" The concept of intelligence itself is difficult to define. A report from the Board of Scientific Affairs of the American Psychological Association discusses human intelligence, noting certain key traits such as the "ability to understand complex ideas, to adapt effectively to the environment, to learn from experience, to engage in various forms of reasoning, to overcome obstacles by taking thought" [1]. The report also states that these abilities can vary from person to person, and even vary for a single person when observed in different contexts. It is a complex phenomenon for which there exists no universally agreed upon definition. However, for the purposes of introducing artificial intelligence, the broad description of the abilities associated with intelligent organisms is sufficient.

This brings us back to the question – what is artificial intelligence? In the text, *Artificial Intelligence*, by Rich and Knight [2], it is described most simply as "the study of how to make computers do things which, at the moment, people do better." In general, it is useful to think of artificial intelligence as using machines (physical or virtual) to simulate such abilities as adapting, learning from experience, understanding, and reasoning.

In the ever expanding field of AI, many different approaches and techniques have been proposed and explored for a wide variety of problems. One such approach is the artificial neural network. This approach seeks to capture the underlying mechanics of animal

brains – networks of biological neurons – and recreate them in hardware or software. The notion itself is relatively simple: a neuron is electrically activated upon receiving a stimulus, and then transmits a signal to other neurons. However simple, it is the vast interconnection of neurons that form human and animal brains, and ultimately makes intelligence possible.

The functioning of artificial neurons is analogous to biology: a neuron, or a row ("layer") of neurons is presented with a stimulus ("input"), typically a vector of numbers. These numbers are additively combined into a single activation value for each receiving neuron. Each neuron also has a particular threshold or bias that impacts the incoming activation signal. This bias can either strengthen or inhibit the signal, which is then transmitted as an output signal, potentially to other neurons.

Artificial neural networks (ANNs) are particularly attractive to AI researchers for a couple of reasons. The great strength of ANNs is that through various algorithms they can be changed, and can learn, similar to the way that biological neurons function. As well, they tend to exhibit good generalizability; that is, they have the ability to respond effectively to stimuli that were never encountered during training. Because of these traits, neural networks have been successful in applications such as pattern recognition, image recognition, sequence prediction, classification, clustering, agent/robotic controllers, and others still.

In their earliest experiments, artificial neural networks were designed by hand, for simple mathematical operations, or for mapping one set of vectors to another. However, these

designs were fairly limited, and algorithms were developed to enable the networks to be trained to produced desired output. The earliest algorithms enabled a single neuron or a simple network to be automatically updated until their output reached a sufficiently low level of error for a given operation.

A major breakthrough was made with the backpropagation algorithm. This algorithm extended previous algorithms to allow neural networks with multiple layers of neurons to be trained. In order to train a network with backpropagation, a data set is needed that contains pairs of inputs and desired outputs. A *desired* output is the output that a network should respond with when a corresponding input is presented. Training takes place by presenting each input to the network and calculating an error based on the difference between the network's actual output, and the desired output. Using the error from each training sample, backpropagation updates the inter-neuron connections within the network. This process is repeated for all training samples, for many iterations ("epochs") through the complete data set until a globally minimum error value is reached, or at least until a particular error threshold is achieved.

Another major breakthrough came when researchers began to use genetic algorithms to evolve networks, rather than training them. This opened many opportunities to use neural networks in ways not previously possible. While training with a data set can be effective, it does require a rather large number of samples with expected outputs – for some domains this information is not available or is impossible to gather with current technology. While we may not always have expected outputs, in many cases it is possible to identify whether a given state is good or not, an idea that forms the basis of

reinforcement learning. By being able to identify when a network has produced an output that is good or effective, it is possible to generate a “fitness” score for the network. In genetic algorithms, this allows us to compare and rank a population of networks, and produce offspring from the best performers. In the past 15 years or so, a great deal of research has been dedicated to techniques that evolve neural networks.

Our research focuses on augmenting HyperNEAT, a genetic algorithm for evolving neural networks, through strategic application of online learning. Several methods are proposed and explored.

All methodologies are tested using a team gathering task. A simulated environment is setup with gathering robots that must locate resources and work together to carry the resources back to a central base location. The robots are controlled by networks produced by the HyperNEAT algorithm (referred to as "substrates").

In the first set of experiments, several types of online learning are combined with HyperNEAT. In all cases, evolution proceeds as normal until HyperNEAT's evaluation phase; at this point the HyperNEAT substrate is trained in an online fashion using a given training technique. The training methods explored are: supervised backpropagation, reinforcement backpropagation, Hebbian learning, and temporal difference learning. These are compared against the baseline HyperNEAT algorithm with no online learning.

Next, the methodology of applying online learning is extended in an attempt to find optimal learning rate parameters for each of the learning techniques; this shall be referred to as parameter selection. The HyperNEAT algorithm uses Compositional Pattern

Producing Networks (CPPNs) to generate the connection weight values for its substrates. The CPPN is augmented to also generate learning parameters for each of the other training algorithms. The initial set of experiments is repeated using the learning parameter selection approach. One additional training technique is added, the ABC variant of Hebbian learning, which uses additional parameters to control neural plasticity.

These two sets of experiments are repeated with an additional enhancement, to treat learning ability as a fitness measure. Each substrate is evaluated multiple times, with the agent environment reset between evaluations. The performance of each is recorded, and then the factor of improvement between evaluations (due to the online learning) is measured, and subsequently incorporated into the fitness score for the chromosome that produced the CPPN and substrate. Thus, individuals that demonstrate responsiveness to online learning will be favored, and will be more likely to produce offspring for future generations.

A different set of experiments is also performed examining a few other approaches. These approaches focus on combining HyperNEAT with a variant of heuristically supervised backpropagation for online learning.

This approach involves performing geometric translations (in this case, rotations) to training samples during backpropagation, in attempt to take advantage of the substrate's symmetry. This is compared with baseline HyperNEAT, with basic backpropagation, and with repeated backpropagation, where each training sample is issued multiple times. The latter approach is introduced in order to account for the possibility that the performance

of rotational backpropagation is enhanced purely due to the number of training iterations performed per sample.

Based on the results from this set of experiments, and the different strengths of the original HyperNEAT algorithm versus the addition of online learning, an additional set of experiments is performed using a technique we call bootstrapping that uses online learning during the early stages of evolution, but switches it off when a certain average level of fitness is achieved. The results of these experiments suggest that some initial online training may produce more optimal results than with constant online training, or with none.

One final approach is to attempt to reinforce useful behaviors performed by the agents during evaluations. This approach, referred to as HyperNEAT with training banks, identifies when the agent arrives in a state that should be rewarded, and collects the inputs and outputs that resulted in that state in a repository (the training bank). Then, between HyperNEAT evaluations, the inputs and outputs from the training states are used as training samples, and the network is trained using backpropagation, repeated backpropagation, or rotational backpropagation. The results from these experiments show that networks evolved with HyperNEAT using rotational backpropagation applied via training banks exhibit a a higher degree of generalizability than HyperNEAT alone.

2. NEURAL NETWORKS AND GENETIC ALGORITHMS

2.1. Neural Networks

Before delving into the functioning of artificial neural networks it is useful to describe, at a very general level, how biological neural networks function.

Carlson [3] provides an introduction to this topic in *Foundations of Physiological Psychology.* Biological neural networks, as the name implies are (vast) networks of interconnected neurons. Each neuron consists of several components that receive, process, and transmit information. The body of a neuron is called the *soma;* it combines signals received from other neurons through many hair-like extensions called dendrites. The soma transmits the resultant signal down a pathway called an axon. The axon splits off into several branches, each ending in a terminal button. The signal travels through the axon to the terminal buttons, and then jumps a small gap, a synapse, to reach the dendrites of other neurons. In this way, a single neuron may be connected to a multitude of other neurons, collectively forming a neural network. Figure 2.1 (source [3]) depicts a biological neuron.

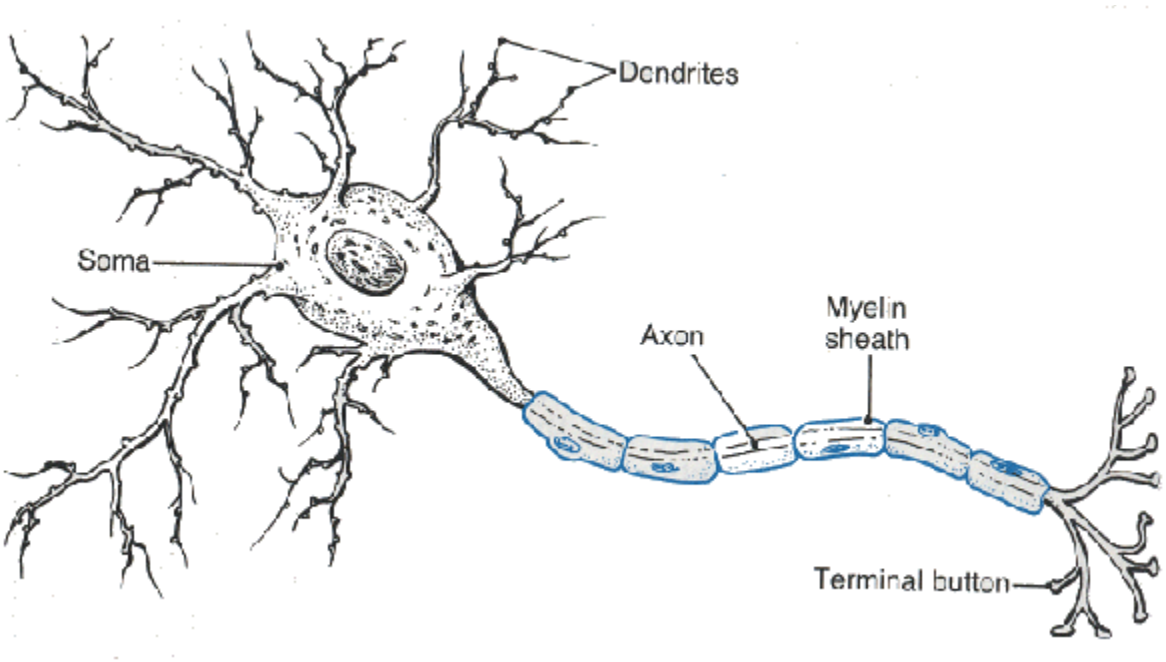


Figure 2.1: A biological neuron.

Synaptic contacts from one neuron to another may be excitatory or inhibitory. Excitatory contacts increase the electrical activity of the neuron that receives them. Inhibitory decrease that activity. If a neuron is sufficiently stimulated by incoming impulses, it will "fire" and transmit its own signal to other neurons. Each neuron may have a threshold that determines what constitutes sufficient stimulation. Thus some neurons will fire more easily and more often than others.

Obviously this is a highly simplified explanation of how neurons and neural networks function, but these elements provide the basis for modeling artificial neural networks (ANNs) in hardware or software. In similar fashion to their biological inspirators, ANNs are composed of many interconnected (simulated) neurons.

Mehrotra, et al. [4], describe common architectures for neural networks in *Elements of*

Artificial Neural Networks. Typically the layout or topology of a network consists of multiple layers of nodes (neurons). At a minimum, there are two layers of nodes, an input layer and an output layer. Optionally, one or more 'hidden' layers, layers in between the input and output, may be present. In common cases, ANNs are fully connected, that is, each node in a layer is connected to each node in the next layer up.

As an example, consider a simple network with four input nodes, a layer of five hidden nodes, and three output nodes (Figure 2.2). There would exist twenty connections between the input and hidden layers, and fifteen connections between the hidden and output layers.

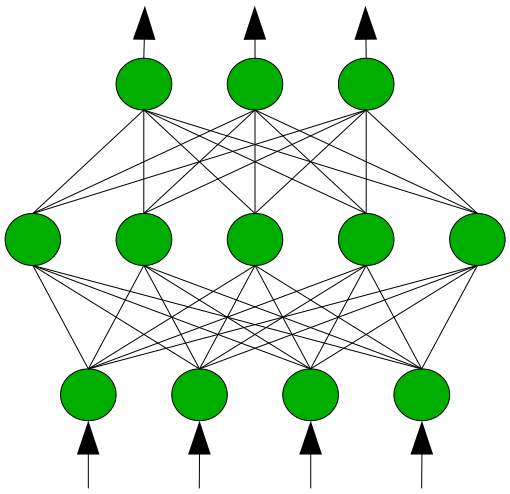


Figure 2.2: Topology of a simple artificial neural network.

Mehrotra, et al. [4], further describe the arrangement of layers in artificial neural networks. In practice, neural networks are commonly designed with three layers: one input layer, one hidden layer and one output layer. There is a relationship between whether or not the network has a hidden layer, and the complexity of problems the

network can be used to solve. Networks without a hidden layer are relatively limited; they are only capable of approximating linearly separable functions. The addition of a hidden layer removes this limitation. It has been proven that networks with a hidden layer can be used as universal function approximators for continuous functions [5]. It is possible to use multiple hidden layers, but little research shows any advantage to doing this, and can even present problems when used with the backpropagation algorithm, as error signals diminish as they are propagated backward through a network [4].

How many nodes should be included in a hidden layer of the network is something of an unsolved problem. While many theories and formulae have been posed to determine how many neurons should be included in a network, either for specific problems, or the general case, often the optimum number of neurons is determined through ancillary algorithms, analysis, or simple experimentation [4].

Artificial neural networks can be constructed in software using implementations as simple as a tables of real valued numbers. The activation value of a node may be represented this way, as can the weight value on a connection, with positive or negative values being analogous to excitatory or inhibitory connections, respectively. Input values to the network are typically binary, or real valued. If real values are used, they are typically normalized between 0 and 1 or between -1 and 1.

Following this schema, it is easy enough to understand the basic functioning of an ANN. Inputs are presented to the network and propagated through each layer to the output layer. The activation value of a given node in a hidden or output layer is calculated by summing

the value of all incoming connections. The value of an incoming connection is the product of the weight of that connection and the activation value of the node on the pre-synaptic end of the connection. The activation values of the nodes in the output layer are the network's outputs [4].

While these explanations provide the foundation for the functioning of an artificial neural net, a net by itself does not solve problems or produce meaningful output unless very carefully designed from the outset. Some method must exist to train the network to produce desirable outputs. This is the advantage of neural networks: instead of being fixed, they are able to change over time and can thus learn to approximate functions.

2.2. Traditional Training Methods

In order to be useful and solve problems, neural networks must be trained. In most cases, weight connections are randomized when the network is created, thus the output itself is random. Through the course of training, the network gradually improves in accuracy, until a specified goal or cutoff point is reached.

There exist many methodologies for training neural networks. The ones relevant to this research are described here.

2.2.1. Backpropagation

One of oldest but most successful methods of training neural nets is backpropagation. In essence, the backpropagation algorithm works by propagating activation signals through a network as usual, and then when the output is observed, providing some training value

that is propagated back through the network, updating the connection weights at each layer. This method can be used for associative learning, that is, to map an input to an output; for classification tasks that label an input as belonging to one of a set of categories; for prediction, where a value or outcome is predicted based on an input; and for countless other tasks.

Backpropagation for Supervised Learning

Mehrotra, et al. [4], provide the original form of the backpropagation algorithm that is used to train networks. Training is accomplished with supervised learning, that is, where training samples consist of input patterns paired with known output patterns. In this way, each output value in a training sample is accepted to be the correct output for the corresponding input; these output values are often referred to as teacher values or desired outputs. Outputs produced by the network in response to an input pattern are known as actual outputs. The difference between the desired outputs and the actual outputs is the error. The goal of backpropagation is to find the set of weight values for the network that produce the smallest global error for a set of training data (or a global error within an accepted range).

Mean Squared Error (MSE) is the most common measure of error used in backpropagation. The MSE for a network for a given set of training data is calculated by taking the average of the squared error values for all training samples. For a neural network, each training input is presented to the network, and the difference between the actual output and the desired output is squared and recorded. When all training samples

are exhausted, the squared error values are averaged. Squared errors are used in order to make larger error values exponentially more significant than relatively small errors.

For a given network and set of training data, different weight values will produce differing amounts of error; some sets of weight values will produce more accurate outputs than others. One way of modeling this is to represent the accuracy of the network with respect to the weight values as a hyperparaboloid. To graph this, the weight value vector forms a set of multidimensional coordinates, and the error for that weight vector is the vertical coordinate. The figure below illustrates a possible graph of a weight vector with two weight values; as a representation of a hypothetical network accuracy, the x and y ranges represent the values of a two-value weight vector and the z range represents the error.

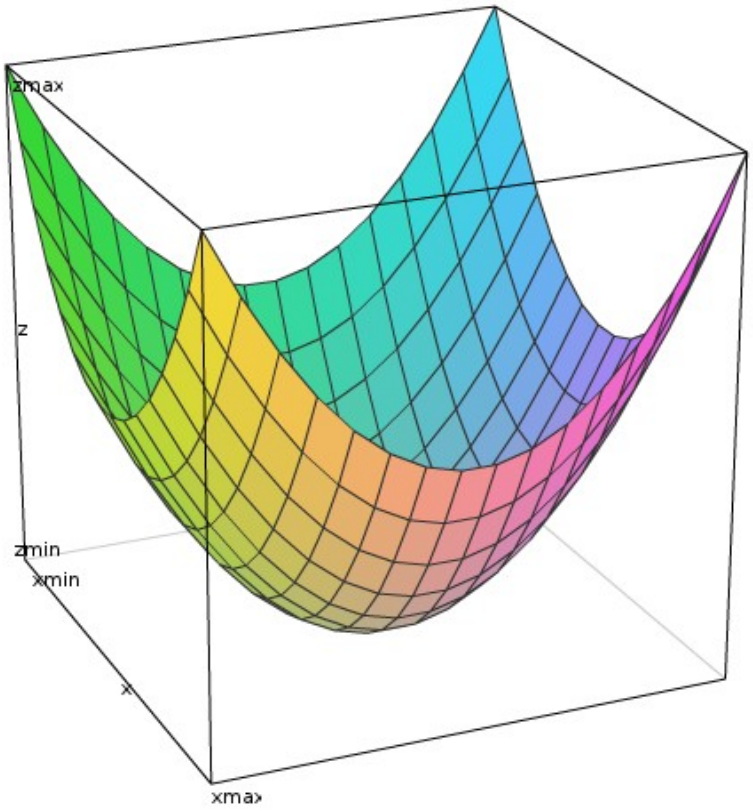


Figure 2.3: A paraboloid.

It is important to note that in many cases the surface of such a hyperparaboloid is not smooth; it may be shaped more like a rolling mountain range with multiple peaks and valleys. Small peaks and valleys in the error are known as local maxima and local minima respectively. The most extreme values are global maxima and minima, as illustrated in Figure 2.4.

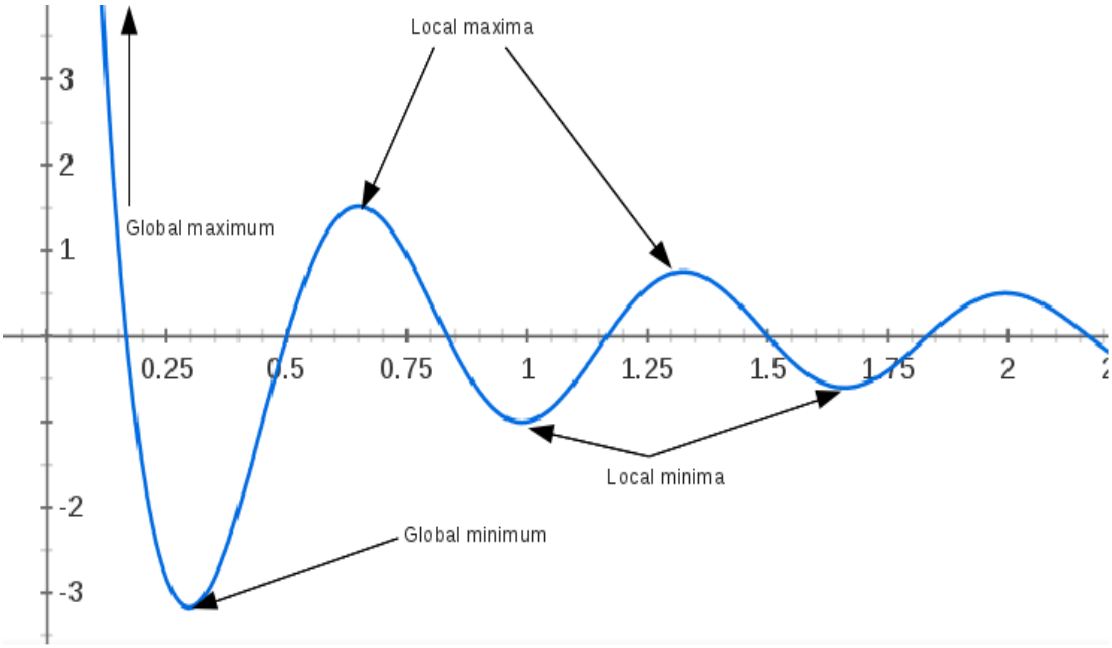


Figure 2.4: Graph of cos(3πx)/x, illustrating global and local minima and maxima.

In order to find the global error minimum for a network, a method known as gradient descent is employed. Gradient descent calculates the direction of the steepest downward slope of the error curve, and adjusts weight values accordingly. Over (many) successive adjustments, the error values may reach the global minimum. By using the steepest downward slope, the algorithm avoids getting stuck in local minima. When global minimum is reached, the algorithm is said to have converged.

The steepest downward slope may be found by calculating the derivative of the error with respect to the weights. This is done by using the chain rule of derivatives to combine the partial derivatives of (1) the error with respect to the output; (2) the output with respect to the input values of the output layer; and (3) the input values of the output layer with respect to the weights.

The derivative of the error with respect to the output is:

$$\frac{\partial E}{\partial o_k} = -2(d_k - o_k)$$

where *d* is the desired output for node *k* and *o* is the actual output for node *k*.

Next, we need to find the derivative of the output with respect to the input of the output layer. Assuming a network with a single hidden layer, the input to a node *k* in the output layer is calculated:

$$net_k^{(2)} = \sum_j w_{k,j}^{(2,1)} * x_j^{(1)}$$

where *w* is the weight between node *k* in the output layer (layer 2) and node *j* in the hidden layer (layer 1), and *x* is the activation value of node *j* in the hidden layer. This gives $\partial o_k / \partial net_k^{(2)} = S'(net_k^{(2)})$ for the derivative of the output with respect to the incoming inputs, where *S'(x) = dS(x)/dx*.

The derivative of the net input to a node in the output layer with respect to the weight is $\partial net_k^{(2)} / \partial w_{k,j}^{(2,1)} = x_j^{(1)}$. Adding that to the chain rule results in:

$$\frac{\partial E}{\partial w_{k,j}^{(2,1)}} = \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial net_k^{(2)}} \frac{\partial net_k^{(2)}}{\partial w_{k,j}^{(2,1)}}$$

giving:

$$\frac{\partial E}{\partial w_{k,j}^{(2,1)}} = -2(d_k - o_k) S'(net_k^{(2)}) x_j^{(1)}$$

That gives us the gradient for the weights between the output layer and previous layer. However, in networks that make use of one or more hidden layers, it is also necessary to calculate the gradient for the weights between the hidden layer and input layer. For a network with one hidden layer, continuing the chain of dependencies through to the connections between the input and hidden layer gives:

$$\frac{\partial E}{\partial w_{j,i}^{(1,0)}} = \sum_{k=1}^{K} \frac{\partial E}{\partial o_k} \frac{\partial net_k^{(2)}}{\partial x_j^{(1)}} \frac{\partial x_j^{(1)}}{\partial net_j^{(1)}} \frac{\partial net_j^{(1)}}{\partial w_{j,i}^{(1,0)}}$$

$$= \sum_{k=1}^{K} [-2(d_k - o_k) S'(net_k^{(2)}) w_{k,j}^{(2,1)} S'(net_j^{(1)}) x_i]$$

With the gradients calculated for each weight, the weight updates can be calculated thus:

$$\Delta w_{k,j}^{(2,1)} = \alpha * \delta_k * x_j^{(1)}$$

where

$$\delta_k = (d_k - o_k) S'(net_k^{(2)})$$

for the hidden-to-output connections and

$$\Delta w_{j,i}^{(1,0)} = \alpha * \mu_j * x_i$$

where

$$\mu_j = \left(\sum_k \delta_k w_{k,j}^{(2,1)}\right) S'\left(net_j^{(1)}\right)$$

for the input-to-hidden connections.

In these formulas, α is a learning rate parameter that controls how much weights are changed each time. Larger learning rates cause more significant weight change, and potentially faster convergence, though in some cases may cause too much change and ultimately prevent the algorithm from converging to an optimum. Smaller learning rates obviously cause a slower rate of change, and may require many more trials in order to converge. It is also possible that learning rates that are too small will not overcome local minima.

Note that $S'(x)$ is the derivative of the node's activation function, $S(x)$. In order to use backpropagation, an activation function must be used that is differentiable. If all nodes in a network use sigmoidal activation as is common, then the derivative is:

$$S'(x) = S(x)(1 - S(x))$$

which gives:

$$\delta_k = (d_k - o_k) o_k (1 - o_k)$$

The backpropagation algorithm works by propagating an input vector through each layer of the network to produce an output. The output is compared against the expected output and the squared error and gradients are calculated. The connection weights of each layer

are updated as described above. The process is repeated for each input in the training set. The completion of the full set of training data is called an 'epoch'; epochs are repeated until the error rate is reduced to zero or below a desired threshold. At that point, the weights may be frozen with their current values.

As an alternative, instead of updating weights for each training sample, weight changes may be accumulated for an epoch and the changes applied to the network weights at the end of the epoch.

The complete backpropagation algorithm for a three-layer network is as follows [4]:

```
Algorithm Backpropagation:

  Start with randomly chosen weights;

  while MSE is unsatisfactory:

     for each input pattern X_p, 1<=p<=P:

        Compute hidden node inputs ( net^(1)_{p,j} );

        Compute hidden node outputs ( x^(1)_{p,j} );

        Compute inputs to thee output nodes ( net^(2)_{p,k} );

        Compute the network outputs( o_{p,k} );

        Compute the error between o_{p,k} and desired output d_{p,k};

        Modify the weights between hidden and output nodes:

           Δ w^(2,1)_{k,j} = α (d_{p,k} − o_{p,k}) S'(net^(2)_{p,k}) x^(1)_{p,j}
```

```
        Modify the weights between input and hidden nodes:

            Δ w_{j,i}^{(1,0)} = α Σ_k ((d_{p,k} − o_{p,k}) S'(net_{p,k}^{(2)}) w_{k,j}^{(2,1)}) S'(net_{p,j}^{(1)}) x_{p,i}

    end for

end while
```

Reinforcement Learning

An alternative approach to supervised learning is reinforcement learning. In supervised learning, training samples are pairs of inputs and desired outputs. By contrast no such desired outputs exist in reinforcement learning. Yet, reinforcement learning still provides feedback to the mechanism. This is most often used in control type problems where an agent must navigate or otherwise interact with an environment. The agent performs some action in an environment, and a new environment state is observed. The state may be beneficial, neutral, or detrimental to the agent. In the case that the action was beneficial, some positive reinforcement is given to the agent; if it was detrimental, an anti-reinforcement or punishment may be given [6].

Different strategies for using backpropagation for reinforcement learning exist. A simple model employs backpropagation to directly reinforce the network. Backpropagation is performed as normal, but instead of using a target value at each step, a reward value of 1 or 0 is provided for each node depending on whether the state resulting from the selected action was beneficial or not [7]. This serves to strengthen weight connections that contributed to the selected action.

2.2.2. Hebbian

Hebbian learning was one of the earliest strategies for changing the weights of a neural network. This is based on a concept from biological neural nets, that connections between neurons will be strengthened if those neurons are frequently activated at the same time [4].

The method of updating the weights is very simple. Each time a weight is updated, its new value is equal to its old value plus the product of the activation value of the weight's incoming node and the activation value of the node the weight is projecting into. Typically some learning rate parameter is applied to the weight change to put a bound on how quickly the changes occur, just as with other training algorithms. The simplest form of Hebbian uses this formula for weight changes:

$$\Delta w_{ij} = \eta o_i o_j$$

where w_{ij} is the weight between node i of the next layer and node j of the previous layer, and o_i and o_j are the output values of the nodes i and j respectively, and η is the learning rate parameter.

There exist a number of variants to Hebbian learning. One such variant, Hebbian ABC introduces additional parameters to control the importance of each value in the weight change formula. Hebbian ABC uses the formula:

$$\Delta w_{ij} = \eta\left(A o_i o_j + B o_i + C o_j\right)$$

Hebbian learning is very simplistic, but has been shown to be useful in a few

applications, such as in associative Hopfield networks [4].

2.2.3. Temporal Difference

Temporal difference (TD) is a type of reinforcement learning that is particularly well suited to tasks of prediction and sequence learning.

TD learning operates off the principle that adjacent states are often correlated. Sutton [8] offers the following example: consider how a person might make predictions about the weather on some future day of the week, knowing only the current state of the weather. If it is Monday and you wanted to predict Friday's weather, you could use the current weather to help gauge the weather later in the week. On Tuesday you would have more information - both weather states from Monday and Tuesday, and your prediction for Friday would be potentially more accurate. As each day progresses, you have more and more information until you reach Friday and have the actual weather conditions. If this were a case of supervised learning, it would be necessary to know Friday's weather in order to train the weather prediction method, along with weather data from each other day of the week. However, TD learning attempts to use the information at each step, as it becomes available, without knowing the final outcome in advance. This is particularly advantageous when dealing with sequences. While supervised learning requires discrete pairs for training, temporal difference uses multiple states to predict an outcome. This aligns with the idea that predictions on future states are "not confirmed or disconfirmed all at once, but rather bit by bit" as additional information becomes available [8].

Temporal difference uses rewards to provide feedback to a function approximator. If

after a sequence of inputs a desirable state is achieved, a positive reward value is provided, and the parameters of the approximator updated based on that reward [9].

The gradient descent form of the TD(λ) algorithm is as follows [10]:

```
Initialize w randomly

Repeat

   e = 0

   s = initial state

   Repeat for each step:

       a = action selected for s by the agent

       Take action a, observe reward, r, and next state, s'

       δ = r + γV(s') - V(s)

       e = γλe + ▽w V (s)


       w = w + α δe

       s = s'

   until s is terminal
```

The error term (δ) for the TD algorithm is derived from the reward plus the difference between the output at the next time step ($V(s')$) and the output at the current timestep. For this calculation, the output at the next time step is multiplied by a parameter γ. This parameter controls how important future timesteps are to the update of weights; for $\gamma=0$

only the current timestep is considered; as γ increases so does the importance of considering future timesteps.

Why use the difference between the current and next outputs? Ideally, the return (reward) for the current timestep would be known prior to producing said output; this is the principle behind supervised learning. For reinforcement learning, only the reward (if any) is known after the output is observed and the state updated. The output of the next timestep is used as an approximation of the expected return for the current timestep. This is a process known as bootstrapping, the algorithm using its own future output to correct the current output.

Of course, in order to use future values, it is first necessary to observe them. As such, current outputs and weight gradients must be stored until the next step when the next output can be observed. Then, updates to the current weights are made using the newly observed output and the stored values.

The eligibility trace, e, is a matrix of values representing the sum of the gradients of the weights and the accumulation of past eligibility traces. Here, the γ parameter is used again as well as an additional parameter λ. This parameter provides a discount for past gradients. A value $\lambda=1$ produces weight changes as would a supervised learning method, treating state observations as inputs and resulting outcomes as training pairs. If $\lambda=0$, it is as a supervised method where the input is the current state and the desired output is the output of the following state, that is, s_t is the input and $V(s_{t+1})$ is the desired output. Thus values for λ between 0 and 1 produce updates between these two extremes.

Note that the term $\nabla_w V(s)$, the gradient of the output with respect to the weights, is calculated in the same manner as in the backpropagation algorithm when TD is used for neural networks. The difference is that backpropagation uses the gradient of the error with respect to the weights. However in TD, the error term is not available until the following time step. Thus, the partial derivative for the output with respect to the weights is stored, and incorporated into the weight change at the time the error value becomes available.

Using the gradient descent form of this algorithm lends itself naturally to neural networks. The network itself is the function approximator and its connections weights are the algorithm's parameters. States are used as the inputs, and the output *V(s)* is the network output.

2.3. Genetic Algorithms

As artificial neural nets were inspired by biology, so too have other techniques arisen from ideas found in nature. Genetic algorithms borrow concepts from genetics and evolution.

The core of this group of algorithms is based on the idea that successive generations of organisms evolve due to the mechanisms of natural selection, inheritance, mutation, and genetic crossover. In nature, an individual or population has traits that help or hinder survival. According to natural selection, beneficial traits help an individual (or

population) survive, and so those traits will be passed on to future generations. Traits are inherited from parent to offspring, but traits can arise in other ways. Existing genes may mutate somewhere in the reproductive process, potentially giving rise to traits previously unmanifested. Similarly, in complex organisms that reproduce sexually, a process called genetic crossover takes genes from two parents and recombines them. This can produce highly varied traits, as traits may be produced from single genes, or from combinations of genes.

Genetic algorithms use the same principles. Koza [11] provides an overview of genetic algorithms. Individuals are created using some form of genetic encoding, i.e., a simplified representation of the characteristics they have. Individuals are created as part of a population, and their performance on a task is measured against a fitness metric. Once evaluated, the individual is given a fitness score to use for comparison against the rest of the population. Individuals that perform well (or at least, better compared to others) will be used to create a new generation. This can happen in several ways. The individual can be reproduced in the next generation completely unchanged; it can be used as an asexual parent, where the offspring is generally the same, but with some small mutation; and it can be used as a sexual parent, where its genes are combined with those of another individual [11].

While the behavior of genetic algorithms is inspired by biological evolution, they can be used for a wide variety of problems. An example might be an agent that must navigate an environment; or it could as easily be a classifier that labels data samples. The genetic model is used to produce individuals, but how those individuals are evaluated (the type of

tasks) has no superficial bounds. One of the most promising applications of genetic algorithms is in the evolution of neural networks.

2.3.1. Neuro Evolution of Augmenting Topologies

A few genetic algorithms exist for evolving neural networks. One of the more popular algorithms is known as Neuro Evolution of Augmenting Topologies, or NEAT. NEAT evolves both the topology and the weights for a network. The algorithm is based on several key concepts: starting minimally; using mutation and crossover to produce new individuals; complexification; tracking innovations; and speciation. The description presented here is based on the original methodology proposed by Stanley and Miikkulainen [12].

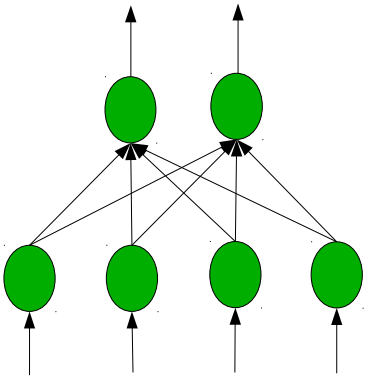
Genetic Encoding Methodology

First, it is important to understand the encoding NEAT uses to produce a network. A chromosome describes a complete network: how many input and output nodes are present; any hidden nodes that are present; and each connection that exists. Each gene on the chromosome represents one such element. A gene consists of the type of element (node type, or connection), the values relevant to that element, and whether it is enabled or disabled. In the case of nodes, the gene will indicate whether the node is an input, output, or hidden node. It will also store the value of a bias, if biases are stored on a per-node basis. For connections, the gene stores the incoming node and outgoing node, and the weight value. By including information regarding the topology of the network as well as specific weight values in the chromosome, both of these elements can be evolved

concurrently.

It is also important to note that hidden layers do not necessarily exist in NEAT networks in the same way they do in typical multilayer networks. Hidden nodes are added, but do not belong to any layer as such; they may be connected to nodes in the output layer, but may also be connected to other hidden nodes.

NEAT genes also store another critical piece of information: innovation markers. Each gene is assigned a unique innovation number at the time it is created. Innovation numbers are used to track compatibility when performing genetic crossover; this is explained in more detail later.



```
<chromosome id="84">
<neuron id="0" type="in" activation="linear"/>
<neuron id="1" type="in" activation="linear"/>
<neuron id="2" type="in" activation="linear"/>
<neuron id="3" type="in" activation="linear"/>
<neuron id="14" type="out" activation="linear"/>
<connection id="15" src-id="0" dest-id="14" weight="-0.11226618"/>
<connection id="16" src-id="1" dest-id="14" weight="0.30858034"/>
<connection id="17" src-id="2" dest-id="14" weight="0.14108664"/>
<connection id="18" src-id="3" dest-id="14" weight="0.037286043"/>
<neuron id="29" type="out" activation="linear"/>
<connection id="30" src-id="0" dest-id="29" weight="0.08622551"/>
<connection id="31" src-id="1" dest-id="29" weight="-0.45483726"/>
<connection id="32" src-id="2" dest-id="29" weight="0.48847902"/>
<connection id="33" src-id="3" dest-id="29" weight="0.35063475"/>
</chromosome>
```

Figure 2.5: Sample network topology and NEAT chromosome definition.

Minimizing Dimensionality through Complexification

One of the main concepts and advantages of NEAT is that it starts minimally and maintains minimal dimensionality. That is, it starts with a layer of input nodes and of output nodes and some connections between them - no hidden nodes are present at the start. As well, the initial networks may be fully connected, or they may only have one

connection for each node, a technique known as feature selection. As evolution progresses, additional structures are gradually added to the individual chromosomes; if they enhance fitness, individuals bearing those traits will produce offspring. If not, those individuals will ultimately be eliminated from the population. In this way, the size of the networks is kept as small as possible, and growth to the network only occurs when it provides a gain in fitness. This process is known as complexification.

What is the advantage to beginning minimally and maintaining the smallest possible networks? As mentioned earlier, one of the design problems facing multilayer networks is how many hidden nodes to include. NEAT starts without any hidden nodes. Through successive generations, individuals are mutated (or created via genetic crossover) to gradually add nodes, and add connections to and from those nodes. This process potentially finds an appropriate number of hidden nodes, and in practice, tends to produce much smaller networks than through hand-design and conventional training. The main advantage of a smaller network is reduced computation time.

Mutation

In order to evolve, each generation must produce new individuals. One method of doing this is through mutation. An individual from a previous generation is taken as the basis for an individual in the next, albeit with some change. Mutations include: adding a new connection between two previously unconnected nodes; perturbing a weight connection (or bias) value, that is, adjusting it up or down slightly; adding a new hidden node by inserting it along an existing connection; disabling an active gene or enabling a

previously disabled gene. Other mutations are possible. A common variant is to add nodes with alternative activation functions, such as Gaussian or linear, as opposed to sigmoidal activation.

Crossover

The other method NEAT uses to produce offspring is through genetic crossover. This is accomplished by taking some or all of the genes from one individual and combining them with another individual. However, in order to combine genes, individuals must be checked for compatibility. The problem can be illustrated using two simple networks with three hidden nodes; one has hidden nodes A, B and C, while the other has nodes C, B, and A (Figure 2.6). If crossover is performed using these two individuals, permutations exist that lose information contained within the parents' genes. One such permutation is a hidden node arrangement A, B, A; another is C, B, C. In these cases, some genetic information is lost while other information is duplicated, resulting in an individual that may or may not be viable. This is known as the competing conventions problem. Unfortunately, analyzing the networks to determine their compatibility is difficult, potentially computationally expensive, and often does not yield accurate results.

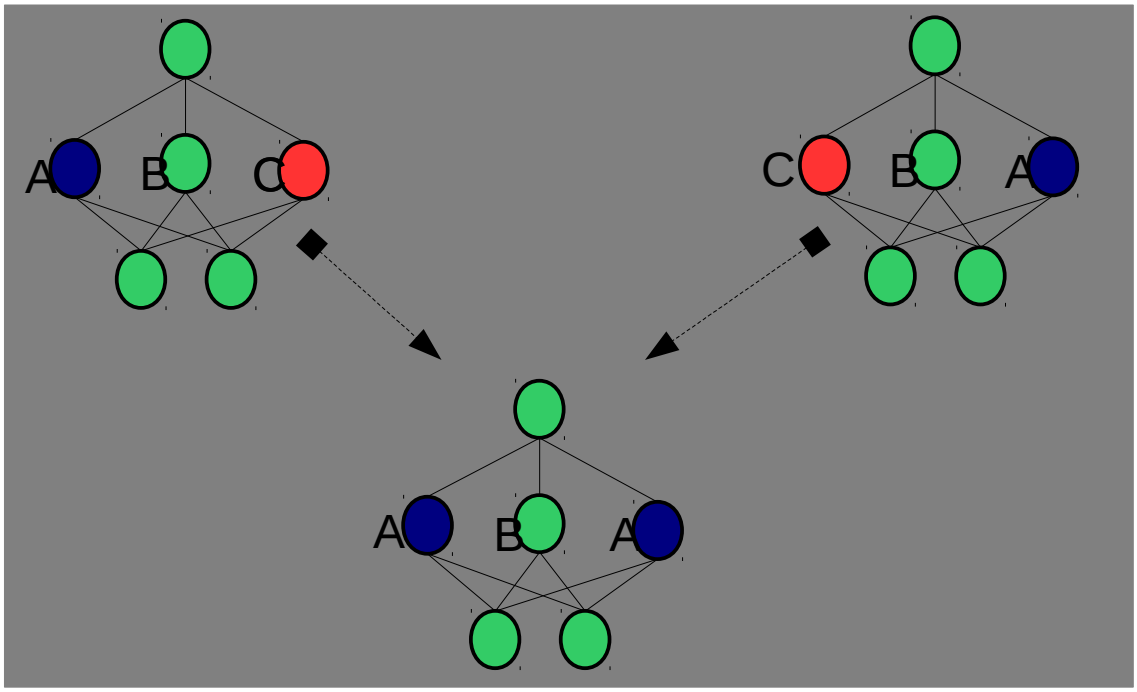


Figure 2.6: Two networks are recombined to include a duplicate of node A.

To address this issue, NEAT employs historical markings known as innovation numbers. As each gene is added to a chromosome, it is assigned a unique innovation number. When crossover occurs, chromosomes are aligned using matching innovation numbers, with unmatched genes being either disjoint (occurring in between innovations in the opposite parent) or excess (occurring beyond the end of the chromosome of the opposite parent). Figure 2.7 illustrates this process. The two parent chromosomes are aligned on matching genes 1, 2, 3, 4, 5. Parent 1 has disjoint gene 8; parent 2 has disjoint genes 6 and 7 and excess genes 9 and 10. In this case, the resulting offspring combines all genes from both parents (image from [13]).

In NEAT crossover, disjoint and excess genes are inherited from the more fit parent; if parents have equal fitness, disjoint and excess genes are inherited randomly from both

parents.

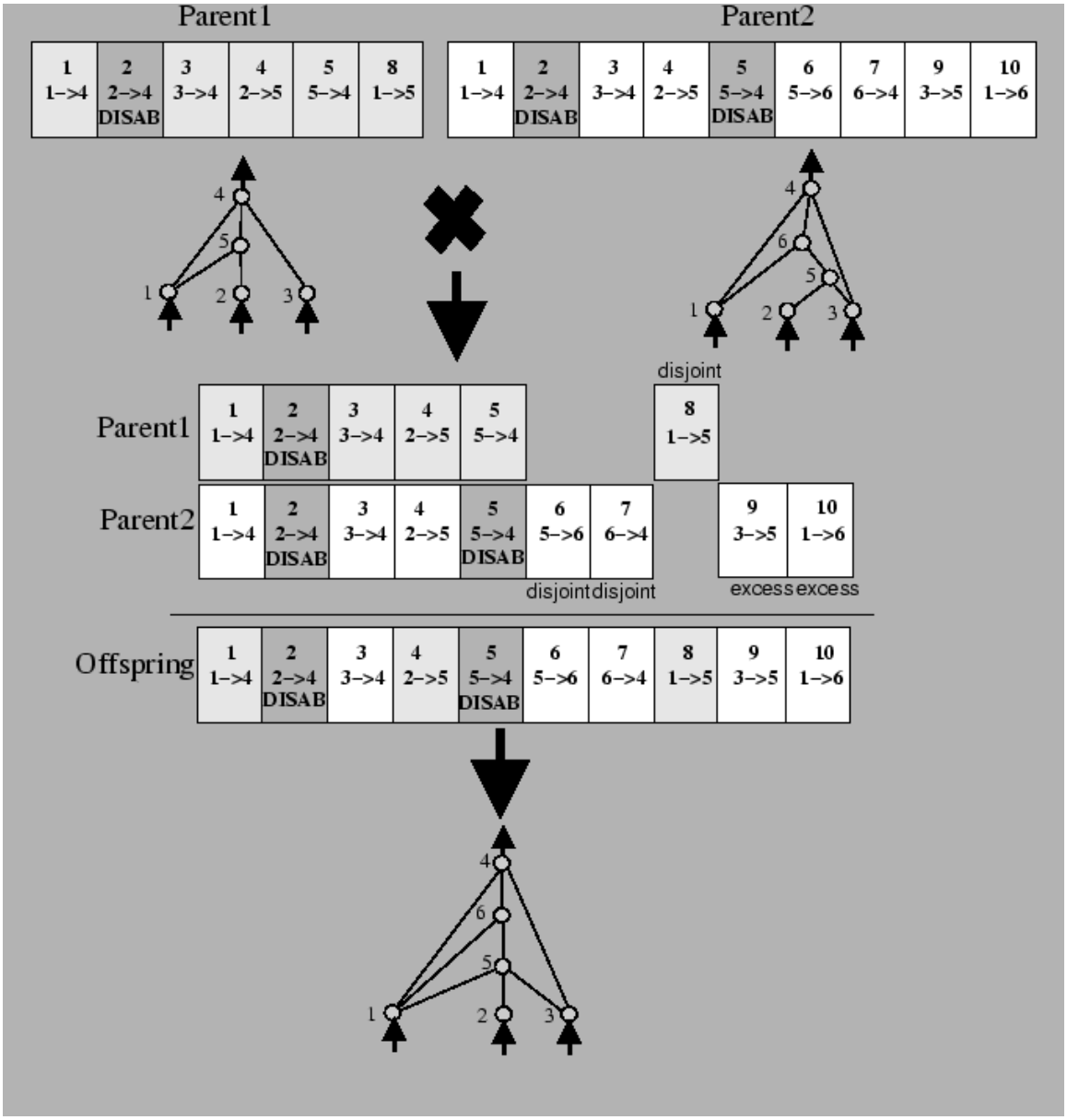


Figure 2.7: NEAT crossover.

Speciation

An additional key to NEAT is speciation. An individual competes within its own species rather than the population at large. When an individual is created, it is assigned to a species based on its topological similarity. The idea is that as new innovations are added to a network, they may need time to optimize; initially they may hurt the fitness of the individual, but may be quite competitive as successive generations experience additional mutation and crossover. By measuring them against the fitness of the species instead of the whole population, the time for optimization is allowed.

Individuals are assigned to a species by comparing the distance between the number of matching innovation markers in the individual with a randomly selected member of each species. The distance is calculated:

$$\delta = \frac{c_1 E}{N} + \frac{c_1 D}{N} + c_3 * \bar{W}$$

where *E* is the number of excess genes, *D* is the number of disjoint genes, *N* is the number of genes in the larger genome (normalized for size), $\bar{W}$ is the average weight differences of the matching genes, and *c1*, *c2*, and *c3* are parameters controlling the significance of each measure.

The individual is placed into the first species of the previous generation where the distance δ is less than a species compatibility threshold parameter. If no species from the previous generation is selected, a new one is created, and the individual placed in it. The compatibility threshold may be adjusted during evolution to constrain the number of

species, if too many species are created, or not enough are surviving.

The number of offspring that are carried forward from each species is constrained to prevent the population from growing too large, and to prevent any one species from taking over the entire population. This is accomplished through explicit fitness sharing, where individuals in a species must share the fitness niche of the species. An adjusted fitness value is calculated for each individual according to its topological distance δ from every other individual in the population:

$$f_i' = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i, j))}$$

where f_i is the individuals fitness, n is the number of individuals in the population, and j is another member of the population. The function $sh(x)$ is 0 when the distance is above the compatibility threshold, otherwise 1. This effectively sums the fitness only for individuals in the same species as individual i. Each species is then allowed a number of offspring proportional to the sum of its members adjusted fitness'. Within a species, the lowest performing members of the population are not used as parents, and thus eliminated.

The result is that structural innovation is protected when it is first introduced, and allowed some time to optimize.

2.3.2. HyperNEAT

The NEAT algorithm has been very successful and has inspired many variants and extensions. Gauci and Stanley [14] offer a very promising extension called HyperNEAT.

HyperNEAT was introduced as a way to incorporate geometry into the evolution of networks. Many real world problems have inherent geometry. For example, robots usually have an array of sensors at various positions. Also, in board games such as checkers or chess, the boards have a high degree of symmetry. While it is entirely possible to evolve networks that exploit such geometry, there is nothing inherent in the NEAT algorithm that can capture it; geometric relationships must be discovered individually through the course of evolution. Consider the robot example: in a very simple case, a network may be evolved to control the robot to move in the direction of the strongest signal received from its sensors. Each sensor might feed its value into an input node on the network, and network outputs control the direction the robot travels. In this topology the input values all represent the same type of information, a specific type of signal. Yet, NEAT does not take advantage of this; connections from the inputs are effectively evolved separately. Very likely, connection weights will have converged to similar values at the end of a successful evolution run, but NEAT does not necessarily arrive at those values in the most efficient way. On the other hand, it is possible for HyperNEAT to learn a particular policy that is applied to multiple weight values simultaneously.

To capture geometry, HyperNEAT uses an indirect encoding for describing networks.

Whereas NEAT has a 1:1 correspondence between a gene and a structure in the network, HyperNEAT uses genes to encode a pattern of connectivity in a network. This indirect encoding is accomplished through a Compositional Pattern Producing Network (CPPN) designed to represent patterns of regularity such as symmetry, repetition, and repetition with variation.

The CPPN is a special type of neural network; each node in the network can potentially use any type of function for activation. Common activation functions include sigmoidal, Gaussian, absolute value, linear, step functions, with other possibilities as well. The CPPN is evolved using an extension to the NEAT algorithm that permits chromosomes to represent the expanded set of node activation functions. And as with any multilayer neural network, CPPNs are capable of approximating any function in a given n-dimensional space.

The use of the CPPN is to encode spacial patterns with regularity. The key to how HyperNEAT functions lies in how those patterns are captured. Spatial patterns in 2n-dimensional space are isomorphic to connectivity patterns in n-dimensional space. That is, points in 4-dimensional space may be represented by a set of four coordinates. Those four coordinates may also be represented by a connection between two points in 2-dimensional planes. The origin of the name HyperNEAT comes from the idea that a CPPN paints a pattern on the inside surface of a hypercube, a 4-dimensional cube. When represented as a pattern of connectivity in 2-dimensional space, that pattern effectively forms a neural network unto itself. This emergent network is called a substrate.

The substrate is designed to represent the geometric structure of a problem, and the CPPN fills in its connection values appropriately. Consider the arrangement of nodes in a given layer of the network. Conventionally, nodes are laid out in a one dimensional line. This makes it simple to implement. Alternatively, nodes could be arranged according to some geometrically relevant schema, in particular one that resembles the geometry of the problem. Recall the robot example from earlier. A simple square shaped robot might have one sensor on each of the four sides of its body. These sensors could be connected to a one dimensional row of network inputs for use in NEAT and other techniques. However, since the sensors in the real world exist on a 2-dimensional plane, that layout could be mimicked in the input layer itself, having a two-by-two grid of nodes. This is the idea behind laying out the substrate to match problem geometry.

Once the topology of the substrate is defined, the CPPN is queried for the value of each connection in the substrate. The CPPN is provided the coordinates of each node as input, and the output of the CPPN is the weight value for the connection between those nodes. Figure 2.8 illustrates this: the substrate has two 2-dimensional layers of nodes. The coordinates of two connecting nodes (the node at [1,0] and at [1,1]) are fed as input to a CPPN. The output of the CPPN is used as the weight value for that connection.

The pattern encoded by the CPPN manifests continuous, regular weight patterns in the substrate, and inherently captures the geometry of the problem.

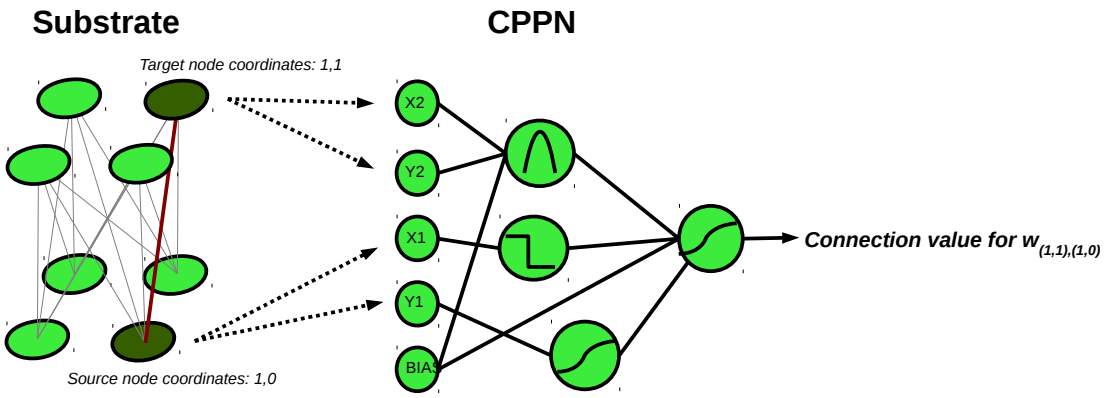


Figure 2.8: HyperNEAT substrate and CPPN.

Additionally, other geometric information may be provided as input to the CPPN. The layer of either the source or target node may be provided, as well as information such as the distance between the nodes (distance between coordinates), the angles between coordinates. Another common addition is the use of a bias input node. In addition to outputting weight values, the CPPN may also be used to output additional information such as node bias values for the substrate or other parameters; this technique will be discussed in more detail in the application chapter.

An advantage to HyperNEAT is that it is possible to represent a problem in different ways, by choosing different designs for the substrate. One design might layout multiple sets of 2-dimensional inputs in a single 2-dimensional plane; another design might stack the inputs in 3 dimensions. It has been found that although HyperNEAT can successfully evolve networks with arbitrary substrate geometry, it tends to perform best when the substrate matches human intuition about the geometry of a problem [15]. Still, this affords HyperNEAT some flexibility in substrate design.

HyperNEAT can be further extended to handle subsections of the substrate for varied functionality. Key research in this area has centered around producing a large substrate that represents an entire team of agents, with subsections used by individual members of the team, and on having different subsections of the substrate used by an agent for different types of tasks [19],[20].

3. RELATED RESEARCH

Our research builds on previous research that focuses on the effect of learning on evolution, the HyperNEAT algorithm, and extensions to HyperNEAT.

3.1. The Influence of Learning on Evolution

Parisi and Nolfi [16] explore the interaction between learning and evolution. They make the claim that learning will influence evolution and conversely that evolution will influence learning.

They argue that with evolution alone, genotypes can only be selected based on their current position in the fitness surface. They cite an example where two individuals are in different locations on the fitness surface, but have the same fitness value. It would seem that producing offspring from one of these would be as good as the other. But, consider that the surface area around on of those individuals is significantly better than the other. It would be preferable to select the individual with the more favorable surrounding areas, however this is not evident until an offspring is actually produced (where, the offspring is in a slightly different area on the fitness surface). However, by introducing learning during an individual's lifetime, the thought is that some of the fitness surface around an individual is also explored, resulting in the fitness measure being based not solely off the individuals starting position, but also some function of the area that is explored. Thus over time, learning should allow the selection of better individuals.

Parisi and Nolfi test their hypothesis using a simulated environment with network controlled agents that must gather food scattered randomly. Fitness is a measurement of

how much food is gathered during a trial. The network architecture is fixed at the start and offspring are only produced through creating individuals with mutations of the parents' connection weights. Furthermore, during an organism's "life", it is trained to predict how the perceived position of the food changes with respect to its movement actions. Note that the learning task is different than the evolutionary task. The chances that an individual will reproduce are based solely on its performance on the evolutionary task, and not the learning task. Based on their simulation, they observe that learning has positive influence on evolution, with trained networks outperforming untrained ones. The figure below compares the results of trained versus untrained networks.

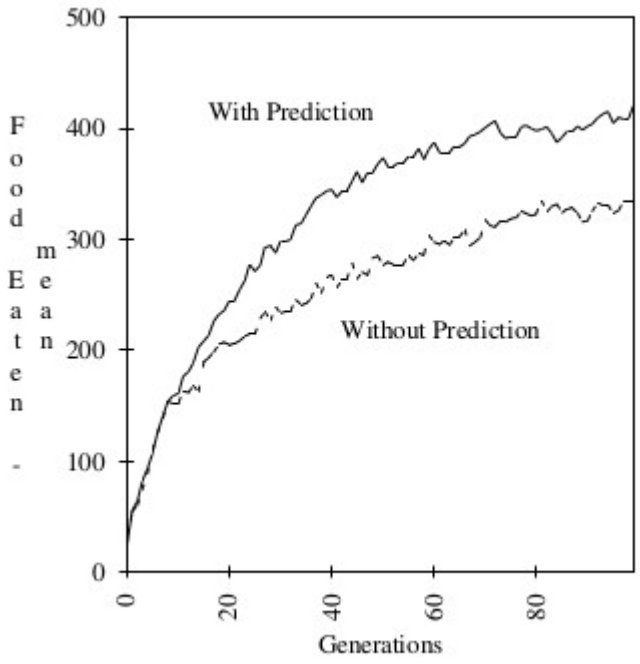


Figure 3.1: Trained versus untrained networks.

Parisi and Nolfi note that since the evolutionary task and the learning task are not quite the same, when an individual learns, the knowledge acquired may not translate to

evolutionary fitness; in fact it is possible that the learning lowers the individual's fitness.

The result is that over many generations, evolution will naturally select individuals in a position such that what they learn in their lifetime correlates with evolutionary fitness. They demonstrate this with a simulation in which all individuals are trained to predict food location. The results show that individuals in the first generation show no improvement in their ability to capture food during their lifetime. However, individuals in later generations do increase their ability to find food over the course of their lifetime. Furthermore, all individuals show similar responses to training epochs, that is, their error decreases roughly the same amount over the course of training. The conclusion can be made that while learning the prediction task does not directly increase food collection ability (as indicated by the results from the earliest generations), it does serve to guide evolution to select individuals where learning does result in an increase in fitness. In effect, evolution selects for learning ability. The figure below shows the simulation results: the amount of food collected over the training epochs of an individuals life at various generations.

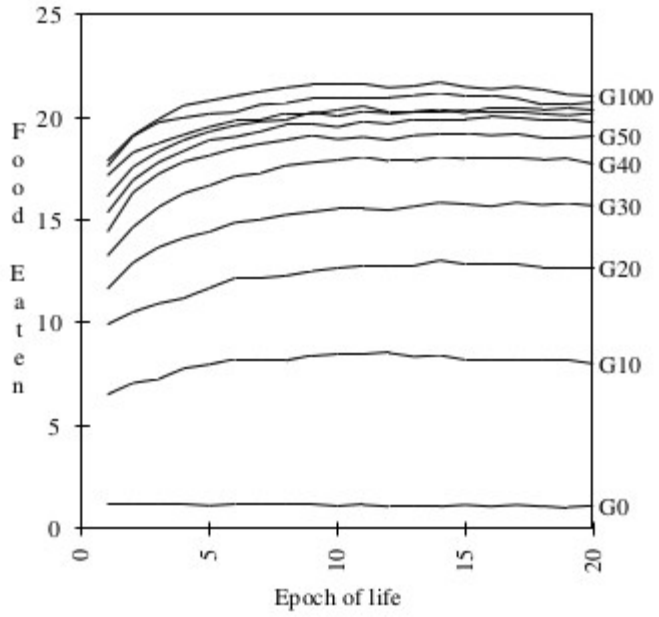


Figure 3.2: Food collected over an individual's life.

Similarities to our research:

- Both study the effect of learning on evolution.

Our research is distinguished by:

- Studying the effect of learning on a modern evolutionary algorithm (HyperNEAT).
- Using multiple different learning approaches.
- Focus is given to how to use HyperNEAT extensions to aid online learning.
- Based on our results, we conclude that while online learning can improve the performance of evolved individuals, this effect is not universal; there are some

cases where online learning inhibits evolution.

3.2. Culling and Teaching in Neuro-evolution

McQuesten and Miikkulainen [17] explore the combination of genetic algorithms and supervised training. Their thesis holds that evolving populations contain a "culture", that is, information regarding the behaviors exhibited by population members. They propose that this culture can be used to accelerate the evolution of a successful individual in two ways: 1) culling large litters and 2) teaching, that is, training offspring with their parents.

A pole-balancing task is used to evaluate performance of each approach. Their implementation uses a simple genetic algorithm based on genetic crossover.

The first technique, culling, attempts to eliminate poor performing individuals from the population. The rationale is that within a set of offspring, both in nature and in computational evolution, may individuals are unfit. Their first experiment uses a "perfect oracle" to select the best offspring for evaluation and subsequent reproduction. The perfect oracle in this experiment selects an offspring by executing a full evaluation using the fitness function. This is useful to demonstrate that such a selector will in fact produce a successful individual much faster, but is not practical since it requires the full computation time of an evaluation. Their initial results suggest that the perfect selector does in fact improve performance of the genetic algorithm: all runs using the perfect oracl were successful and on average offspring where 62% as fit as their parents, up from 30%, with 3% being twice as fit as the parent. This then leads to the question of whether efficiency can still be improved with a less than perfect oracle.

Their goal is to introduce a method that can cull offspring without full fitness evaluations but still recognize poor performing ones with reasonable probability. Their approach is to quiz each offspring and grade them using the population's knowledge. The methodology is simple: a set of "questions" (an input vector to the network) is randomly chosen and presented to each offspring, and their response is compared to that of the parent. The values in the input vector are generated in the range of 0.45 to 0.55. The rationale is that this is merely a qualifying exam, and neither the parent nor offspring would output a perfect response to extreme values.

Their second proposal is to use parents to teach offspring. This is relatively straightforward: an offspring is trained with backpropagation using the parent's output as the expected response, using Euclidean distance between the parent and offspring output as the error signal.

It is noted that excessive training would lead to an offspring emulating its parent too closely, thus inhibiting evolutionary progress. Since the goal here is only to incrementally improve the offspring before it is evaluated with the more computationally expensive fitness evaluation, the set of training examples is constrained, and only given a single training iteration.

Based on positive results from each technique, McQuesten and Miikkulainen attempt a third set of experiments that combines the two techniques. In this experiment, a set of offspring are generated, then trained using a set of twenty test cases. Then the offspring with the lowest training error is selected to enter the population.

3.2.1. Results

The culling experiment was able to produce a successful individual 28% more frequently than the genetic algorithm alone and with only 55% of the number of evaluations. The teaching experiment used only 41% of the evaluations and was successful 98% of the time. The combination of these techniques had a 100% success rate and used only 25% of the evaluations of the base genetic algorithm.

McQuesten and Miikkulainen also observe an interesting phenomenon in the teaching and combination trials: the networks produced are actually not good for the task, and require training to be successful. That is, optimal weights are not evolved, but rather, networks are evolved that have good teaching capability.

Similarities to our research:

- Training is used during evolution in order to improve performance.
- Networks evolved with training do not necessarily perform well until they receive training.

Our research is distinguished by:

- Individuals are trained with heuristic supervision, or reinforcement upon arriving in reward states, as opposed to being trained by their parents.
- In the case of intermittent training (experiments 2.3), individuals are trained with experiences from their own lifetimes, as opposed to being trained by their parents.

- No focus is placed on attempting to cull individuals.

- Several different kinds of online training are used, and several other measures are taken to enhance how they interact with the HyperNEAT algorithm.

3.3. Evolving Adaptive Neural Networks with and without Adaptive Synapses

Stanley, Bryant, and Miikkulainen [18] attempt an experiment to answer the questions: 1) are plastic synapses necessary for networks to adapt to changing environments, and 2) does the addition of local learning rules aid the network's ability to adapt, when necessary? To this end, they setup an experimental domain that should require a policy change during the network's lifetime, and evolved network controllers with local learning rules and controllers with only fixed weight connections. All networks were also capable of having recurrent connections.

They used the NeuroEvolution of Augmenting Topologies (NEAT) method to evolve their controllers, but extended it to support the evolution of local Hebbian learning rules for individual connections in the network. The addition of local learning rules allows the connection weights to change over the network's lifetime, based on the evolved rules at each connection.

They evolved a single general learning rule for both excitatory and inhibitory connections that uses only two parameters. Excitatory connections are updated with the formula:

$$\Delta w = n_1 (W - w) x y + n_2 W x (y - 1.0)$$

where W is the max value of all connections, w is the value of the current connection, n_1 is the Hebbian learning rate and n_2 is the decay rate, which controls how rapidly the connection weakens when the presynaptic node does not affect the postsynaptic node. Inhibitory connections are updated:

$$\Delta w = -n_1(W-w)x\,y + n_2(W-w)x(1.0-y)$$

The term n_1 is negative because correlated activation implies that the connection does not have an inhibitory effect. The term n_2 strengthens the connection when the input is high and the output is low, increasing the contribution of the inhibitory connection.

This rule is incorporated into the NEAT algorithm. In the evolving networks, each connection has the parameters n_1 and n_2 in addition to its own weight. While it might be possible to use separate learning rules, this would greatly increase the parameter space. This system ensures that: the same rule can be used by many genes; the number of learning parameters in the genome does not grow as the genome grows; the adaptation rules can be adjusted separately from the connection genes.

A foraging domain is constructed where a network-controlled agent must eat food. In the environment, there are two types of food, A and B, and food can either be edible or poison; the type of food (A or B) is independent of whether it is poisonous. The agent has two sets of five rangefinder sensors in an array on its front; one set detects food A, one detects food B. The agent also has a pleasure sensor and a pain sensor. The former is activated when edible food is consumed; the latter when poisonous food is consumed. For a given trial, food is entirely of a single type (A or B), and all are poisonous or all edible. Thus it is necessary for the agent to consume at least one piece of food in order to

determine if it should continue feeding for the duration of the trial.

Multiple trials are used to evaluate the network: two trials with edible A items, two with edible B items, two with poisonous A items, and two with poisonous B items. Before the start of each trial, networks were reset: internal activations were set to zero and connection weights were reset to the values defined in their genome. The collection of edible food items increased the fitness of the network, while the collection of poisonous items decreased fitness; a score of 60 held over multiple generations is considered a solution.

3.3.1. Results

For the trials with fixed-weight networks, all five runs consistently scored 60 or above prior to reaching the cut off of 350 generations. The best run found a solution by the 250th generation.

The networks evolved with local learning rules also solved the task, but only in three of the five runs, with the cut off at 500 generations. The best run found a solution by the 350th generation. This demonstrates that it is possible to encode dynamic policies with Hebbian learning.

The fixed-weight networks with recurrent connections outperformed the networks evolved with local learning rules; this was an unexpected result. The authors of the research sought to understand why that was the case. One of the networks with fixed weight connections evolved a rather simple policy of moving through empty space, to a

wall after consuming poisonous food. This was an effect of having a strong recurrent connection on its left turn output node that was muted by a strong inhibitory connection from the pain receptor. This caused the robot to turn to the right when faced with poisonous food, heading eventually to a wall, and thus preventing the robot from consuming any more poisonous food during that trial. Other fixed-weight recurrent networks that were successful evolved similar "trick" behaviors that prevented them from consuming food once it was known to be poisonous.

Agents evolved with learning rules tended to rely on the functioning of the network as a whole, rather than key signals in only a couple of nodes and connections. By removing hidden nodes from the fixed weight networks and from the adaptive networks it was observed that the fixed weight networks were generally still able to complete the task, but the adaptive networks could not, suggesting greater reliance on complex internal mechanisms. This explains why networks with fixed weights and recurrent connections found solutions more quickly and easily.

The conclusion they arrive at is that while adaptive networks can be successfully used to find solutions in domains that require policy changes, recurrency may be sufficient for many tasks due to the smaller search space.

Similarities to our research:

- Networks are evolved that can have their weights changed during their lifetimes.
- An evolutionary algorithm is extended to support evolution of additional

parameters.

- Both conclude that the larger search space represented by the evolution of extended parameters means that this methodology may not perform as well as simpler methods.

Our research is distinguished by:

- The primary focus of our research is on the effects of different applications of online learning during evolution, as opposed to explicitly evolving rules for policy change.

3.4. Generative Encoding for Multiagent Learning

In this paper, D'Ambrosio and Stanley [19] propose a new methodology for multiagent learning. They implemented a version of the HyperNEAT algorithm that is modified to encode policy for heterogeneous agents. They claim that this is beneficial in coordinating a team of agents due to the use of indirect encodings and information reuse.

They note that a good deal of research has been performed in the area of multiagent learning, with varying levels of success. Some past approaches involve multiagent reinforcement, where cooperative states and agent actions are rewarded. These solutions are typically unable to incorporate complete information from the whole team, or if they do, they scale poorly due to dimensionality increasing exponentially with each agent that is added. Another approach, cooperative evolution assigns fitness to agents on their ability to execute tasks alongside other evolving agents. Historically, these approaches

tend to either produce agents with a high degree of specialization, but few shared skills, or the opposite: a good shared skill-set, but poor specialization. Still another approach uses global communication, and uses a single large genome to control all agents. This too increases dimensionality with each agent added to the team.

D'Ambrosio and Stanley propose a different approach: Multiagent HyperNEAT.

For a homogeneous team of agents, a single controller is copied for each agent. In heterogeneous teams, each agent has its own distinct controller. To augment HyperNEAT for heterogeneous teams, each agent's network controller is placed on the HyperNEAT substrate; in this way the CPPN produces the connection weights for the networks of the entire team. Further, two special nodes are added to the CPPN. These provide the horizontal coordinate frame of the substrate, that is, they tell the CPPN which agent's network to which a connection belongs. This allows the CPPN to encode both shared patterns within agents (by referring to the coordinate frame) and patterns that differ between agents.

Their proposed algorithm also allows for the possibility of "seeding" a team, that is, evolving an agent to have certain abilities, then using that controller as the basis for the entire team. It is possible to use this in a heterogeneous team by adding the coordinate frame nodes to the seed controller along the existing *x* coordinate connections - such that the connection between the existing *x* inputs and the coordinate frame nodes has a weight of 1.0 and the coordinate frame nodes connect to the output nodes with the same weight values as the *x* nodes had originally. This allows the initial behavior of each agent on the

team to match the seeded controller, but to specialize as evolution takes place.

D'Ambrosio and Stanley set up a predator-prey experiment to test the effectiveness of their approach. In this simulation, agents are predators and cannot see one another. Prey will run from predators, making it possible for one predator to knock a fleeing prey off the path of another pursuing predator. Thus predators must learn consistent roles that work with those of their allies. However, the predators will still need a basic skill-set in order to locate and pursue prey. Since HyperNEAT creates the agents from a single CPPN, it may be able to balance these elements.

Each predator has a set of five range-finding sensors, spanning a 180 degree arc, that detect prey inside 300 units. Predators must capture prey by placing themselves facing a prey within 25 units. Predators can turn 36 degrees and move up to 5 units forward for each timestep. Prey do not move until a predator is within 50 units of them, at which point they flee in the opposite direction of the closest predator at a rate of 5 units per timestep. Because the movement rates of predators and prey are the same, predators cannot capture prey through pursuit alone and must work together to trap prey. The predators start in a line, 100 units apart facing a prey formation. Since predators cannot see one another, they must infer the state of the rest of the team, and learn *a priori* strategies to capturing pray.

Each trial is 100 timesteps. At the end of each trial the team is scored with this formula:

$$score = 10000P + (1000-t)$$

where P is the number of prey captured and t is the time taken; if no prey are captured, t is set to 1000.

Four different types of teams are used: heterogeneous and homogeneous teams, one each with and without a seed. For those teams that use a seed, an agent was evolved that was effective at chasing prey.

Several prey formations are used for training agents: triangle, diamond and square. Each team is trained on two variations of one of the three formations, encouraging specialization to the specific formation, but providing some generalization to its variants.

3.4.1. Results

Performance is measured as the time remaining after all prey are captured, averaged across each formation variant. Each trial runs for 5000 timesteps. The maximum score is 5000, the minimum is 0, in the case that no prey were captured. This method measures task completeness but is distinct from the fitness score; in the case that a controller solves one training example but not other, it may need to sacrifice some performance on the solved formations in order to gain on the others.

The most successful approach was the seeded heterogeneous team, which outperformed all teams across all configurations. The seeded heterogeneous team was only slightly better than the unseeded heterogeneous team, and only on the square and diamond formations, but outperformed the homogeneous approaches on all formations.

In every formation the unseeded heterogeneous team performed second best, followed by

the seeded homogeneous team. Both homogeneous team types could not solve all training formations consistently.

The solutions were further tested for their generalization ability. Each was presented with seven variants of each training formation. For each team type, only the most general solutions were used. They found that generalization performance was strongly correlated to training performance; thus, heterogeneous teams tended to generalize the best. Only the seeded heterogeneous teams were able to solve all presented scenarios.

D'Ambrosio and Stanley observed the behaviors of the homogeneous and heterogeneous teams. The top performing homogeneous teams used a strategy of each predator chasing a prey in a circular pattern. When those circles overlapped, one predator would break off from its own chase and pin the other predator's prey. This strategy or similar was used by all the top performing homogeneous teams.

Heterogeneous teams used varying strategies. Some searched in packs of two or three agents, and coordinated to approach prey from opposite sides. Some "corralled" prey into a tightly packed cluster to capture them. Another strategy was for some of the predators to form a fence while other predators chased prey into them.

This demonstrates multiagent HyperNEAT's ability to represent team behavior as variations on a theme, and encoded as a single genome. This allows key skills to be shared across a team, and not need to be rediscovered for each agent. Further, agents can be specialized, something that is not possible in homogeneous teams.

D'Ambrosio and Stanley conclude that their new approach can produce a heterogeneous team of agents from a single strong seed that can learn genuinely cooperative behavior with specialized roles.

Similarities to our research:

- Both use HyperNEAT, with CPPN extensions to allow multiple subnets to be encoded on the substrate.
- Both use HyperNEAT substrates to control a team of agents.
- The tasks to be carried out by the agents require effective teamwork for success.

Our research is distinguished by:

- Only a homogeneous team is used in our experiments.
- The primary focus of our research is on the effects of different applications of online learning during evolution.

3.5. Task Switching in Multirobot Learning through Indirect Encoding

In this paper by D'Ambrosio, Risi, and Stanley [20] a team of robots is trained to patrol an environment. The robots are trained to cooperate and perform complimentary tasks, something that has been tricky in previous approaches. The goal of the research is to establish the viability of using an extended version of HyperNEAT to capture both team policy geometry as well as situational policies.

This research extends previous work where HyperNEAT was used to generate networks for each member of a team. The advantage is that all agents learn a general set of behaviors and strategies, but based on their position in the team or environment, may evolve slightly varied behaviors. This was shown to be a very successful technique.

The extension this research proposes is to also incorporate situational context in the evolved policies. When agents must perform multiple tasks, it is difficult to know when to switch tasks and requires a more complicated (and thus more difficult to evolve) policy in order to encompass all behaviors. This approach extends the team policy approach by generating multiple policies (networks) per agent.

For comparison, teams are evolved that use the standard team policy and the situational policy. As well, this research goes on to test the evolved policy networks by using them to control real robots in a physical environment.

In the standard team policy, the substrate consists of a "stack" of networks, one for each agent on the team. Using the HyperNEAT algorithm, the weights of the networks in the substrate are filled in using the Compositional Pattern Producing Network. A new input, the "*z*" coordinate is added to the CPPN representing the coordinate of the network in the substrate.

For situational policy geometry, agents must switch policies depending on their state. When used with HyperNEAT there is the potential to exploit similarities amongst subtasks, and make each subtask policy simpler to evolve. The team policy substrate is extended to accommodate this by adding an additional dimension for the task. This "s"

(for situation) dimension is added as an input to the CPPN. Thus there are multiple networks generated for each agent - one for each type of situation or task. In this research, three robots are used and there are two tasks that must be completed, resulting in a total of six networks in the substrate - one per robot per task.

The overriding task the robots must learn is to patrol an environment and return to a home position when complete. This is broken into two subtasks - patrolling and returning. This is where the two task networks come into play, one for each of the subtasks. In these tasks, the robots must navigate a "plus" shaped environment, starting from one hallway and navigating through each of the three branches of the plus before returning home. The robots do not communicate with one another, so it is necessary for them to learn an *a priori* strategy, optimally where each robot chooses a different hallway to explore. This requires cooperation to ensure maximal coverage of the environment with minimal overlap, and to avoid collisions.

For training, a simulation is used that mimics the dimensions of the environment and the sensory and motor capabilities of the real robots. Each robot is equipped with six infrared sensors that serve as inputs to the robot, indicating the proximity to obstacles. Obstacles can be walls of the environment or other robots; no distinction is made between the two. A signal is used to indicate when the robots should return to their home position. Each robot can take one of three actions based on the output of its network: move forward, turn left, or turn right.

For the networks evolved using standard team policy, an additional input is used to

indicate the presence of the return signal. For those that use the situational policy, the robot switches to the situational network.

Fitness for evolution is recorded for the team as a whole, based on two criteria. The first is to minimize the distance between any robot and the end of the halls. The second applies to when the return signal is initiated, and measures the distance from the bots to their respective home positions.

Fifteen evolution runs were executed for each type of policy. In these runs a successful solution is one where robots reach the end of all hallways in the plus, and return to their home positions when the return signal is given. Evolution is not stopped if a successful solution is found, allowing for multiple solutions per run.

For the robots using the standard team policy, only three successful solutions were found. For those that used the situational policy, every run resulted in at least one solution, and did so in an average of 264.6 generations.

Solutions were tested for generality by subjecting the simulating bots to sensor noise, random forced turns, and small changes to initial position and directional orientation. Of the ones using situational policy, the five most general were selected for real-world testing.

The real world testing involved using the evolved networks to control the real-world versions of the simulated robots. An environment was constructed to match the simulated plus-shaped environment. As a further test of generality, an asymmetrical plus was also

constructed where hallways were of different lengths and in slightly different positions. None of the evolution or prior tests had incorporated the asymmetrical plus.

All five of the teams using situational policy were able to successfully traverse both the original plus and the asymmetrical plus, demonstrating good generality. Of the three teams using the standard policy, only two were able to successfully navigate both environments.

Based on these results, the authors concluded that using situational team policy geometry produces solutions more frequently and more general solutions than does standard team policy.

Similarities to our research:

- Both use HyperNEAT, with CPPN extensions to allow multiple subnets to be encoded on the substrate.
- Both use subnets of HyperNEAT substrates for executing distinct tasks.
- The tasks to be carried out by the agents require effective teamwork for success.

Our research is distinguished by:

- Only a homogeneous team is used in our experiments.
- The task to be performed is different.
- The primary focus of our research is on the effects of different applications of

online learning during evolution.

3.6. Directional Communication in Evolved Multiagent Teams

Pugh, Goodell, and Stanley [21] seek to empirically establish the importance of directional communication in evolving cooperative multiagent teams.

They observe that a good deal of research has been dedicated to the use of communication in cooperative tasks. Some of this research explores directional reception, where the receiver is aware of the speaker's relative position. As well, much research has also been devoted to communicating agents that are unaware of the speaker's position.

Pugh, Goodell, and Stanley pose the hypothesis that knowing the relative location of the speaker is vital in tasks involving group coordination, and further that relative position should be implicit in the communication (via directional reception), so that evolutionary effort is not used in explicitly encoding position into an evolving language.

To test their hypothesis, they built out a simulation using a team of five agents in a bounded room, that must collect as much food as possible in given time. Only one piece of food is present at a time, and when it is collected, a new one is placed in the room randomly. Food is only collected when it is touched by three agents. This serves to encourage communication, in order to collect food efficiently; the optimal behavior would be for agents to produce a "come here" signal when food is discovered. This is more difficult when directional reception is not a part of the communication; this is

because additional language must be developed to describe the location of "here".

Agents are controlled with HyperNEAT substrates. The HyperNEAT algorithm is chosen for its ability to capture the geometry of a domain.

Agents have various sensors for reading the environment; these sensors are the same for all agents across all experiments. Agents can sense food within a limited radius using a set of five equal-size pie-slice sensors that are situated across the forward 180 degrees of vision. Five wall sensors are arranged in front of each agent in a similar fashion. Agents can detect other agents through a set of ten pie-slice sensors that completely surround the agent; these have no range limitation. Each agent can move forward, and can turn left or right.

Three schemes of communication are implemented, along with a control scheme that uses no communication (NoCom). The three schemes are DirCom, OneBit, and FiveBit.

DirCom transmits on a single channel, controlled by an output neuron, with values in the range of 0.1 to 1.0. These agents can "hear" the transmission via a set of ten input sensors that are arranged to correspond to the direction of the transmission.

OneBit agents transmit over a single channel, controlled by an output neuron. They do not hear values directionally, but rather receive inputs through one of five input sensors, corresponding to the other agents. Agents do not sense their own transmission.

FiveBit agents may transmit over five channels, controlled by five output neurons. Like OneBit agents, they do not hear values directionally, but have a set of 25 input neurons

that receive transmissions (a set of five for each agent). Again, they do not hear their own transmission. This schema allows for the evolution of different "words" with which to communicate.

The performance of each team is averaged over 20 trials. Each trial consists of 2000 timesteps. Teams may collect up to a maximum of ten food items. Scoring is as follows: 10 points are awarded when an agent sees the food; 40 points are awarded when the food is collected and from 0 to 50 time-dependent points are awarded based on how quickly the food is collected, to provide a more smooth fitness gradient.

Evolution is run for 1,000 generations. The training phase consists of 20 evolution runs for each communication scheme. After training, the champions are given a separate, more stable test: the total number of food items collected in 5,000 time ticks, averaged over 10,000 trials. In this test, agents are not limited to collecting only 10 food items.

3.6.1. Results

The directional communication scheme significantly outperforms all others ($p < 0.05$; Student's t-test).

There is no significant difference between the OneBit, FiveBit, and NoCom schemes.

Most NoCom teams use random wandering strategy, and achieve an average of 10 food items collected. The best NoCom team used coordinated search strategy, where 4 of the agents followed a single lead agent in a wall search; this team achieved an average of 12.3 food items.

The best OneBit teams exhibited a similar strategy as the best NoCom team, receiving scores of 12.7 and 12.6.

OneBit teams tended to emit nonsensical signals (they bore no significant correlation to events in the simulation) or emitted no communication at all.

The best FiveBit team succeeded in evolving a "come here" signal, earning it a score of 14.4. On this team agents spread out, and when an agent sent a signal, the receiving agents would begin to seek other nearby agents, ultimately leading to the collection of the food. The behavior demonstrates that the agents are unaware of the location of the transmitting agent, since often the four remaining agents will cluster together before locating the fifth. This strategy was only achieved in one of the FiveBit runs, suggesting its difficulty.

The DirCom teams learned to produce a "come here" signal in 50% of the evolution runs. Their performances ranged from 13.00 to 19.3. The best team used both a "come here" signal and exhibited efficient exploration (agents spread out).

The five best performing DirCom teams where transferred for use in real Khepera III robots and placed in arena containing a single food item. The transferred teams demonstrated the same level of group coordination, despite the difference between the simulation and the real world.

These results demonstrate that while it is possible to evolve effective communication without directional reception, it is less feasible from an evolutionary standpoint than

when directional reception is used.

Similarities to our research:

- HyperNEAT is used to control a team of agents.
- Agents must perform a task that requires teamwork.
- Agents use signals to communicate with one another.

Our research is distinguished by:

- Only one communication schema is used by our agents, similar to their DirCom schema.
- The primary focus of our research is on the effects of different applications of online learning during evolution.
- Our research explores multiple extensions to the HyperNEAT algorithm.

3.7. Indirectly Encoding Neural Plasticity as a Pattern of Local Rules

Risi and Stanley [22] introduce a method called adaptive HyperNEAT that encodes both weights and local learning rules as a pattern of geometry. Their idea is to extend HyperNEAT so that the CPPN generates not only connection weights of the substrate, but also generates "local learning rules" for neural plasticity. That is, the weights will be updated during the lifetime of the substrate, and the CPPN will generate values that

control how the weights will be updated.

They compared three different adaptive methods that encode different levels of generality for the learning rules.

The first model they implement, and the most general, adds three additional inputs to the CPPN: presynaptic activity (o_i), postsynaptic activity (o_j), and the value of the current connection weight (w_{ij}). The output of the CPPN remains the value of the weight. The update of the weight in this model is performed iteratively, that is, at every tick of the clock. So, not only is the CPPN queried to produce the initial weight value, it is queried every time the substrate is activated.

The next model they implement is the Hebbian ABC model. It introduces four additional outputs to the CPPN: learning rate η, correlation term A, presynaptic term B, and postsynaptic term C. The weight update rule becomes:

$$\Delta w_{ij} = \eta(A_{oi} o_j + B_{oi} + C_{oj})$$

In this case, since the learning rule is a static formula, the CPPN need only be queried once, to retrieve the initial value of the weight, and the values for the ABC model.

Note that this model is less general than the iterative one. Given that the CPPN may only produce values for the parameters A, B, C and η, the update rule will always be some variant of the formula above. However, using the iterative model, the CPPN itself supplies the weight change values, and thus may learn any arbitrary function, including nonlinear ones.

The third model is plain Hebbian. It is as the ABC model, but uses the formula:

$$\Delta w_{ij} = \eta o_i o_j$$

In this case, the CPPN outputs only the initial weight value, and the learning rate η for weight change.

Risi and Stanley setup an experimental environment to test these models. They use a simple T-maze depicted below.

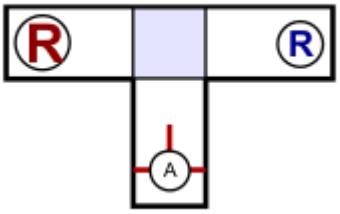


Figure 3.3: A T-Maze.

The agent starts in the bottom of the maze, and in each of the arms of the maze lies either a high reward or a low reward. An agent is subjected to many trials through the maze; the goal is for the agent to maximize the reward received over all trials. Sometimes, the position of the rewards changes, and the agent will need to alter its strategy and recall the new position of the high reward to be successful.

The agent has range finders that detect the walls to its left, right, and front. A "color" input is set to the color of the reward collected at the end of the maze. The outputs control the agent's movement forward, or turning left or right, and correspond to the spacial placement of the rangefinders.

Two scenarios are setup to experimentation.

Scenario 1: The traditional T-maze is used. The position of the high reward alternates between deployments. Within a deployment, its position switches after an average of 50 trials. Color input values for the rewards are 1.0 and 0.1 for the red (high) and blue (low) reward, respectively.

Scenario 2: Color input values 0.3 and 0.8 are introduced for the colors yellow (high) and green (low), respectively. By adding these intermediate colors strategically in the maze, the reward signature becomes non-linearly separable. Since the substrate controlling the agent has no hidden-neurons, it is necessary for learning rule to be non-linear.

For all experiments, the fitness function is the same: each high reward receives a value of 1.0, each low reward receives a value of 0.2, and a collision with a wall gives a value of -0.4. Total fitness is the sum of these values across 100 trials.

3.7.1. Results

Scenario 1: The iterated model took and average of 89 generations to find a solution; the ABC model 141 generations. The plain Hebbian model never solved the task.

Scenario 2: Plain Hebbian was excluded from this scenario. The iterated model solves the task in 19 of 20 runs, in an average of 367 generations. The ABC model is not able to solve the task, suggesting that this scenario required a non-linear learning rule. The more general (but more computationally expensive) iterative model is able to evolve a

successful rule.

Their experiments demonstrate that there is a trade-off between the generality of indirect plasticity encoding and how computationally expensive it is. However, in some cases (e.g. non-linear problems) only a general encoding is able to solve the task.

Similarities to our research:

- Networks are evolved that can have their weights changed during their lifetimes.
- The use of the Hebbian weight change rule, and variants, are explored in conjunction with HyperNEAT substrates.
- HyperNEAT is extended to support evolution of additional parameters that control weight change during a network's lifetime.

Our research is distinguished by:

- Additional strategies for weight change are explored, beyond Hebbian methods.
- The primary focus of our research is on the effects of different applications of online learning during evolution, as opposed to explicitly evolving networks that can change policies in response to environment changes.

4. ENHANCING HYPERNEAT WITH ONLINE LEARNING

The focus of our research is to produce advances in the HyperNEAT methodology through the introduction of additional learning algorithms applied during evolution, and extensions that may potentially support the integration of HyperNEAT with these other techniques. HyperNEAT is used in an offline fashion, where individuals are produced as per the algorithm and then, during the evaluation phase, trained with either supervised or reinforcement algorithms. A number of methodologies are proposed and explored:

- Using Backpropagation, Hebbian, and Temporal Difference learning algorithms for training HyperNEAT substrates during evolution.
- Extending HyperNEAT is to produce learning rate parameters as part of the evolutionary process.
- Using the effectiveness of an individual's ability to use the integrated learning techniques as an additional measure of its fitness.
- Using geometric translation of training patterns to take advantage of HyperNEAT substrate geometry.
- Using a technique called Bootstrapping that uses online learning for only part of the evolution process, allowing it to aid in the evolutionary selection process without limiting the behaviors and strategies of individuals in later evolution.
- Using a technique called HyperNEAT with Training Banks that records the states that agents experience and stores them for intermittent training.

This chapter describes each of these concepts and techniques in detail.

4.1. Applying Neural Net Learning Algorithms to HyperNEAT Substrates

As seen in previous sections, the HyperNEAT algorithm evolves the connection weights for a network referred to as a substrate. Aside from being designed to capture the geometry of a problem domain, the substrate is fundamentally no different from any other network; it is fed stimulus inputs, propagates activation values from node to node through connections, and provides output activation values that may be read and utilized. As such, it is possible to train the substrate with conventional network training methods. This is done through alternating phases of offline and online learning. The process of producing a generation in HyperNEAT is treated as offline learning. Then each individual undergoes a phase of online learning where some training technique is applied to the substrate. This may be done prior to or during evaluation of each individual; our research focuses on the latter method. In either case, the resulting fitness of the individual is passed back to the HyperNEAT algorithm as normal, and the process repeats. It should be noted that the online learning methods only update the weights of the network; the structure, as evolved by HyperNEAT, is not modified until the following phase of offline learning. The technique of combining online learning with NEAT has been explored in previous research with good results [16],[17],[23], [24],[27]. Thus it stands to reason that HyperNEAT may be benefited as well.

For these techniques, online learning is performed at the time the substrate undergoes fitness evaluation. The HyperNEAT algorithm generates the CPPN and substrate; the

substrate is passed to some form of evaluation function. During the course of evaluation, the learning technique is applied. Thus the learning technique should contribute to the fitness of the individual. For the purpose of this research, weight changes are maintained on an individual for the duration of its life in the current generation, i.e., weight changes are not propagated to any individual in subsequent generations.

Our research utilizes several online learning algorithms in this fashion, specifically Hebbian, Hebbian ABC variant, Supervised Backpropagation, Reinforcement Backpropagation, and Temporal Difference learning. These algorithms are described in detail in Chapter 2. Each of these is applied to the substrate in separate test trials. The exact methodology used depends on the individual technique.

4.1.1. Supervised Backpropagation

In order to apply supervised backpropagation to the substrates, it is necessary to have training values that map to input states. This could come in the form of a traditional data set, with pairs of input data and expected outputs, but it is also possible to generate desired outputs through some fixed policy or heuristic method. This research uses the latter method; the details are described in Chapter 5.

This is similar to previous research where learning was combined with evolution in an attempt to improve the results achieved through evolution [16],[17],[18]. Parisi and Nolfi's work [16] trained networks on a prediction task that differed slightly from the fitness function used in evolution. As indicated, our research uses a heuristic method to train the networks. McQuesten and Miikkulainen apply training prior to the evaluation

of a network using an oracle as a teacher; our approach applies training during the course of evaluation.

4.1.2. Reinforcement Learning

The four reinforcement learning techniques, temporal difference learning, reinforcement backpropagation and the two Hebbian techniques are used by applying a reward or punishment value to the substrate and updating weights accordingly.

For substrate applications that operate by selecting a winning output node (that is, the output with the highest value) the reward is a vector matching the length of the outputs. In the reward vector, the element with an index matching the winning node of the substrate is given some positive value (or a negative value for punishment); other elements in the vector are given a zero value. The exact value used as a reward or punishment is configurable based on the algorithm in play and the needs of the experimental domain. Frequently, different states are assigned different reward values depending on how good or valuable they are, or how bad they are in the case of punishment. Marginally good states may be given rewards in the range of 0.01 to 0.5, very good or optimal states may receive values up to 1.0. Again, these values may be different depending on the algorithm and task. Conventionally, temporal difference uses rewards in the range of 0 to 1 ([9], [10]), and reinforcement backpropagation uses discrete 0 or 1 values exclusively [7]. Chapter 5 describes the specific reward values used for the experiments in this research.

The Hebbian algorithms are not reinforcement algorithms per sé, but are used in a

generally reinforcing (or anti-reinforcing) fashion when combined with HyperNEAT. Hebbian learning adjusts connection weight values based on the activation values of the nodes on either end of the connection. For basic Hebbian, new weight values are simply the product of the two activation values and a learning rate plus the current weight value. In general, connections between nodes with high activations will increase, and those between nodes with low activations will decrease. To adapt this to reinforcement learning, only a small change is made. If the current state should be rewarded, the Hebbian update is applied as normal. If the current state should be punished, the product of activations and learning rate is subtracted from the current weight value, instead of added to it. The effect is that if, say, a desirable state is observed, weights will be increased (or decreased) in proportion to the activation values of their adjoining nodes. Conversely, if an undesirable state is observed, connections strengths will be reduced, particularly where adjoining nodes had high activation values. This process is identical for the ABC variant, excepting that three additional parameters also figure into the weight update calculation.

The theme of this approach is similar to prior research only in that it combines learning and evolution. As mentioned, some research focused on using backpropagation in a supervised fashion to augment evolution [16],[17], which differs from our use of reinforcement learning. As well, our approach applies Hebbian changes in the fashion of reinforcement learning, a divergence from other research that combines Hebbian techniques with evolution. Research by Stanley, Bryant, and Miikkulainen [18], and by Risi and Stanley [22] applied Hebbian weight changes to evolved networks by

automatically updating weight values every time step, instead of only for reinforcement or anti-reinforcement, as our approach does.

4.2. Using HyperNEAT for Learning Parameter Selection

In addition to applying online learning techniques to the HyperNEAT substrate, we also use the HyperNEAT CPPN for learning parameter selection. For a moment, reconsider the HyperNEAT algorithm. A substrate is designed, and its weight and bias values are filled in by an evolved CPPN. However, the CPPN is not limited to these two types of output; using the CPPN, it is possible to output other potentially useful values such as per-weight learning rates for neural plasticity [22].

In the same way, our research continues and extends the previous work by applying the technique in conjunction with the learning techniques referenced in the last section. For the techniques that use only a single learning rate parameter (basic Hebbian), the CPPN is modified to have one additional output node to represent the value of that parameter. That node is queried at the same time as the node for the weight, resulting in a potentially unique learning rate for each connection. Similarly, for the techniques with multiple parameters, the CPPN is modified to include a number of output nodes corresponding to each relevant parameter. For backpropagation, two nodes are added, one for weight change rates and one for bias change rates. For Hebbian ABC, four nodes are added, one each for the parameters A, B, C, and the learning rate. For temporal difference, three nodes are added, one for γ, one for λ, and one for the learning rate.

While it is possible that the CPPN will select unique values for each parameter for each

connection in the network, it is also possible that the values will correspond to the layer in which they occur, or will be the same across the entire network. As an example, it is possible that all learning rates on the first layer of connections of a network have a value of 0.25 and all connections on the next layer have a value of 0.1. Chapter 6 looks more deeply into the values of actual CPPN produced learning rates and their significance.

The goal is that HyperNEAT will select learning rate values that will optimize the online learning, improve the fitness of individuals and produce solutions to tasks in fewer generations than those with fixed learning rates, or baseline experiments with no online learning.

Previous work by Risi and Stanley [22] used the same method modifying the CPPN to output additional parameters, for the purpose of evolving effective neural plasticity. This method takes advantage of more recent technology than the work done by Stanley, Bryant, and Miikkulainen [18], using HyperNEAT instead of a modified NEAT algorithm. Our research further expands these ideas, by applying the technique to other learning techniques, beyond Hebbian learning.

4.3. Using the Effectiveness of Learning in Repeated Trials as a Fitness Measure

Using the methods described in section 4.1 and 4.2 for online learning, there may be room for additional improvement. The effectiveness of online learning that takes place during evaluation may be used as a measure of individual fitness. That is, the degree that an individual improves as a result of online learning may also be used when considering how fit the individual is. This is may be of particular importance when learning

parameters are generated by the CPPN to help ensure that values are chosen that permit online learning to support the evaluation task, and help eliminate those individuals that have poor learning ability.

The primary method for accomplishing this is by subjecting each individual to multiple evaluation trials, and allow online learning to progress during each. The individual's performance is tracked during each trial, and an improvement factor is calculated. The improvement factor is a general measure of how much the individual's performance improved across trials. The idea is to use a weighted average of the performance values, with performance in each subsequent trial being weighted more than previous trials, and to compare the result with the performance in the first trial.

This should favor individuals that have structures, weight, and parameter values that are more conducive to learning. This is analogous to biological life: those individuals with genetic traits that support learning are more likely to exhibit learning, and thus be more effective at survival. This has the potential to enhance performance when combining online learning techniques with HyperNEAT.

In particular, this approach may increase the effectiveness of the technique described in section 4.2. Since learning rates and other parameters are output jointly with the substrate weights, using the improvement factor as a fitness affords the effectiveness of learning to be directly improved and optimized by the HyperNEAT algorithm.

We developed an algorithm to test these ideas. This algorithm functions as an extension to HyperNEAT. It is applied at the time that HyperNEAT evaluates an individual's

fitness. The algorithm is as follows:

```
Using the current individual:
      For N trials:
            Observe individual in evaluation task,
                  while applying online learning
            Calculate performance for individual
            Record performance in a list
      Set improvementFactor = 0
      Set weight = 1
      Set weightTotal = 0
      For Each performance value in list:
            Set improvementFactor =
                  improvementFactor + (currentPerformanceValue + 1) *
weight
            Set weightTotal = weightTotal + weight
            Set weight = weight + incrementAmount
      Set improvementFactor = improvementFactor / weightTotal
      Set basePerformance = first value in Performance list
      Set improvementFactor =
            (improvementFactor – basePerformance + 1) / (basePerformance +
1) - 2
      If improvementFactor < 0
            Set ImprovementFactor = 0
```

where *weight* is the weight for each performance value in the average and *incrementAmount* is a parameter used increase that weight for each trial, that is, to control how much to consider the performance in each subsequent trial. Chapter 5 indicates the specific values we used for these parameters.

Note that the variable *basePerformance* is incremented by 1. This is done to handle the case where the *basePerformance* was 0 and prevent an undefined result in the improvement factor calculation. The addition of 1 to *currentPerformanceValue* and the subsequent subtraction of 2 in the final step of calculation is done to offset that.

Previous research [17],[18],[23] suggests that learning ability can lead to better evolved individuals, but does not attempt to alter the evolutionary fitness function to accommodate it. Our methodology described here focuses on using the degree to which an individual learns as a fitness measure directly.

4.4. Geometric Translation Training

Geometric Translation Training is our original technique that trains a neural network with training samples that are presented to the network multiple times, with each presentation being some translation of the original sample pair, such as a mirror image or rotation. This technique may be used in conjunction with neural evolution techniques, to augment the effectiveness of evolution. This technique has added potential for enhancing HyperNEAT, by using translations that align with the geometry of the HyperNEAT substrate.

Conventional supervised and reinforcement network training approaches work thusly: inputs are fed to the network as usual, producing an output. Then, some learning mechanism provides feedback to the network - either an expected value for supervised learning, or a reward or punishment value for reinforcement learning. In cases where there is geometric regularity or relationship amongst input values and output values,

training could theoretically be enhanced by performing some form of geometric translation on the input and feedback values. This is accomplished by translating the original input, propagating it through the network, then performing the same translation operation on the feedback value used for training. This technique may be applied multiple times; for example, our research focuses on the use of rotation for translating training samples, thus a ninety-degree rotation may be applied three times after the initial input and feedback cycle with no duplication of effort.

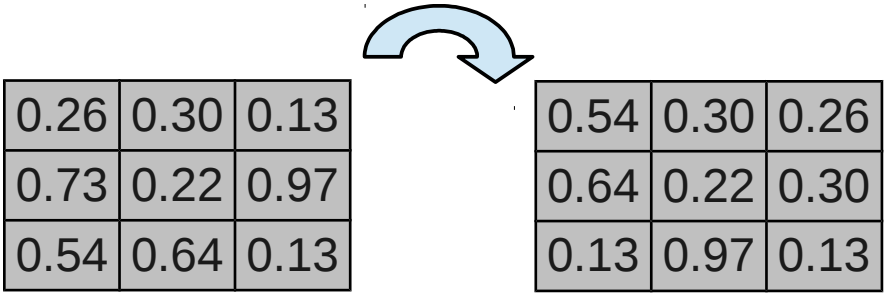


Figure 4.1: A hypothetical network input is rotated 90 degrees clockwise.

This technique allows collections of training samples to be expanded even when domain knowledge is incomplete, as a network experiences more and more states in a domain. Using a single state experienced by an agent, additional states may be generated and trained upon, maximizing the the knowledge available in the form of training pairs. This is especially beneficial in agent-environment scenarios where domain knowledge can only be gained through the agent's experience of the environment. Consider a very simple case: a robot with four wheels, two infrared sensors on opposite sides of its chassis, and a neural network controller. Its motor can cause the robot to move in two directions, forward or backward. The robot is controlled by a neural network that receives input from the two infrared sensors and provides two output nodes, one assigned

to forward movement and one assigned to backward movement. It is given a task of minimizing distance between itself and the nearest wall. The robot's movement response can be trained, say through supervised feedback. The expectation is that it move in the direction of the proximity sensor reporting the nearest wall. As an example, a "near" value signal may be sent to the network by the front sensor and a "far" value signal sent from the rear sensor; therefore we expect the robot to move forward. A feedback value is provided by the trainer indicating that the net should have issued a forward movement command. Since we know that the front sensor should correlate to forward movement and the rear sensor to reverse movement, the training pair (input and expected output), can be translated, in this case, flipped vertically. That is, a new training pair is generated consisting of the original input vertically flipped (as if it had been the rear sensor reported the nearest wall) and the expected value flipped - to indicate that the reverse movement was expected. In this way it is possible to capture and train upon states not yet encountered in the environment.

Given that HyperNEAT produces substrates that 1) reflect a particular geometry and 2) are regular neural networks and may be trained through conventional techniques, the algorithm should be a natural choice to benefit from application of geometric translation training.

In the example of the dual-sensor robot, if we had evolved a network controller with HyperNEAT rather than train it, we would very likely see that the connection weight(s) from the front sensor input node to the forward output node would be very similar to those from the rear sensor input node to the reverse output node, since substrates have

been shown to have geometric connectivity patterns [14]. This process could be augmented earlier in the course of evolution by providing online training, that is, ongoing network weight adjustments during the course of fitness function evaluation.

The use of geometric translation during online training is one of our original contributions. While previous research [16],[17],[23],[24] explored the combination of training and evolution, it did not focus on the online aspect or any translations to training pairs.

4.5. HyperNEAT with Supervised Online Learning and Bootstrapping

The results of the some of our initial experiments with online learning prompted an effort to achieve optimum results with greater frequency; this is discussed at length in Chapter 6. This lead us to develop a "bootstrapping" technique, that uses online learning as a means to guide evolution in its early stages.

The approach is to bootstrap evolution by starting the evolution run with online learning and then, after some number of generations, turning the learning off and continuing with evolution as normal. The rationale is that since the best results were only achieved without online learning and consistently good results were achieved with it, that some combination of the the two could achieve the best results with greater consistency. This can also be useful for environments where the optimum strategy is not known a priori. Using backpropagation in an online fashion requires a good heuristic to be successful. However, the results from the of experimental trials (see experiment description in section 5.6.3.1 and the results in chapter 6) show that the heuristic employed for training

is not sufficient to achieve optimum performance. Bootstrapping theoretically allows evolution to benefit from an imperfect heuristic.

The bootstrapping technique is implemented by using online training in the same fashion as described previously. The training is applied through each generation until an average fitness threshold is reached, that is, when the average fitness of each individual in the population meets a target threshold. From there, online learning is shut off for the duration of the evolution run. Multiple experiments are run to see the effects of using different thresholds.

Like geometric translation training, bootstrapping is one of our original contributions that is not explored by previous research.

4.6. Storing Memories for Intermittent Offline Training

A problem in many agent-environment systems is that there is no a-priori knowledge with which to train the networks. It is necessary for the agents to explore and interact with the environment to gain useful experiences. While we may not know the most desirable action for a particular state, often it is possible to identify when an agent has arrived in a good state. Following this line of thought, we developed a means to capture information about an agent's experiences – those times when an agent performs well - and store it for training purposes. By recording these states and experiences, it should be possible to glean knowledge to be used for training, i.e., pairs of sensory input and expected outputs, when the agent performs actions leading to positive or negative states. Once stored, these pairs may be used for supervised training. This process may be extended to record not

only individual states, but a sequence of linked states.

This technique departs from the other methodologies we describe in that it combines HyperNEAT with offline training, instead of online training. This is similar to the research conducted by McQuesten and Miikkulainen [17]; the evolutionary algorithm is paused to allow training to occur, prior to the final evaluation of an individual network. However, their research uses a high performing individual from the previous generation to train the next. Our approach uses the experience gleaned by an individual during its lifetime for training. As well, their research only executed training once per generation, and did so prior to evaluation in the environment. Ours executes training only after an initial evaluation phase, as it is necessary to amass a starting training bank.

4.6.1. Recording States and Sequences

As an agent explores an environment, states and sequences of states are accumulated in a training bank. Each agent keeps a memory of the states it experiences. These states are comprised of observed inputs, network outputs, and a reward designation: either reward, punishment, or neutral. The reward designation is provided by the environment or some other mechanism to determine the appropriate feedback to give to the agent about the state resulting from its action.

These experiences may be used for future training. For instance, positive experiences can be used to reinforce agent behavior. How this is accomplished may depend on the specific technique used. If backpropagation is used, the usual strategy of supervised training can be applied by treating the experience state as a training pair. That is, the

original input is used as the input part of the pair and the action that was originally selected by the agent is used as the expected output. States that have a punishment or neutral designation may also be used for training; using these states is described later.

For training networks used in agent-environment scenarios, it is not very practical to train using only individual reward or punishment states, given that much of the time spent in the environment will be exploration and seeking states with potential reward. It makes sense to include those states that lead up to reward states, particularly in environments where rewards are received only after a series of possibly complex behaviors have occurred. This is referred to here as a “sequence”.

Sequence collection occurs thus: when a state is encountered that provides a reward or punishment, that state along with previous states are transferred from the agent's memory to the shared training bank. At that point, the agents memory may be cleared for the collection of new sequences.

When a sequence is transferred to the bank, states within the sequence that are designated as neutral may be revisited. If a series of neutral states leads to a reward state, it may be of benefit to convert these states to reward states to be used for reinforcement. Likewise, it may be useful to discourage neutral behaviors leading to punishment states. Given that neutral states leading to reward or punishment states may be of less consequence (due to the possibility of multiple paths to a desirable or undesirable state), training may be enhanced by applying a lesser weight or learning rate to such samples.

4.6.2. Training

The training bank amassed as an agent explores the environment is used for training. This may be done alone or in conjunction with any of the techniques previously described. In particular, this fits with the practice of having multiple periods of evaluation for a given network; training may be applied between the evaluation intervals.

Training proceeds by taking each state from the bank and applying it to the network via a training algorithm. Given the structure of the samples as sets of input-output pairs with reward designations, backpropagation is an ideal method. As previously mentioned, reward states may have the net's original input treated as the desired output to that state's associated input.

States with a punishment designation may require adjustment for use with backpropagation. Various strategies for doing this are possible. A simple method that seems effective when using the network output as a selector (i.e., in cases where each output node represents a discrete action, and the highest valued node is selected), is to reduce the winning node's value to zero while leaving the value of other nodes unmodified. This is the approach used in our research.

Training can be performed until a desired mean squared error is achieved or after an arbitrary number of epochs have elapsed. The goal of the offline training is to augment and speed up the evolutionary process, not to produce a fully trained network by itself. As such, training for a large number of epochs or to a low mean squared error is neither necessary nor desirable, since this activity will increase the amount of computation time

required.

4.6.3. Technical Limitations

While results are promising with the use of training banks, current hardware limits the extent to which they may be implemented. In theory, training states could be continuously collected from agents and stored in the training bank. However, the number of agents present and the length of time they spend in the environment are factors that multiplicatively combine to form the size of the training bank. This presents a couple of problems. The first obviously is that physical memory is limited and may be unable to support unbounded training banks, especially in conjunction with other algorithms. Perhaps the more significant problem is the amount of computation time required for training, which will grow with every sample added to the training bank. The size of the bank might be reduced by only adding unique samples, however that also adds computation time to compare each new sequence to each other sequence already in the bank.

Many strategies are possible that would mitigate or obviate these limitations. Chapter 5 describes how the training bank technique is applied in this research, given current technology limitations.

4.6.4. Notes on Combination with other Techniques

HyperNEAT with Training Banks

The use of the training bank technique may be used with HyperNEAT as described previously for evolutionary algorithms. In this case, evolution proceeds as normal until a chromosome is evaluated. At that time, the substrate produced by the chromosome is subjected to multiple discrete episodes where training data is collected. After each episode the substrate is trained using the data collected during that episode. The fitness may be evaluated for any of the episodes or overall performance across all episodes. Our research focused on the latter method.

HyperNEAT with Supervised Online Learning and Training Banks

Section 4.1 mentions that online supervised learning (learning that occurs during a substrate's evaluation period) may be combined with HyperNEAT by using a heuristic feedback provider to produce an expected response for each timestep experienced by an agent. Following this approach, training pairs are created using the input to the agent and the expected response. This pair may be immediately accepted to the training bank without regard to the agent's actual response. Since the sample represents the action an agent should take in a given state, it is encoded as a reward sample for the purpose of training; thus there are no punishment samples added to the training bank. Further since every sample is treated as a reward, only individual samples are collected; no sequences are ever recorded.

Chapter 5 describes how these techniques were implemented for this research.

5. APPLICATION ANALYSIS

This section describes the task and experiments used in this research, and how the techniques in Chapter 4 have been implemented.

In order to test the proposed methods, some sort of problem was required. A robot gathering task was chosen to evaluate the effectiveness of each methodology. We created a software simulation of a simple environment to model the task. In this simulation, robots are controlled by a neural network and must locate resources and carry them back to a base location. In addition, the robots must work together, as it requires multiple robots in order to carry resources. Thus, the goal is for the network controlling the bots to evolve and learn how to perform this task effectively, gathering as many resources as possible during a specified period of time.

For this research, the neural network that controls the robots is the substrate produced by the HyperNEAT algorithm, with or without the online learning enhancements previously described.

We designed the simulation, the substrates, and the experiments based on our own needs and goals for the research, but drawing influence from prior research. In particular, we drew from HyperNEAT research performed by Clune et al. [15], D'Ambrosio and Stanley [19], D'Ambrosio et al. [20], Pugh et al.[21], and S. Risi and K. Stanley [22]. Specific details regarding these influences appear in relevant sections below.

5.1. Environment Design

Our environment uses a simple map consisting of cells arranged in a two-dimensional grid. Each cell can hold one entity at a time. An entity may be a robot ("bot"), or a resource ("food").

Certain sections of the map are designated as a "base" for the bots. Cells are marked as bases may contain bots or food as with any other cell on the map; the distinction is that base cells are collection points for food. The task in the environment is for the bots to pickup food and return it to the base. When a piece of food lands on a cell designated as a base cell, it is removed from the map, and the tally for the number of food items collected is incremented. The goal is for the bots to have collected a certain amount of food by the end of an evaluation period.

5.2. Agents

The agents in this experiment are simulated robots (“bots”). The bots have a set of sensory apparatus that informs them of the state of the environment and may interact with the environment by moving and coming into contact with resources. A HyperNEAT substrate receives sensory input and controls the actions of the bots.

Bots can move one cell per timestep, in any of the eight cardinal and inter-cardinal directions: North, Northeast, East, Southeast, South, Southwest, West, and Northwest. This is distinctly different from most other research that uses neural net controlled agents. Most other research uses simulated robots that can only move forward or turn [18],[19], [20],[21],[22].

5.2.1. Interaction with Food

In order for the bots to pickup a piece of food, it is required that two of the bots bump into the food. When a bot bumps into a piece of food, it will "attach" to it, if it is not already carrying any food. Once attached, the bot will begin transmitting a signal that is receivable by other bots anywhere in the environment. The bot then remains stationary until another bot attaches to the food, four timesteps have elapsed, or the end of the evaluation period is reached. While a single bot is attached to a piece of food, its network is not used; it performs no action other than transmitting the signal for assistance. If four timesteps have elapsed and no other bots have attached to the food, the lone attached bot will detach and is free to act again; this prevents bots from being permanently stuck to a piece of food in the event that no assistance ever arrives.

Note that this is very similar to the design used by Pugh et al. [21]. Both their research and ours requires multiple bots in order to collect food, and uses a signal transmitted by a bot to indicate the location of discovered food. However ours differs in several ways. In our environment, only two bots are required in order to carry food, and it is not sufficient that they merely come in contact to the food, they must also coordinate their movements to carry it back to a base. As well, even though we use a communication signal, it is not the focus of our work. Despite the similarities, Pugh's work did not influence this particular design aspect of our research, as it was done concurrently (2013) to ours, and with no collaboration.

When two bots have attached to a piece of food, they enter a towing state and may carry the food. In a timestep, the first bot that acts is designated as the "lead tower". When the

lead tower moves, the food will be immediately moved into the cell previously occupied by the lead tower. This prevents the food from running into obstacles.

From this point, two modes of towing are possible: single bot towing and double bot towing. In single bot towing, The movement of the second bot follows the piece of food. That is, the lead tower moves in a direction, the food moves to the previous cell of the lead tower, and the second agent moves into the previous cell of the food. In this mode, the second bot is not consulted for a direction, it merely follows the food and the lead tower. In the double bot towing, the lead tower moves the food just as in the first, but the second bot must decide on its own in which direction to move. Should the second bot move more than two cells away, the food is "dropped" in its current cell, and it is necessary for two bots to attach to it again in order to continue moving it. Thus in the first method of towing, it is not possible for food to be dropped, in the second, it is.

Having two modes of towing is advantageous. Earlier in the course of evolution, single bot tow mode permits the networks to learn to move food toward the goal. As evolution progresses, and the networks become more competent, the mode may be switched to double bot towing, to permit the networks to learn to coordinate between bots. Section 5.5 describes how these two modes are used in conjunction with fitness shaping.

5.2.2. Sensory Apparatus

Bots have three senses that tell them about their environment. These sensors are laid out in a stack of 3 by 3 grids. That is, each sensor provides information in a 3 by 3 grid, and these are fed to the substrate one on top of another, in a stack.

Proximity Sensors

The first sensor informs the bot of the status of adjacent cells. For cells that are empty (free for the bot to move into), a value of 1.0 is provided to the sensor. Cells containing food use a value of 0.5. Cells that contain other bots, or that mark the boundary of the environment use 0. This way, higher values of 1 and 0.5 are favored for selecting a direction over the low 0 value representing an obstacle.

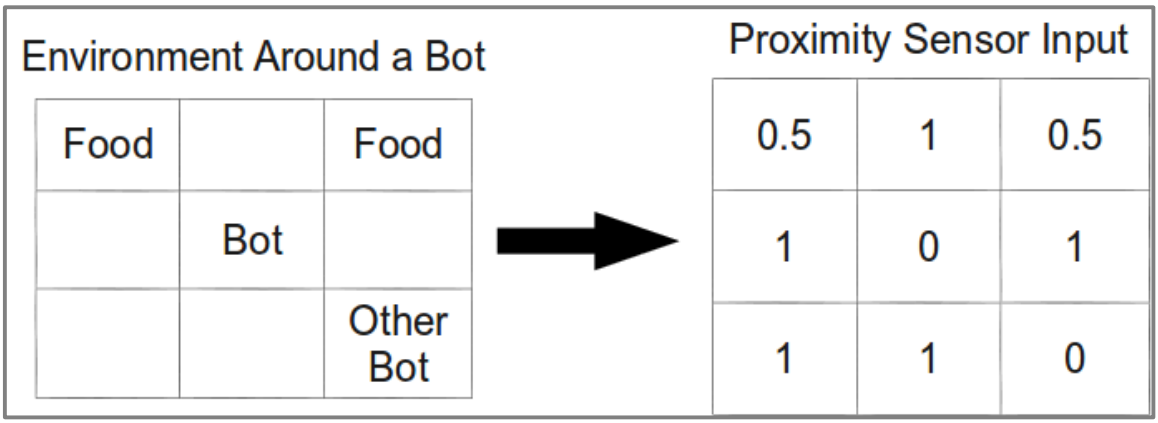


Figure 5.1: Generation of proximity sensor input.

Vision Sensors

The second sensor is a short-range vision sensor. Bots can "see" food. Food within a radius of 3 cells is noticed by the bot (excluding the bot's current cell), and is encoded as the inverse of the normalized distance from the bot to the food. That is,

$$1-(distToFood / maxEnvironmentDist)$$

where *distToFood* is the distance from the bot to the food, and *maxEnvironmentDist* is the maximum possible distance from any cell to any other cell on the map. Inverse distance is used so the network favors higher values over lower ones, as with the first sensor

apparatus.

The inverse normalized distance is placed in the sensor slot corresponding to the direction of the food. For example, if the food is located generally to the north of the bot, it is placed in the sensor slot with coordinates (1,0); food to the south east of the bot would be placed in the coordinates (2, 2). If there is more than one food that would fall into a given sensor slot, then the distance value is added to the existing one, up to a maximum value of 1.0. Figure 5.2 illustrates this.

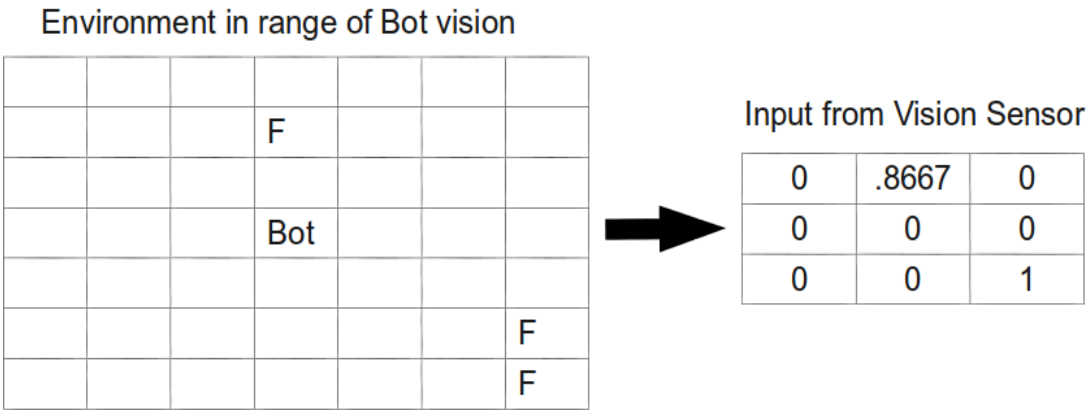


Figure 5.2: Generation of vision sensor input.

The general direction of the food is determined using the formula to calculate the angle of the food relative to the bot:

$$bearing = Math.toDegrees(Math.atan2(dy, dx)) + 360$$

where *dy* and *dx* are the distances between the bot and food with respect to their vertical and horizontal coordinates. The *Math.atan2* and *Math.toDegrees* are predefined functions in the Java API. The *Math.atan2* function returns the angle *theta* in radians from the conversion of rectangular coordinates (*x*, *y*) to polar coordinates (*r*, *theta*). The

Math.toDegrees function converts the angle to an approximately equivalent angle measured in degrees.

This gives the *bearing*. This angle value is then discretized into a direction from the eight available with the formula:

$$direction = bearing / 360 * 8$$

Signal Sensors

When a bot attaches to a piece of food, it sends out a signal for assistance. This signal is received by all other bots. The sensor that receives these signals works in much the same way as the vision sensor, except there is no limit on the signal distance. The input is calculated by using the formula for inverse normalized distance. The value is placed into the three by three input grid according to its general source direction, calculated the same way as the vision sensor. Multiple signals may be combined into a single value, up to 1.0, for signals emitting from the same direction.

These signals are only received by bots that are not currently attached to or towing food. Bots that are towing food instead receive a signal from the base. These signals work in roughly the same manner. The signal from the base is placed in the input grid in the position corresponding to the direction of the base. The inverse normalized distance from the input cell to the center of the base is used as the input value. To aid in navigation, the two input cells adjacent to that cell also receive an inverse normalized distance value. Figure 5.3 illustrates a bot towing a piece of food; the signal received corresponds to the general direction of the center the base. This provides additional options for navigation

that will lead the bot in the same general direction as the base.

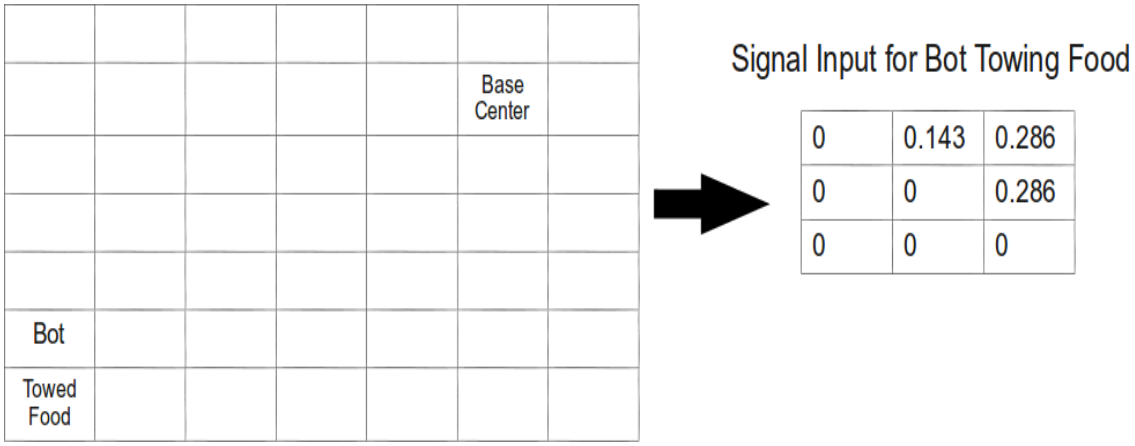


Figure 5.3: A bot towing food and the corresponding signal input.

5.3. Substrate Design

The substrate serves as a "brain" for the bots, determining the direction for movement. Two substrates are used for this purpose, one to direct a bot when it is not carrying food, and one to direct the bot when it is carrying food. It is possible for the HyperNEAT to produce two substrates by feeding the substrate index or coordinates through additional inputs to the CPPN. Using multiple substrates, each for a different aspect of a task, is supported by previous research [20]. The weights for the two substrates may differ, as produced by the HyperNEAT algorithm, but their topology (described below) does not differ. As well, the information input to the substrates is the same, and the output of each is used for navigation. All of the bots use the same two substrates for navigation.

The substrates for this experimental design use several multidimensional layers. The input layer consists of nodes corresponding to the sensory apparatus. As described in the previous section, these are arranged in three by three grids, and stacked on top of one

another into the third dimension. Effectively, the input layer is a three by three by three matrix.

A hidden layer is used in the substrate. Its dimensions are 3x3x2. In some earlier experiments for this research, a (two-dimensional) 3x3 node hidden layer was used; however it performed very poorly and was not used in the final run of the experiments.

The output layer is a 3x3 node layer. The output node with the highest value is known as the winning node, and is used to indicate the direction to move the bot. If the center node, i.e., with coordinates (1,1), is the winning node, the bot does not move. This serves no practical purpose, and is unlikely to manifest in an evolved network, but is preserved for completeness.

5.4. Implementation Notes

All components of the environment and experiments were written in the Java programming language. The HyperNEAT implementation was based on the ANHI (Another HyperNEAT Implementation) engine, written by Oliver Coleman. The source code can be found at:

https://github.com/OliverColeman/ahni

The ANHI engine was extended for this research to support the following additional required functionality:

1) Substrates with arbitrary layer dimensions

2) Substrates that store learning rate parameters, for use in other training algorithms

3) CPPNs that transcribe learning rate parameters to the substrates

4) CPPNs that may transcribe multiple substrates

5) Support for online training algorithms: backpropagation, Hebbian learning and variant, temporal difference learning.

These extensions, collectively referred to as AHNI-ND, were written by Shaun Lusk, the author of this research, and are available at:

https://code.google.com/p/ahni-nd/

Appendix A contains additional information regarding the code availability and licensing.

5.5. Configuration Common to All Experiments

The most relevant configurations settings and parameters are described below. Appendix B contains a list of parameters and corresponding values used for configuring the environment and the algorithms.

5.5.1. Environmental Configuration

All experiments use the environment described in the previous sections. The environment uses a map of size of 15 by 15. A 3 by 3 base area is designated in the center of the map. At the start of evaluation, 6 bots are placed in the base and 24 pieces of food are distributed in periodic fashion throughout the map. Should the current

number of pieces of food drop below 10, an additional piece of food will be generated and placed randomly on the map, so as to maintain a minimum of 10 pieced of food. This was implemented as a fail-safe to prevent running out of food, though there were no observed cases of this occurrence in any evolution run. Figure 5.4 illustrates the layout generated at the start of each evaluation. Green cells with '@' are food. Black cells are empty. Blue cells with '*' are bots. Yellow cells are base cells; note that some base cells are obscured by the bots in this image.

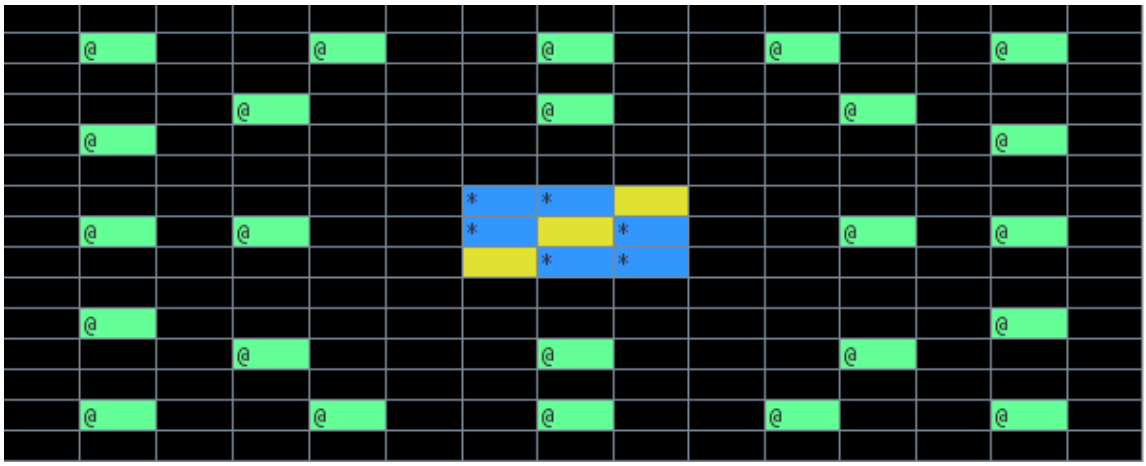


Figure 5.4: The evolution environment.

Initially, single bot tow mode is used, to make it easier for the bots to collect food. When at least one team of bots returns an average of 6 or more pieces of food to the base during their evaluations, double bot tow mode will be turned on for all subsequent generations. This is used as a form of “fitness shaping”, whereby an easier behavior is permitted initially, and then but more complicated requirements are installed after a particular fitness threshold is reached. Section 5.6.2.2. contains more detail about fitness shaping, and how it is implemented for these experiments.

Each substrate HyperNEAT produces is assigned to a set of bots and put through 3 evaluation trials, with the results averaged. This is really only important when evaluating the HyperNEAT / online learning hybrid algorithms, as the networks for these algorithms may change during the course of evaluation. Networks produced from base HyperNEAT with no online learning would not be expected to change; however, they are evaluated 3 times as well for consistency. Each evaluation trial runs for a total of 50 timesteps.

Each technique is executed for 20 evolution runs. Since the nature of genetic algorithms is that they are non-deterministic, it is necessary to evaluate the effectiveness of a technique by running the algorithm many times.

Each evolution run progresses for up to 1000 generations or until the success criteria is met; the success criteria varies between the set of Experiments 1 and 2.

In cases where online learning is used, one training step is applied to the substrate during each timestep where a bot takes an action (this is not applied when bots are waiting). Note that since an individual substrate controls a team of bots, it will receive the training multiple times in a given timestep. The environment will generate feedback for each action taken by a bot. If the supervised backpropagation technique is being used, an expected output value will be heuristically generated (see below). For all other reinforcement techniques, the environment will generate a reward or punishment value to supply to the learning algorithm. These values vary by the algorithm used; the experiment descriptions in 5.6 and 5.7 contain the specific values used by each.

5.5.2. HyperNEAT Configuration

A population size of 100 is used by HyperNEAT.

Both the CPPNs and Substrates are feed-forward only with no recurrent connections. The input and output layers of the CPPN use linear activation; other nodes added through evolution each receive a random activation function. The hidden and output nodes of the substrates use sigmoidal activation.

Substrates use a per-node bias, with values as transcribed by the CPPN. The CPPN itself uses a bias input node that outputs a value of 1.0. The use of a single bias node for the CPPN was chosen for simplicity; only one node is needed, and multiple connections may emerge from it with different weights, as dictated by the course of evolution. The choice to use per-node bias for the substrate is based on the geometric structure of the substrate. Since the structure is designed in such a way as to capture the geometry of the problem, it is not clear where a bias node would be placed. As that is not a question addressed in our research, the choice was made to use a per-node bias.

5.5.3. Backpropagation Heuristic

The experiments that use the supervised version of backpropagation must employ a heuristic with which to generate expected values. These values are generated by taking the substrate inputs at every timestep and applying a heuristic to determine an appropriate expected output for use in training. We developed such a heuristic for use in these experiments. For each bot at each timestep, we apply the following logic:

- If the bot is carrying a resource (with another bot), move in the general direction of the base.
- If the bot is not carrying a resource move toward a signal.
- If there are no signals being emitted, move toward a resource that is in visual range.

The heuristic will never generate an expected output that would result in the bot moving into an obstacle, and further, if no appropriate outputs can be determined, backpropagation will not take place for that timestep.

5.6. Experiments Set 1 Setup

5.6.1. Environmental Setup and Experimental Parameters

For these experiments a success criteria is used. That is, if a certain performance threshold is reached, the evolution will be immediately terminated and marked "successful". The success criteria is when the bots can collect an average of 9.6 or more pieces of food over the course of the evaluations. The goal of these experiments is whether each technique is capable of producing a successful evolution run, and how frequently it is possible.

As per the NEAT algorithm, some percentage of individuals from each generation will be used to create offspring for the next. That subset will be further split into two groups, with some being used for sexual reproduction and some used for asexual reproduction. These experiments carry over 50% of the individuals as parents for the next generation,

with 25% being used as asexual parents and 25% used as crossover parents.

5.6.2. Fitness Function

Multiple fitness measures are used in evaluating the performance of the bots. The primary measure of fitness is the number of pieces of food returned to the base. However, the task of having two bots locate a piece of food and jointly tow it back to the base is complex enough that using only the number returned was thought to be too difficult to find a solution in a reasonable amount of time. Other research has shown that evolving simpler behaviors first, and more complex behaviors later is very effective, especially for complex tasks. Furthermore, a method called fitness shaping has also been shown to improve performance by adjusting the degree to which multiple fitness measures are considered during the course of evolution [25], [26].

Fitness Measures

A total of eight basic measures and one special measure are used for evaluating fitness. The primary measure is the count of pieces of food collected; the rest are measures that relate to behaviors that support that goal. Each is described below.

1) Food Collected

The total pieces of food collected over the course of an evaluation.

2) Attached to Food

The count of the number of times that any bot attaches to a piece of food, that is, bumps

into food when it is not already carrying food.

3) Moved Toward Signal

The count of the number of times that any bot moves in the general direction of another bot's signal for assistance. This is only counted if the bot is not currently attached to or carrying food.

4) Assisted with Food

The count of the number of times any bot assists another bot with food, that is, attaches to food that already has another bot attached to it.

5) Moved Food Toward Goal

When two bots are carrying food, the count of the number of times the lead tower moves in the direction of the goal.

6) Hands Full

The count of the number of times any bot bumped into a piece of food while already carrying food. This is calculated as a penalty.

7) Hit Obstacle

The count of the number of times any bot bumped into a wall or another bot. This is calculated as a penalty.

8) Dropped Food

The count of the number of times any bots dropped food they were carrying (only counted once per set of carriers). This is calculated as a penalty.

9) Improvement Factor

Improvement factor is the measure of improvement the bot makes over the course of the three evaluations. This metric is calculated using our original methodology, as described in section 4.3. It is applied only during the experiments where learning ability is used as a fitness measure.

The improvement factor is calculated:

```
improvementFactor = (weightedAveragePerformance - firstTrialPerformance) /
firstTrialPerformance
```

where *firstTrialPerformance* is the performance of the bots in the first trial (that is, first evaluation), and the *weightedAveragePerformance* is the weighted average of performance across trials. To calculate the weighted average, the performance of the first trial is given a weight of 1.0 and for each subsequent trial the weight is increased by 0.1 cumulatively; so 1.0 for the first trial, 1.1 for the second, and 1.2 for the third.

The "performance" of the bots is calculated:

```
FoodCollected / FoodGoal
```

where *FoodGoal* is the number of timesteps in an evaluation run divided by 5; it is the count of food a high-performing team of bots should be able to collect within an evaluation run. The constant 5 is used as the expected average number of timesteps

elapsed for every food collected by said team. The evaluation runs in these experiments used 50 time steps per evaluation giving a *FoodGoal* of 10.

Fitness Shaping

Fitness shaping is a set of methods that may enhance the ability of evolutionary algorithms to develop more complex behaviors. During the course of evolution, the weight of each measure is adjusted so as to increase the importance of some behaviors and decrease that of other behaviors. This has been shown to improve the quality of evolved solutions, and reduce evolution time [25],[26]. There are a number of specific methods that constitute fitness shaping. In our research, we chose a simple method of linear adjustment.

Each fitness measure is given a starting value and ending value. For each generation, the weight associated with a fitness measure is set according to the following formula:

$$measureWeight = generation * slope + startWeight$$

where *generation* is the number of the current NEAT generation, and the *startWeight* is the initial weight value for a measure. The *slope* is calculated:

$$slope = (endWeight - startWeight) \div numGens$$

where *endWeight* is the target weight value the measure will have in the final generation of evolution and *numGens* is the number of generations for the evolution. Once the *measureWeight* is calculated for the current generation, the weighted score for a given fitness measure can be calculated:

$$weightedMeasureScore = measureScore \div fitnessDivisor * measureWeight$$

When the weighted scores for all measures have been calculated, they are summed to produce a single fitness value to provide to the HyperNEAT algorithm. Starting and ending weight values are chosen to constrain the resultant fitness value to between 0 and 1.

The effect of this shaping is that some measures increase in importance and some decrease in importance over the course of evolution. In general, simpler behaviors, such as those represented by fitness measures *attachToFood* and *moveTowardSignal* are given greater weight at the start and less weight at the end. Similarly, the more complex behavior of collecting food is weighted less in the beginning and much more at the end. The values for each are as follows:

Food Collected Weight Start = 0.4

Food Collected Weight End = 0.7

Attached to Food Weight Start = 0.25

Attached to Food Weight End = 0.5

Moved Toward Signal Weight Start = 0.4

Moved Toward Signal Weight End = 0.25

Assisted with Food Weight Start = 0.6

Assisted with Food Weight End = 0.4

Moved Food Toward Goal Weight Start = 0.9

Moved Food Toward Goal Weight End = 0.05

Hands Full Weight Start = 0.001

Hands Full Weight End = 0.2

Hit Obstacle Weight Start = 0.1

Hit Obstacle Weight End = 0.4

Dropped Food Weight Start = 0.05

Dropped Food Weight End = 0.4

5.6.3. Experiments Performed

Experiment Set 1.1: HyperNEAT with Online Learning vs. Baseline HyperNEAT

The first set of experiments compares the effectiveness of combining online learning with HyperNEAT versus basic HyperNEAT with no online learning in the following trials:

- Baseline HyperNEAT - No online training
- HyperNEAT + Supervised Backpropagation

- HyperNEAT + Reinforcement Backpropagation
- HyperNEAT + Hebbian Learning
- HyperNEAT + Temporal Difference Learning

The criteria for success is whether a network can be produced that allows the bots to collect a total of ten pieces of food during a fifty timestep evaluation period.

For these trials, all learning rate parameters are fixed at the start of evolution with a value of 0.5. Other algorithm-specific configurations are described below.

Baseline HyperNEAT - No online training

This is the experimental control. The evolution runs are carried out without the application of online learning.

HyperNEAT + Supervised Backpropagation

During the evaluation step of HyperNEAT, the substrate receives training through supervised backpropagation, using the heuristic technique described earlier. Backpropagation is only applied for a single training pair per timestep.

HyperNEAT + Reinforcement Backpropagation

This experiment uses the reinforcement backpropagation algorithm.

The reward settings are set as follows:

Attached To Food = 1.0

Assisted Another Bot With Food = 1.0

Delivered Food = 1.0

Moved Toward Goal With Food = 1.0

Dropped Food = -1.0

Bumped Into An Obstacle = -1.0

Moved Toward Signal When Not Carrying Food = 1.0

Moved Away From Goal With Food = -1.0

All Other States = NO REINFORCEMENT

Since backpropagation operates by training a network toward expected values, when backpropagation is used for reinforcement, values are set at -1 or 1.

HyperNEAT + Hebbian Learning

Hebbian learning is applied at each timestep during substrate evaluation using the standard rule to calculate the weight change:

$$\Delta w_{ij} = \eta o_i o_j$$

If the feedback from the environment is a reward, the weight change is added to the existing weight; if it is a punishment, the weight change is subtracted. If the state is deemed as neutral by the environment, reinforcement is not applied for that iteration. The environment provides the following feedback for Bot actions:

Attached To Food = REWARD

Assisted Another Bot With Food = REWARD

Delivered Food = REWARD

Moved Toward Goal With Food = REWARD

Dropped Food = PUNISHMENT

Bumped Into An Obstacle = PUNISHMENT

Moved Toward Signal When Not Carrying Food = REWARD

Moved Away From Goal With Food = PUNISHMENT

All other states = NOT REINFORCED

HyperNEAT + Temporal Difference Learning

Temporal difference learning is applied at each timestep during substrate evaluation using the algorithm described in Chapter 2.

The environment provides the following feedback for Bot actions:

Attached To Food = 0.05

Assisted Another Bot With Food = 0.1

Delivered Food = 1.0

Moved Toward Goal With Food = .3

Dropped Food = -0.5

Bumped Into An Obstacle = -0.5

Moved Toward Signal When Not Carrying Food = 0.05

Moved Away From Goal With Food = -0.05

All other states = NOT REINFORCED

Unlike reinforcement backpropagation, the Temporal Difference algorithm need not have its reward or punishment values saturated to 1 or -1, and may respond to rewards of lesser magnitude [9]. Thus, these values were chosen to represent the relative importance of each action, with the delivery of food being the most significant of all.

Experiment Set 1.2: Learning Parameter Selection vs. Fixed Learning Parameters

The next set of experiments compares the effectiveness of using CPPN-produced learning rate parameters. Each experiment that applies online learning is repeated, this time with learning parameters defined by the CPPN, instead of being fixed. As well, this set adds the ABC variant of Hebbian learning (since the first set of experiments used fixed learning parameters, there was no benefit to including this variant). The results are compared to both the original HyperNEAT algorithm and HyperNEAT with online learning and fixed learning rate parameters.

Algorithm-specific configurations are described below.

HyperNEAT + Supervised Backpropagation

In this experiment, the CPPN is configured to output the learning rate value for weight change and for bias change, for use in the backpropagation algorithm. A separate learning rate value is generated for each weight and each bias node in the network.

HyperNEAT + Reinforcement Backpropagation

As with supervised backpropagation, this experiment uses bias and weight change learning rates produced by the CPPN. Reward settings remain the same as those in experiments 1.1.

HyperNEAT + Hebbian Learning

For Hebbian Learning, the CPPN was configured the same as the two backpropagation experiments, outputting bias and weight change learning rates, however, only the weight change rate was used. As before, each weight in the network receives a separate learning rate. Reward settings remain the same as those in experiments 1.1.

HyperNEAT + Hebbian Learning, ABC variant

This the same as the experiment with Hebbian learning, however, the learning rule is:

$$\Delta w_{ij} = \eta(Ao_i o_j + Bo_i + Co_j)$$

where parameters *A*, *B*, *C* and η are provided by the CPPN, with separate values for each weight. The reward settings used for the ABC variant are the same as those used in basic Hebbian.

HyperNEAT + Temporal Difference Learning

For Temporal Difference learning, 3 parameters are output by the CPPN: the learning rate parameter α, the future reward discount parameter γ, and the past reward discount parameter λ. Reward settings remain the same as those in experiments 1.1.

Experiment Set 1.3: Learning Effectiveness as a Fitness Measure

The final subset of experiments 1 compares the use of the learning effectiveness as a fitness measure versus the original online learning. These experiments combine HyperNEAT with online learning and with the usage of the improvement factor fitness. Recall from Chapter 4 that the performance during each evaluation trial is given a slightly higher weight than the previous one, so as to favor later trials over earlier trials, thus allowing time for the online learning to impact the substrates. The *incrementAmount* parameter (see section 4.3), that is, the parameter that controls how much of an increase the performance weight receives after each trial, is set to 0.1. The weight for performance of the first trial is set to 1.0, and therefore the subsequent weights become 1.1 and 1.2 respectively.

Using this methodology, the full set of experiments from the previous trial is rerun, testing both fixed learning rate parameters and CPPN produced parameters with the use of the improvement factor. Again these are compared with the original HyperNEAT.

5.7. Experiments Set 2 Setup

5.7.1. Environmental Setup and Experimental Parameters

For this second set of experiments, all parameters are the same as described in section 5.5 above with a few exceptions.

The percentages of individuals used for sexual and asexual reproduction differ from the first set. From each generation, 92% of the individuals are used as parents for the next generation, with an equal split for asexual and crossover parents.

There is no success criteria for these experiments; all evolution runs are permitted to continue until 1000 generations have been evaluated. Thus the goal here is to determine the highest performance a given technique is capable of for a specific number of generations, rather than whether a technique can reach a certain performance threshold.

In all cases where a learning parameter is required (in these experiments, only for backpropagation), a fixed value of 0.5 is used.

5.7.2. Fitness Function

For this round of experiments, we designed a simpler and less noisy fitness function than used previously. We use two tightly coupled metrics: the number of food items collected, and the sum of distances between each remaining food item and the center of the base. Fitness is calculated:

$$fitness = \left(1.0 - \frac{foodDist}{startingFoodDist}\right) * distWgt + \frac{numFoodCollected}{10} * collectedWgt$$

where *distWgt* and *collectedWgt* are weight ratios set to 0.2 and 0.8 respectively. The ratio *foodDist* to *startingFoodDist* is subtracted from 1 to produce an inverted score. Thus the smaller *foodDist* is (meaning the bots were more successful), the higher the score. Including this metric allows networks early in evolution to still earn a score if food is moved in the direction of the base, even if it is not successfully collected.

The count of food collected is divided by 10. This was intended as a target value, such that if 10 pieces of food were collected during an evaluation period, then the resulting values is 1.0. As is evident by the results, it is possible for more than 10 pieces of food to be collected during evaluation, so in a few cases fitness scores greater than 1.0 were observed. There was no fitness cap configured for this algorithm, so this was of no consequence.

Using only two fitness measures with no explicit fitness shaping is a departure from the methodology used in first set of experiments. This was done to reduce the complexity of the fitness calculation, and place more focus on performance of the experimental techniques alone, without shaping.

5.7.3. Experiments Performed

Experiment Set 2.1: HyperNEAT with Online Learning vs. Baseline HyperNEAT

The set of experiments is to compare the effectiveness of combining several online learning techniques with HyperNEAT, versus basic HyperNEAT with no online learning. The following trials are performed:

- Baseline HyperNEAT - No online training
- HyperNEAT + Supervised Backpropagation
- HyperNEAT + Rotation Augmented Backpropagation
- HyperNEAT + Backpropagation with Repeat Training

Algorithm-specific configurations are described below.

Baseline HyperNEAT - No online training

This is the experimental control. The evolution runs are carried out without the application of online learning.

HyperNEAT + Supervised Backpropagation

During the evaluation step of HyperNEAT, the substrate receives training through supervised backpropagation, using the heuristic technique described earlier. Backpropagation is only applied for a single training pair per timestep.

HyperNEAT + Rotation Augmented Backpropagation

This trial repeats the process used in the previous one, however, each time backpropagation is applied to the substrate, the training pair is cloned and used to generate three additional training pairs, one for each distinct 90 degree rotation. The four training pairs are presented in a random order for training.

HyperNEAT + Backpropagation with Repeat Training

This trial repeats the process used with Simple Backpropagation, however when a training sample is generated, it is applied to the substrate four times. This is to determine if there is any benefit to rotating training pairs over simply repeating the backpropagation multiple times.

Experimental Set 2.2: Effect of Bootstrapping on Online Learning

The four experimental trials in set 2.1 were repeated with the use of bootstrapping. Four subsets of experiments are carried out, one for each of the online learning variants used in the previous experiments. For each variant, three experimental runs are executed, using different bootstrapping thresholds for terminating the use of online learning. The thresholds are 0.005, 0.010, and 0.020, based on the average fitness achieved by a generation. The results are compared to each of the previous experiments.

Experimental Set 2.3: HyperNEAT with Training Banks

The four experimental trials in set 2.1 were repeated with the addition of training banks.

In this implementation, the decision was made not to use punishment states. Rather, the heuristic for supervised backpropagation used in previous experiments was leveraged to provide the desired output for a bot at each time step; this is treated as a “reward” state. In cases where a desired output could not be determined, e.g., a bot has no clear path, these states are tagged as “neutral”, and stored in the bot's memory for potential addition to a sequence, if a subsequent reward state is encountered.

Due to technical limitations, to avoid extremely lengthy evolution runs when using this technique certain constraints were placed on the collection of states and the amount of training. The training bank for each individual in the population was capped at 75 states. Rather than simply taking the first 75 states encountered by the team of bots, each state submitted to the bank randomly accepted or ignored. An acceptance rate of 30% was used. The amount training applied also received a cap: training was terminated when the MSE reached or surpassed 0.2, or a given number of training epochs had passed. Two sets of experiments were performed using different epoch thresholds; one set with a threshold of 4 and one set with a threshold of 8.

6. RESULTS AND ANALYSIS

6.1. Experiments 1.1: HyperNEAT with Online Learning

The first set of experiments performed compares the performance of HyperNEAT with online learning to the basic HyperNEAT algorithm.

Figure 6.1 shows the food collection results averaged across populations. These are also averaged across evolution runs, since any individual evolution run may vary significantly from others. Note that since this evaluation looks at performance across the entire population for a generation, the average food collected will tend to be low, since many individuals will not be capable of collecting any food, especially in the early stages of evolution.

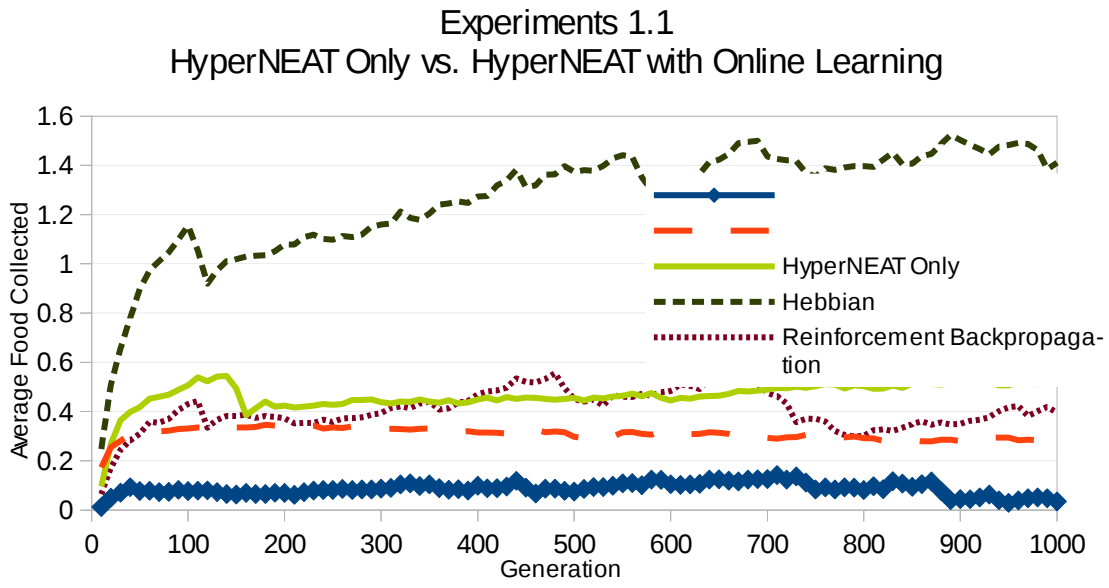


Figure 6.1: Average population performance for Experiments 1.1.

In general, the evolution runs that made use of online learning performed better than

HyperNEAT alone. There is minimal difference in between the Hebbian, reinforcement backpropagation, and temporal difference techniques; the addition of supervised backpropagation performs significantly better than all others, in this respect.

However, this can be slightly misleading, since these results are based on the performance of the population as a whole. In general, when evolving neural networks, the goal is to produce one or a few individuals that perform well. Table 6.1 shows the results of the top performers for each technique.

Table 6.1: Experiments 1.1 Top Performers and Averages.

Experiments 1.1: HyperNEAT Only vs. HyperNEAT with Online Learning				
Technique	**Most Food**	**Avg Best Food**	**StdDev**	**Run Success**
HyperNEAT Only	11.000	9.550	1.023	75.00%
Hebbian	3.000	2.450	0.218	0.00%
Reinforcement Backpropagation	8.667	4.717	1.554	0.00%
Supervised Backpropagation	10.000	8.750	0.924	25.00%
Temporal Difference Reinforcement	10.000	7.150	2.267	30.00%

The “Most Food” column shows the cross-run champion score, that is, the highest score that was achieved by an individual for all twenty evolution runs for each technique. The “Avg Best Food” column is the average score of each evolution run's champion; the next column is the standard deviation of that average. The last column, “Run Success” is the percent of evolution runs that produced a successful individual. Recall from Chapter 5, the success criteria was set to whether an individual was produced that could collect an average of 9.6 pieces of food in 50 timesteps, averaged over 3 evaluations.

These results show that the simple addition of online learning does not increase the

success rate of evolution runs, nor the quality of evolved networks.

6.2. Experiments 1.2: Online Learning with CPPN Generated Parameters

Experiments 1.2 expands the approach used in the first set of experiments by using learning rate parameters produced by the CPPN, instead of fixed rates. This set also adds the Hebbian ABC technique, since the use of variable parameters allows it to be distinct from basic Hebbian. The basic HyperNEAT technique is not re-run for these experiments, since there are no changes that would affect it performance; references in figures and tables below are only for comparison purposes. Figure 6.2 shows the average population results.

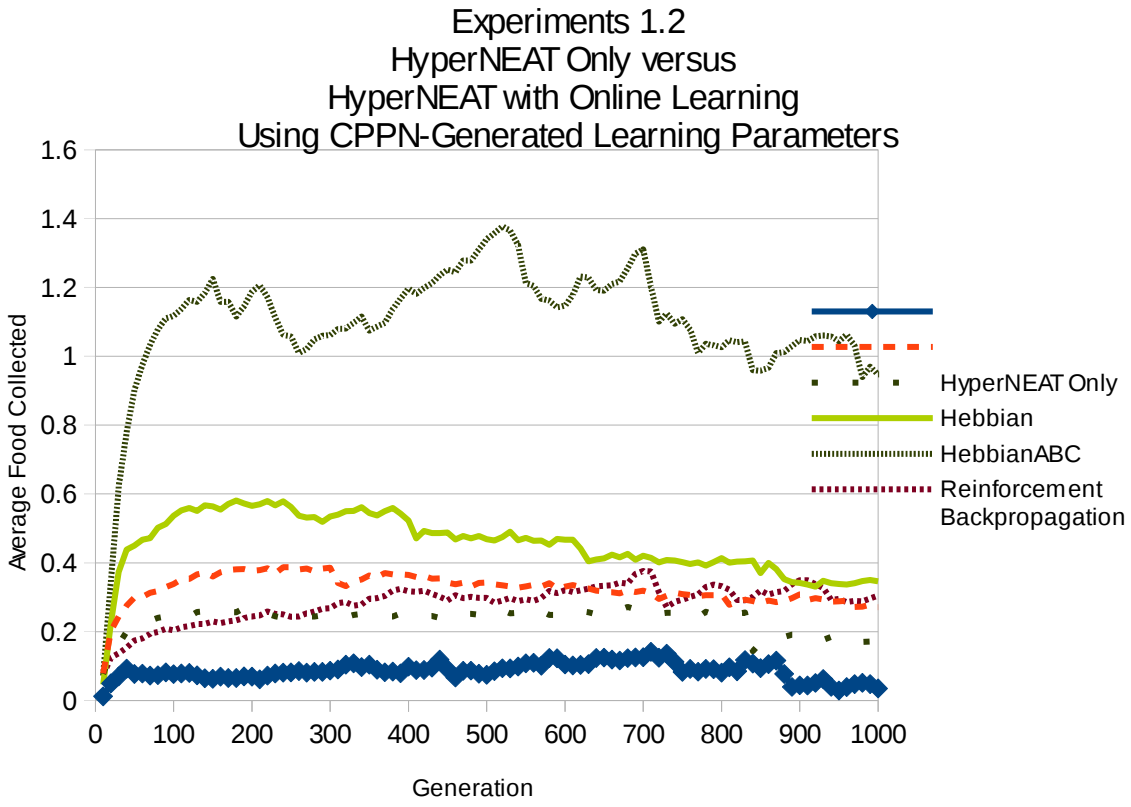


Figure 6.2: Average population performance for Experiments 1.2.

As can be seen, the population level results are very similar to those from experiments 1.1, with supervised backpropagation performing better than other online learning techniques, which in turn perform better than HyperNEAT only.

Table 6.2 shows the results of the top performers for each technique.

Table 6.2: Experiments 1.2 Top Performers and Averages.

Experiments 1.2: HyperNEAT with Online Learning Using CPPN-Generated Learning Parameters				
Technique	**Most Food**	**Avg Best Food**	**StdDev**	**Run Success**
HyperNEAT Only	11.000	9.550	1.023	75.00%
Hebbian	10.000	4.133	2.197	5.00%
HebbianABC	6.000	3.217	1.235	0.00%
Reinforcement Backpropagation	10.000	4.750	1.574	5.00%
Supervised Backpropagation	10.000	8.800	1.166	45.00%
Temporal Difference Reinforcement	11.000	5.700	2.791	20.00%

By allowing the CPPN to select learning rate parameters, several of the techniques were more successful. Notably, Hebbian and reinforcement backpropagation were capable of producing successful evolution runs. As well, the success rate for supervised backpropagation increased. Despite these results, basic HyperNEAT still outperforms its online learning counterparts.

It is worthwhile to note that having the CPPN output learning parameters significantly expands the search space. For the backpropagation and basic Hebbian techniques, each weight and each bias in the network has a (potentially distinct) learning rate; this effectively doubles the search space. The search space for temporal difference and Hebbian ABC and is larger still, with temporal difference having an alpha, lambda, and

gamma for each connection plus an alpha and gamma per-output node, and Hebbian ABC having A, B, C, and n parameters for each connection. This may have prevented the algorithm from finding successful individuals as quickly as HyperNEAT only, where the search space is much smaller. Still, there is value in this experiment. These results demonstrate two things: 1) it is possible to evolve successful individuals by using CPPN selected learning parameters, and 2) that in some cases, this improves the performance of online learning techniques over those using fixed parameters.

6.3. Experiments 1.3: Learning Ability as Fitness

Experiments 1.3 is split into sets A and B. Set A repeats the experiments in 1.1, using learning ability as part of the fitness function. Set B does the same for with the techniques from experiments 1.2.

Figure 6.3 shows the average population results for experiments 1.3A; Figure 6.4 shows the results for experiments 1.3B.

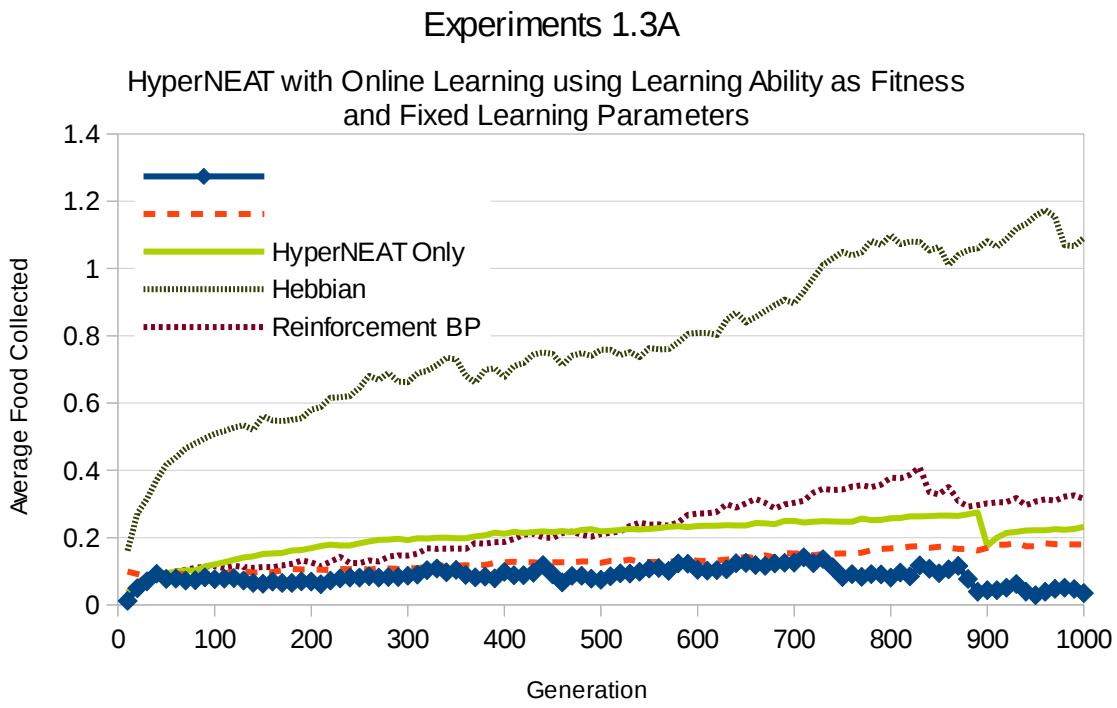


Figure 6.3: Average population performance for Experiments 1.3A.

Experiments 1.3B

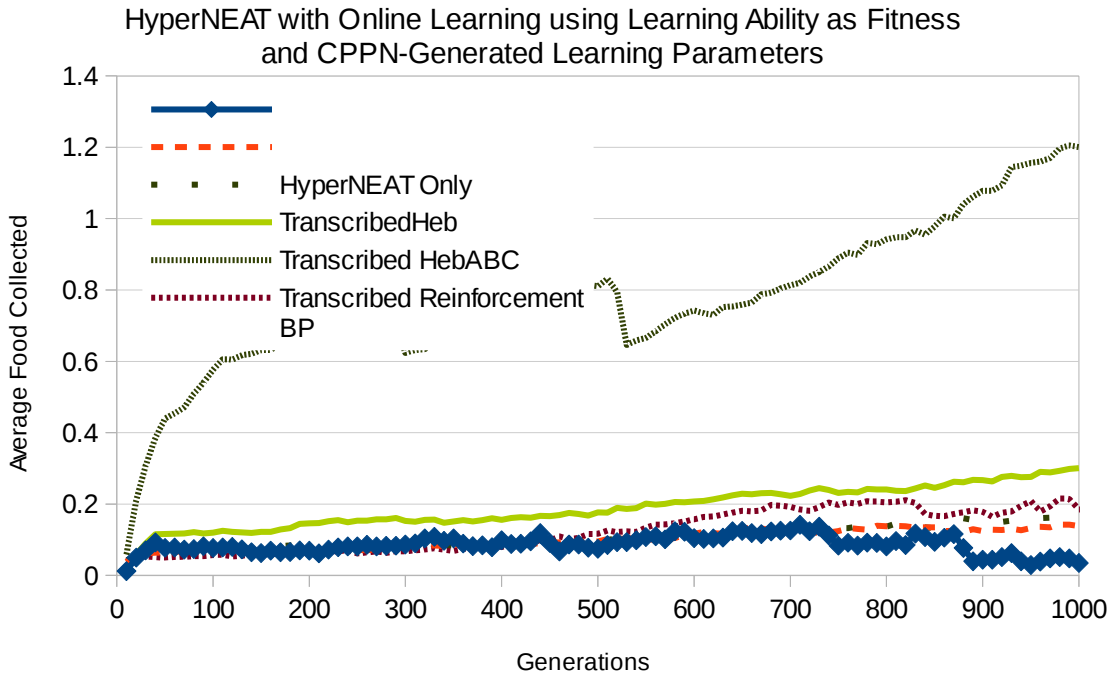


Figure 6.4: Average population performance for Experiments 1.3B.

The population level results for experiments 1.3 A and B show a general upward trend for supervised backpropagation, suggesting that that particular technique might have benefited from having the number of generations for each evolution run increased. Beyond this, there are no real meaningful differences from previous experiments.

Table 6.3 shows the results of the top performers for each technique, for both experiments 1.3 A and B.

Table 6.3: Experiments 1.3 Top Performers and Averages.

Experiments 1.3: HyperNEAT with Online Learning using Learning Ability as Fitness				
Technique	**Most Food**	**Avg Best Food**	**StdDev**	**Run Success**
HyperNEAT Only	11.000	9.550	1.023	75.00%
Fixed Parameters				
Hebbian	2.667	2.200	0.245	0.00%
Reinforcement Backpropagation	6.333	2.950	0.950	0.00%
Supervised Backpropagation	9.667	8.083	1.210	25.00%
Temporal Difference Reinforcement	10.333	5.200	2.301	5.00%
CPPN Generated Parameters				
Hebbian	7.000	2.667	1.183	0.00%
HebbianABC	6.000	2.717	0.979	0.00%
Reinforcement Backpropagation	8.000	3.567	1.207	0.00%
Supervised Backpropagation	9.000	6.467	1.208	0.00%
Temporal Difference Reinforcement	11.000	4.217	2.906	15.00%

As with previous experiments, the online learning techniques do not improve the success rate of evolution versus HyperNEAT alone. In general, the performance is worse than without using learning ability as a fitness measure.

6.4. Observations on CPPN Generated Learning Parameters

After running experiments 1.2 and 1.3, the substrate of each the champions was analyzed, and an interesting phenomenon was observed. In nearly all cases (13 out of 18 champions), online learning was effectively shut off by having the learning rate parameters set to 0. This means that in practice, it was easier for HyperNEAT to find a solution when online learning wasn't part of the equation.

For supervised backpropagation, 3 out of the 9 champions (all from experiment 1.2) had some non-zero learning rates. Two of those had learning rates only for the connection

weights, and one had learning rates for both connections and bias nodes.

For temporal difference, 4 out of 7 champions had some non-zero parameters, however, in two of those cases the *alpha* parameter controlling the actual rate of weight change was set to zero, thus turning off online learning. In one of the other two cases, each of the parameters (alphas, gammas, lambdas) had non-zero values. In the other case, the evolved substrate had non-zero gamma values, and had non-zero alpha values, but only on the lower connection layer (that is, the connections between the input and hidden layers). This means that the temporal difference changes were only applied to the lower connection layer and not the upper layer. Recall from Chapter 2 that the lambda parameter controls the discount applied to past gradients. In this case, lambdas were all set to 0, effectively transforming the algorithm into supervised backpropagation, where the desired output is the output of the network on the immediately subsequent timestep. This is an interesting result, however it may be of little significance, given that it occurred only once.

6.5. Performance of Champions from Experiments 1

Two separate evaluations were devised to test the quality and generality of the evolved champions. The first places each champion in an environment with 15 pieces of food placed randomly throughout; as a piece of food is collected, a new one is introduced in a free random location. They are given 200 timesteps to collect as much food as possible. This is repeated 20 times, and the results averaged across trials. As during evolution, each “champion substrate” actually controls a team of 6 bots.

The second evaluation places the champions in a “sparse” environment where 8 pieces of food are placed periodically near the environment's perimeter. Champions that perform well in this environment should exhibit robust search strategies. Only one trial is performed in the sparse environment. Figure 6.5 shows the arrangement of food and bots in the environment. Green cells with '@' are food. Black cells are empty. Blue cells with '*' are bots. Yellow cells are base cells; note that some base cells are obscured by the bots in this image.

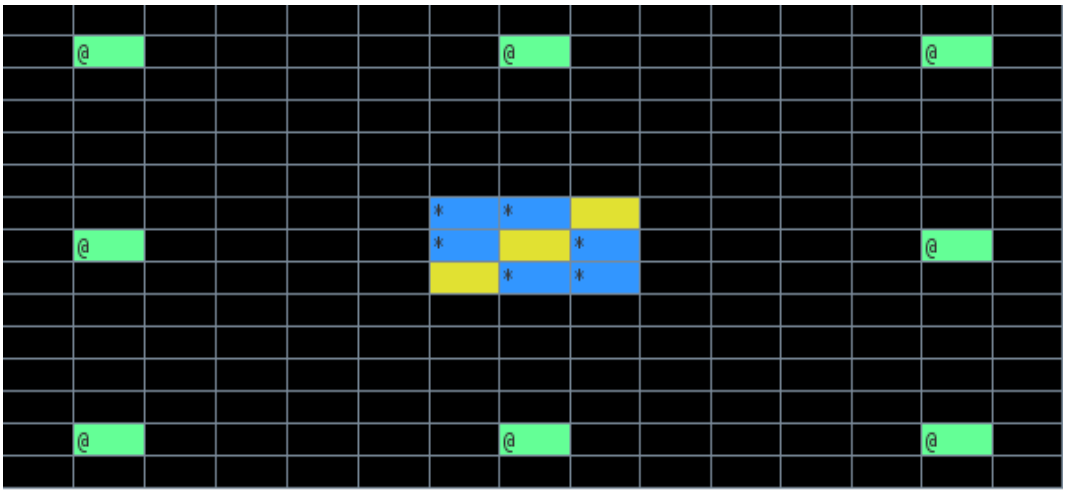


Figure 6.5: The layout of the sparse environment.

For substrates that were evolved with online learning, a pre-training session is executed prior to evaluation in the random and sparse environments. An environment is setup that is identical to the one used in evolution (see Chapter 5.5). The substrate controlled bots are placed in the environment and run for 150 timesteps, with online learning being applied as per the technique used during evolution. This mimics (but does not identically reproduce) the 3 evaluations of 50 timesteps used in the evolution runs, thus giving each the opportunity to be trained similarly to its original evaluation. After pre-training, online

learning is shutoff in preparation for the evaluation environments.

One champion was selected from each of the successful trials, except in a few cases, two were selected. In general, the first, best champion is chosen, i.e., the cross-run champion is selected; if there are multiple champions with the same score, only the first is chosen. In cases where online learning was used with CPPN generated parameters, and the selected champion has zero valued parameters, a second champion is also selected, to highlight any differences between the two cases.

Table 6.4 shows the results of the champions in random environments. An asterisk ('*') indicates that while this champion was evolved with some learning technique, its learning parameters were set to 0 by the CPPN.

Table 6.4: Experiments 1 Champions Performance in Random Environments.

Experiments Set 1: Champion Performance in Random Environments		
Technique	**Avg Food**	**StdDev**
HyperNEAT Only	26.040	7.494
Hebbian, CPPN Gen. Parameters	24.980	7.428
Reinforcement Backprop., CPPN Gen. Parameters	23.760	8.673
Supervised Backprop., Fixed Parameters, Learnability as Fitness	11.920	5.688
Supervised Backprop., Fixed Parameters	7.860	6.309
Supervised Backprop., CPPN Gen. Parameters 1	27.180	11.025
Supervised Backprop., CPPN Gen. Parameters 2	4.980	2.789
T.D., Fixed Parameters	24.700	8.864
T.D., CPPN Gen. Parameters 1	28.140	8.292
T.D., CPPN Gen. Parameters 2	24.700	8.864
T.D., Fixed Parameters, Learniability as Fitness	25.200	7.088
T.D., CPPN Gen. Parameters, Learnability as Fitness 1	31.280	9.531
T.D., CPPN Gen. Parameters, Learnability as Fitness 2	21.640	8.064

The use of random environments can introduce noise into the evaluations, however by

averaging results over multiple trials, this is partially mitigated. Some degree of solution quality and generality can be gleaned from these results. In general, the champions evolved through online learning performed comparably with HyperNEAT only, though it appears those with CPPN generated zero-valued parameters performed slightly better on average than those that had fixed or generated parameters. Surprisingly, with one exception, the champions evolved with supervised backpropagation performed abysmally. This was investigated further to understand why this was the case.

Based on observation of the behavior of the evolved agents, it appears that they had evolved to be highly optimized to the the amount of training received during evolution, and to the layout of the training environment. When pre-trained and placed in the training environment, the bots performed very well, with each of the three teams retrieving an average of over 10 pieces of food for every 50 timesteps.

To determine how reliant the networks were on online training and the environment layout, the random and sparse environment tests were re-run for these three agents with training occurring at every timestep, instead of using the pre-training. Tables 6.5 and 6.6 show the results.

Table 6.5: Supervised Champions in Random Environments with Online Training.

Experiments Set 1: Supervised Champions in Random Environments with Online Training		
Technique	**Avg Food**	**StdDev**
Supervised Backprop., Fixed Parameters, Learnability as Fitness	27.800	10.543
Supervised Backprop., Fixed Parameters	7.850	4.851
Supervised Backprop., CPPN Gen. Parameters 2	25.150	7.458

Table 6.6: Supervised Champions in Sparse Environments with Online Training.

Experiments Set 1: Supervised Champions in the Sparse Environment with Online Training	
Technique	**Food Collected**
Supervised Backprop., Fixed Parameters, Learnability as Fitness	3
Supervised Backprop., Fixed Parameters	3
Supervised Backprop., CPPN Gen. Parameters 2	3

The random test shows a stark contrast for two of the champions. It seems that these networks were very responsive to continuous online training, possibly keeping them adaptive in a shifting environment. The champion evolved with supervised backpropagation with fixed parameters saw no significant difference in its performance. It appears that its evolution had merely tuned it to rely on the specific training environment, and generalized poorly to other environments.

Table 6.7 shows the results of all champions in the sparse environment. An asterisk ('*') indicates that while this champion was evolved with some learning technique, its learning parameters were set to 0 by the CPPN.

Table 6.7: Experiments 1 Champion Performance in the Sparse Environment.

Experiments Set 1: Champion Performance in the Sparse Environment	
Technique	**Food Collected**
HyperNEAT Only	4
Hebbian, CPPN Gen. Parameters	3
ReinforcementBackprop., CPPN Gen. Parameters	3
Supervised Backprop., Fixed Parameters, Learnability as Fitness	4
Supervised Backprop., Fixed Parameters	2
Supervised Backprop., CPPN Gen. Parameters 1	5
Supervised Backprop., CPPN Gen. Parameters 2	2
T.D., Fixed Parameters	3
T.D., CPPN Gen. Parameters 1	3
T.D., CPPN Gen. Parameters 2	3
T.D., Fixed Parameters, Learnability as Fitness	0
T.D., CPPN Gen. Parameters, Learnability as Fitness 1	3
T.D., CPPN Gen. Parameters, Learnability as Fitness 2	3

As the results show, HyperNEAT Only performs better in this evaluation than most of the online learning techniques. Only two of the champions evolved with supervised backpropagation performed comparably or better, and the top performer was in fact one of the ones with zero-valued learning rates.

Based on these results, it seems reasonable to say that while these particular applications of online learning with HyperNEAT are capable of producing effective networks, the larger search space means that this occurs less reliably than and with less quality than the techniques with smaller search spaces, with basic HyperNEAT generally outperforming the others.

6.6. Experiments 2.1: HyperNEAT with Online Backpropagation Variants

Experiments 2.1 compare HyperNEAT with supervised backpropagation, repeated backpropagation, and rotational backpropagation, as described in Chapter 5.

Figure 6.6 shows the population results. This shows the average food collected by each individual in the population, averaged over twenty evolution runs, by generation. The label "Backpropagation_1" is for basic backpropagation; "Backpropagation_4" indicates repeated backpropagation.

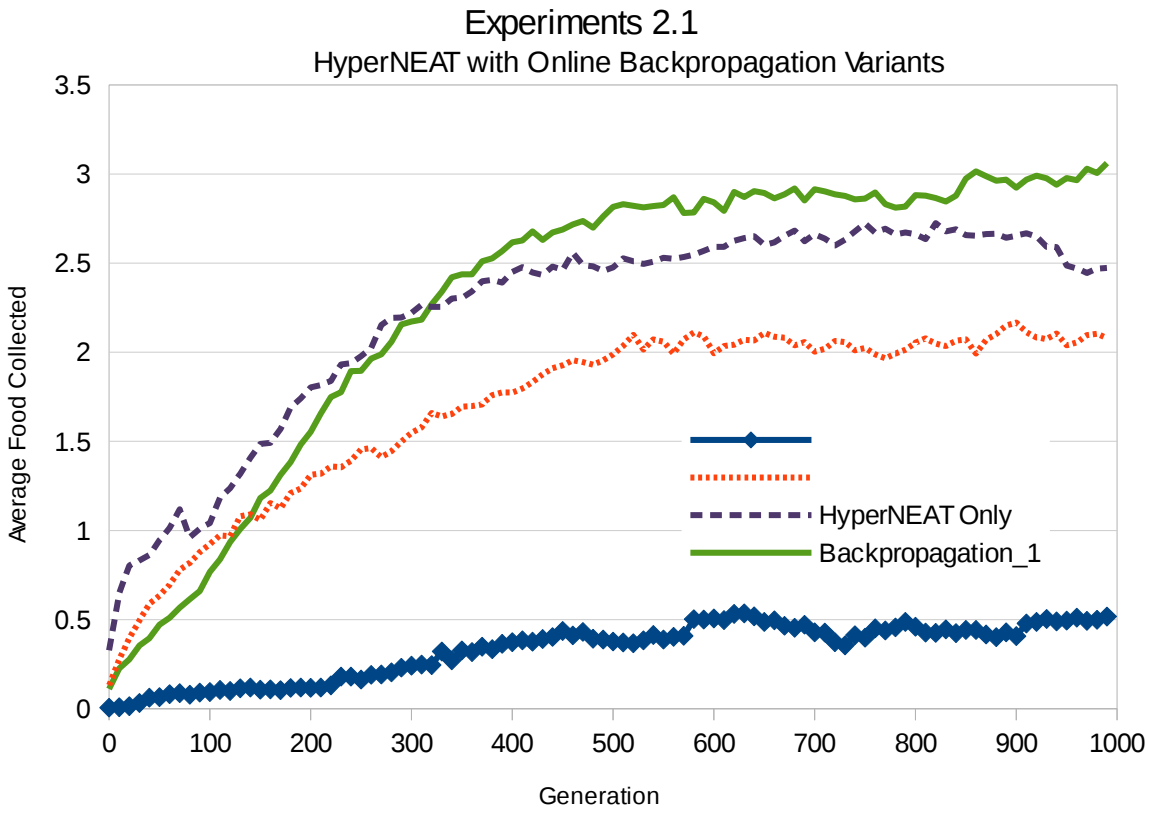


Figure 6.6: Average population performance for Experiments 2.1.

In general, the variants of backpropagation had better performing populations than did

HyperNEAT alone.

Table 6.8 shows the results of the top performers for each technique.

Table 6.8: Experiments 2.1 Top Performers and Averages.

Experiments 2.1: HyperNEAT with Online Backpropagation				
Technique	**Overall Best Performer**	**Best Performer Avg**	**StdDev**	**Perf. >= 9.6**
HyperNEAT Only	12.000	7.650	2.816	40.00%
Backprop. X1	10.333	9.100	0.692	35.00%
Backprop. X4	10.000	9.167	0.734	45.00%
Rotational Backprop.	10.667	9.733	0.478	75.00%

Overall, basic HyperNEAT succeeded in producing the highest performing champion. However each of the backpropagation variants produced higher performing champions on average, and more consistently (based on standard deviation).

This chart includes the percentage of evolution runs that produced an individual that could collect more than 9.6 pieces of food per evaluation, on average. This is somewhat analogous to the "Successful" column used in Experiments 1, although Experiments 2 do not stop evolution after a such performance threshold is reached.

Based on the performance threshold of 9.6, rotational backpropagation performed substantially better than all other techniques; a very encouraging initial result.

Note: The performance of basic HyperNEAT here is slightly different than in experiment 1.1 (i.e., the percent of evolution runs that were capable of producing an individual that was capable of collecting more than 9.6 pieces of food during evaluation). This may be due to two factors: 1) The fitness calculation was simplified in these experiments; 2) 92% of the individuals are used as parents instead of 50% (therefore only 8% of each

generation are brand new individuals as opposed to 50%). Another possibility is that in experiments 1, training stopped when a performance threshold was reached; this may have affected the top performer average, but is unlikely to account for the reduced rate of success in 2.1.

Champions from each set of evolution runs were selected in the same fashion as in Experiments 1, and evaluated in the random and sparse environment test.

Table 6.9 shows the results of the champions in random environments.

Table 6.9: Experiments 2.1: Champion Performance in Random Environments.

Experiments 2.1: Champion Performance in Random Environments		
Technique	**Avg Food**	**StdDev**
HyperNEAT Only	26.600	9.980
Supervised Backprop. x1	15.660	5.559
Supervised Backprop. x4	14.820	9.068
Rotational Backprop.	27.680	7.819

This trial shows that the champions for base HyperNEAT and for HyperNEAT + rotational backpropagation perform similarly, and have generalized better than the non-rotational backpropagation techniques.

Table 6.10 shows the results of the champions in the sparse environment.

Table 6.10: Experiments 2.1: Champion Performance in the Sparse Environment.

Experiments 2.1: Champion Performance in the Sparse Environment	
Technique	**Food Collected**
HyperNEAT Only	3
Supervised Backprop. x1	4
Supervised Backprop. x4	7
Rotational Backprop.	5

In call cases the backpropagation techniques outperformed HyperNEAT only. This is

particularly interesting since the HyperNEAT only champion collected more food in the training environment than did any of the backpropagation variants. This suggests that the HyperNEAT only champion optimized to the training environment, where the others evolved more general search strategies.

6.7. Experiments 2.2: Bootstrapping

As noted in Chapter 4.5, the initial results from experiment 2.1 prompted an effort to see if it was possible to achieve the top performing results with greater frequency and reliability. The goal was to augment the backpropagation techniques in such a way that they could produce champions that performed as well during the initial training as the HyperNEAT only champion, but at a higher and more consistent rate. Experiments 2.2 utilize the bootstrapping approach proposed in previous chapters.

Multiple trials are performed with different fitness cut-off thresholds for bootstrapping (see 4.5 and 5.7.3.2): one each per fitness threshold (50, 100, and 200 respectively), per backpropagation technique. Results are organized by backpropagation technique; experiment 2.2A uses simple backpropagation; 2.2B uses repeated backpropagation; 2.2C uses rotational backpropagation.

Note that the “HyperNEAT Only” results are taken from experiment 2.1, and shown in the charts and tables only for comparison purposes.

Figure 6.7 shows the average population performance for Experiments 2.2A.

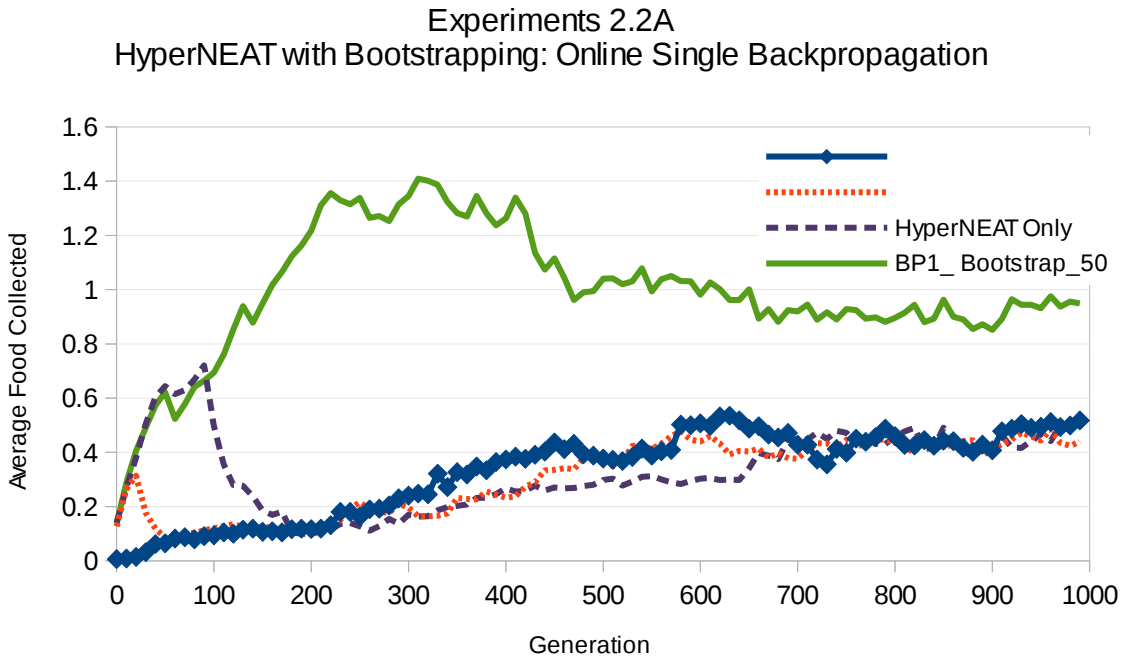


Figure 6.7: Average population performance for Experiments 2.2A.

At the average population level, basic backpropagation performs best when using a bootstrapping threshold of 200; the other thresholds perform no better than HyperNEAT after the first 200 or so generations.

Figure 6.8 shows the average population performance for Experiments 2.2B.

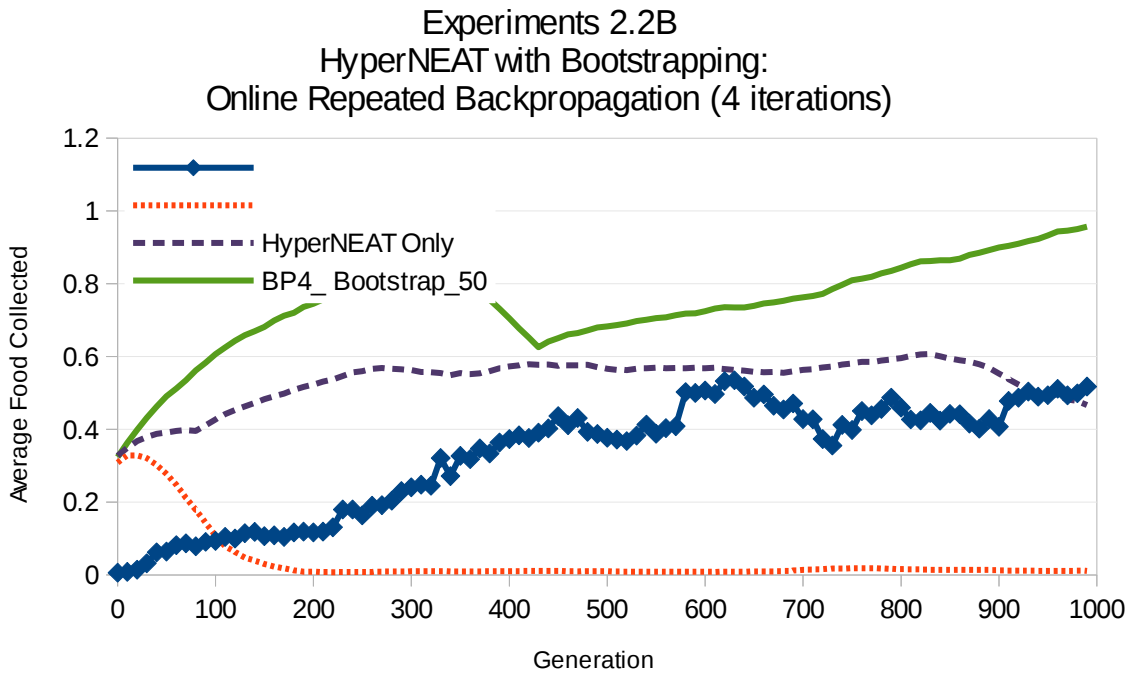


Figure 6.8: Average population performance for Experiments 2.2B.

For repeated backpropagation, the results are slightly different. When the threshold is set to 50, there is a steep drop-off in performance after the first few generations, and a recovery is never made. With a threshold of 100, performance is initially better, but in the long run, basic HyperNEAT catches up. Bootstrapping with a threshold of 200 starts off better and stays better through the end of the run.

Note:The chart above shows a steady decline in performance starting a little after generation 300 and lasting 100 or so generations. In many cases these charts show up's and down's between generations, as cross-run averages show fluctuations in performance. However, in this particular case, and in some other charts showing a steep

drop-off and recovery, this may be attributed to the built-in switch to change the bots from using single bot towing to double bot towing (see 5.2.1). When that switch occurs, the immediately subsequent generations exhibit a sharp drop in performance, with gradual increase as evolution continues. In this case, it appears that the performance threshold that triggered this switch tended to occur after generation 300.

Figure 6.9 shows the average population performance for Experiments 2.2C.

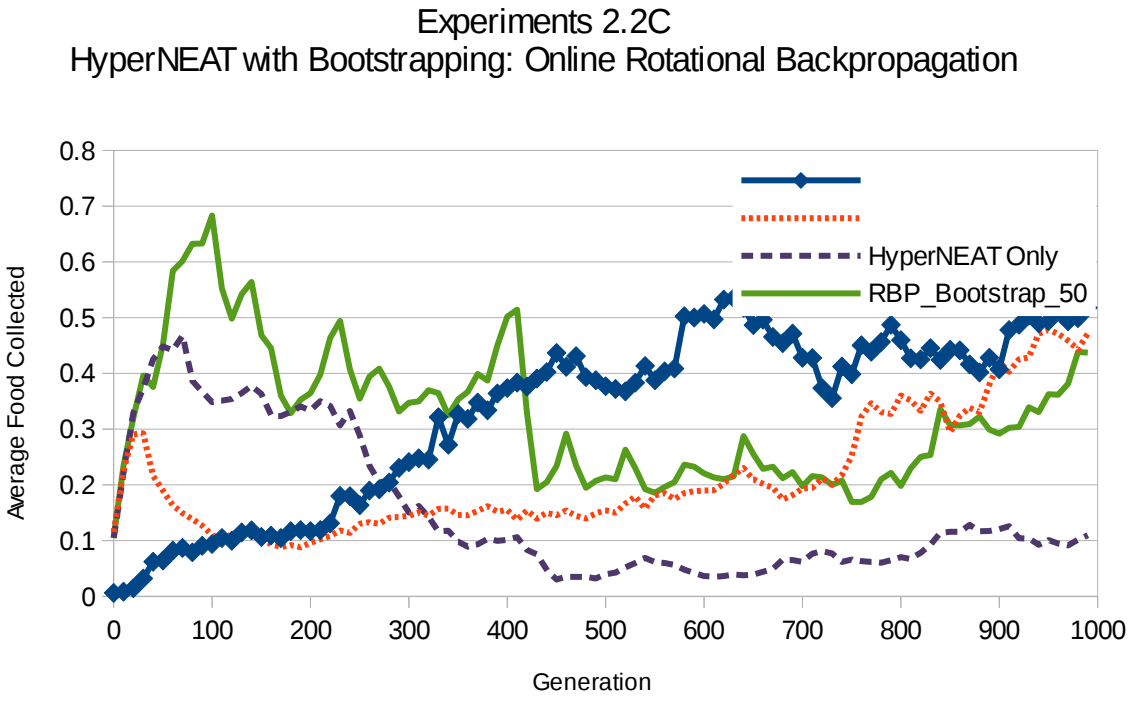


Figure 6.9: Average population performance for Experiments 2.2C.

Population performance for rotational backpropagation seems generally poor after the first few hundred generations, with HyperNEAT only steadily surpassing the other

techniques.

Table 6.11 shows the combined results of top performers for each combination of backpropagation and bootstrapping threshold.

Table 6.11: Experiments 2.2 Top Performers and Averages.

Experiments 2.2: HyperNEAT with Bootstrapping				
Technique	**Overall Best Performer**	**Best Performer Avg**	**StdDev**	**Perf. >=9.6**
HyperNEAT Only	12.000	7.650	2.816	40.00%
BP1_Bootstrapping50	12.000	7.000	3.078	30.00%
BP1_Bootstrapping100	12.000	7.817	2.596	35.00%
BP1_Bootstrapping200	12.000	10.067	1.809	75.00%
BP4_Bootstrapping50	11.000	7.067	2.595	25.00%
BP4_Bootstrapping100	11.000	7.150	2.895	35.00%
BP4_Bootstrapping200	12.000	9.233	2.203	55.00%
RBP_Bootstrapping50	12.000	6.650	2.877	30.00%
RBP_Bootstrapping100	11.000	7.217	1.833	15.00%
RPB_Bootstrapping200	12.000	8.350	1.531	25.00%

As can be seen from these results, bootstrapping tends to perform better with a larger threshold; each instance where a threshold of 200 was used produced a higher champion performance average than did HyperNEAT alone, and these measures increased with each increase of the bootstrapping threshold. As well, the non-rotational backpropagation techniques had a higher rate of individuals surpassing the 9.6 performance threshold. Interestingly, in these experiments rotational backpropagation performs more poorly than the other techniques. In fact, the clear top performer here appears to be basic backpropagation with a bootstrapping threshold of 200, with the highest average champion performance, relatively low standard deviation, and highest percent of successful individuals. This suggests that more iterations of online training are not

always better.

Bootstrapping can allow HyperNEAT to produce top performing individuals more frequently than HyperNEAT alone or even with continuous online training. Based on these results, it appears that through bootstrapping, online training can be used to improve HyperNEAT's ability to select individuals early in evolution that will produce better performing individuals in the long run.

Table 6.12 shows the performance of the champions in random environments.

Table 6.12: Experiments 2.2 Champion Performance in Random Environments.

Experiments 2.2: Champion Performance in Random Environments		
Technique	**Avg Food**	**StdDev**
HyperNEAT Only	26.600	9.980
BP x1, Bootstrapping50	27.380	10.409
BP x1, Bootstrapping100	26.960	8.602
BP x1, Bootstrapping200	28.260	7.825
BP x4, Bootstrapping50	23.620	6.277
BP x4, Bootstrapping100	23.960	7.914
BP x4, Bootstrapping200	26.320	8.803
RotBP, Bootstrapping50	26.740	7.421
RotBP, Bootstrapping100	15.120	9.941
RotBP, Bootstrapping200	29.920	9.915

All champions perform within a few points of one another, with the exception of rotational backpropagation with a bootstrap threshold of 100.

Table 6.13 shows the performance of the champions in sparse environments.

Table 6.13: Experiments 2.2 Champion Performance in the Sparse Environment.

Experiments 2.2: Champion Performance in the Sparse Environment	
Technique	**Food Collected**
HyperNEAT Only	3
BP x1, Bootstrapping50	3
BP x1, Bootstrapping100	5
BP x1, Bootstrapping200	4
BP x4, Bootstrapping50	5
BP x4, Bootstrapping100	3
BP x4, Bootstrapping200	5
RotBP, Bootstrapping50	3
RotBP, Bootstrapping100	3
RotBP, Bootstrapping200	4

In the sparse environment, each technique performs at least as well as HyperNEAT alone.

6.8. Experiments 2.3: Training Banks

Experiments 2.3 use training banks as discussed in Chapters 4.6 and 5.7.3.3. The experiments are divided into 2.3A, using training banks that train for 4 epochs, and 2.3B, that train for 8 epochs. Online training is not used for these experiments; training only occurs between evaluations using the data accumulated in the training bank. The same backpropagation techniques used in experiments 2.1 and 2.2 are used as the training method for applying the data in the training banks.

Figure 6.10 shows the cross-run average population performance for experiments 2.3A.

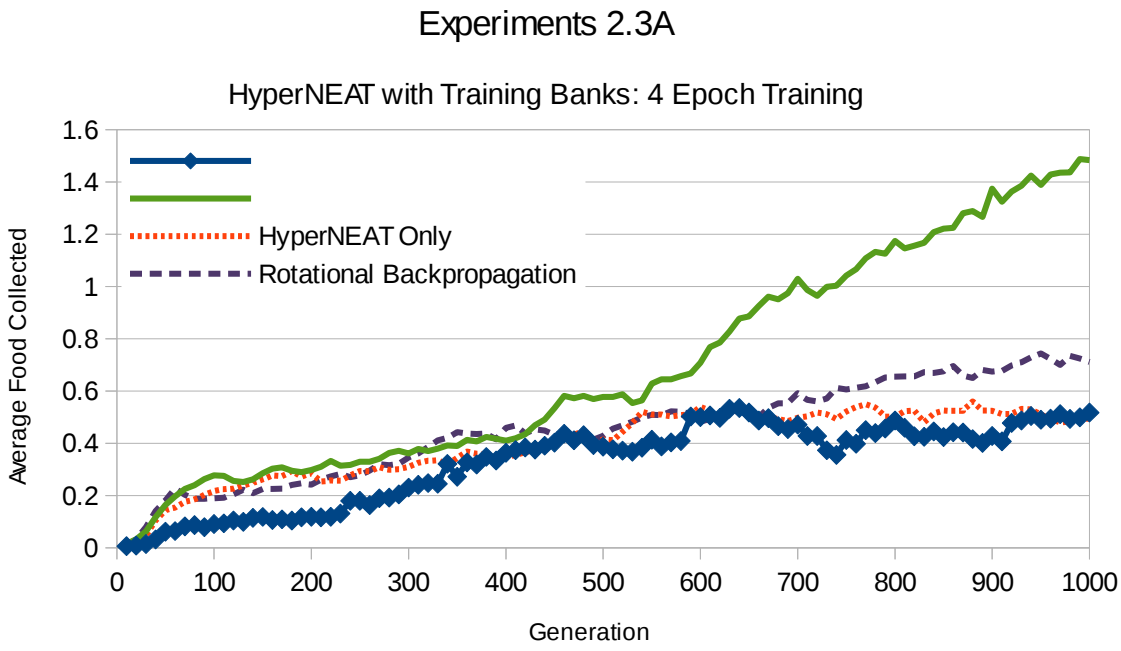


Figure 6.10: Average population performance for Experiments 2.3A

Figure 6.11 shows the cross-run average population performance for experiments 2.3A.

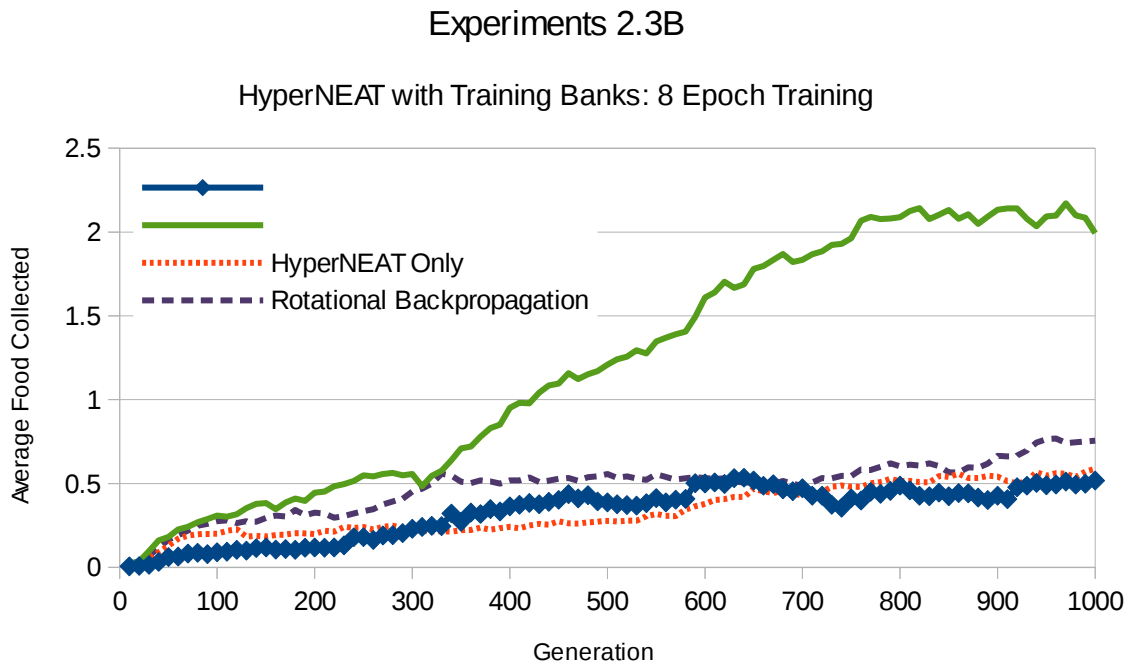


Figure 6.11: Average population performance for Experiments 2.3B

In both experiments 2.3A and B, training banks using rotational backpropagation perform better at the population level than the other techniques, which perform at very similar levels.

Table 6.14 shows the combined results of top performers for each combination of backpropagation and training banks.

Table 6.14: Experiments 2.3 Top Performers and Averages.

Experiments 2.3: HyperNEAT with Training Banks				
Technique	**Overall Best Performer**	**Best Performer Avg**	**StdDev**	**Perf. >9.6**
HyperNEAT Only	12.000	7.650	2.816	40.00%
BP x1, TrainingBank 4	10.000	5.450	2.671	20.00%
BP x1, TrainingBank 8	10.333	5.383	2.591	15.00%
BP x4, TrainingBank 4	10.000	5.633	2.814	10.00%
BP x4, TrainingBank 8	10.667	5.217	2.704	10.00%
RotBP, TrainingBank 4	10.667	7.833	2.880	45.00%
RotBP, TrainingBank 8	11.000	9.233	2.216	75.00%

The champion results from these experiments correlate roughly to the population level results. In general, using training banks with basic and repeated backpropagation perform no better than basic HyperNEAT; they are capable of producing champions that come close to those produced by HyperNEAT alone, but do so with less frequency. Although the rotational backpropagation technique does not produce an overall champion that quite matches the best from basic HyperNEAT, it produces higher performing champions on average, and with greater frequency.

The champions for each technique are evaluated using random and sparse environments. Similar to previous experiments, champions are pre-trained. Since these experiments performed training in an offline fashion, pre-training takes place in the same way – each substrate is placed in the training environment for 3 periods of 50 timesteps, with training occurring between each period.

Table 6.15 shows the results of the champions in random environments.

Table 6.15: Experiments 2.3 Top Performers and Averages.

Experiments 2.3: Champion Performance in Random Environments		
Technique	**Avg Food Collected**	**StdDev**
HyperNEAT Only	26.600	9.980
BP x1, 4 Epoch Training	12.250	6.441
BP x1, 8 Epoch Training	26.200	9.621
BP x4, 4 Epoch Training	21.900	6.811
BP x4, 8 Epoch Training	26.700	9.587
RotBP, 4 Epoch Training	28.750	5.243
RotBP, 8 Epoch Training	29.700	8.355

In the random environment evaluation, training individuals for 8 epochs performed slightly better than those trained for 4 epochs. When trained with basic and repeated backpropagation for 8 epochs, performance was similar to HyperNEAT alone. The individuals trained with rotational backpropagation performed slightly better with either 4 or 8 training epochs.

Table 6.16 shows the results of the champions in the sparse environment.

Table 6.16: Experiments 2.3 Champion Performance in the Sparse Environment.

Experiments 2.3: Champion Performance in the Sparse Environment	
Technique	**Food Collected**
HyperNEAT Only	3
BP x1, 4 Epoch Training	2
BP x1, 8 Epoch Training	3
BP x4, 4 Epoch Training	4
BP x4, 8 Epoch Training	3
RotBP, 4 Epoch Training	8
RotBP, 8 Epoch Training	8

The use of training banks with rotational backpropagation was the only technique that

enabled the collection of all 8 pieces of food in the sparse environment. This suggests that these champions had evolved the best strategy for searching the environment.

7. CONCLUSIONS AND FUTURE WORK

7.1. Experiments 1.1

All of the online training techniques improve the fitness of the population as a whole; this is expected since the training generally pushes individuals toward behaviors that are captured in the fitness calculation. However, the champion results show that the addition of online training does not produce better performing champions and produces slightly-poorer performing champions with substantially less frequency than HyperNEAT alone.

This is a somewhat interesting result in and of itself. Hinton and Nowlan [27] observed that even by introducing random changes to a genetic algorithm chromosome during its lifetime is capable of finding solutions more quickly than evolution alone. Similarly, Parisi and Nolfi [16] claimed that applying learning to chromosomes during evolution inevitably leads to finding a solution faster. The idea behind this is that by changing the network weights, more of the search space around the chromosome is explored at the time of evaluation than would be through evolution alone. Thus two individuals that perform the same but exist in different areas of the search space could be differentiated, if one is actually closer to the global maximum than the other. However, it appears that at least in some circumstances these claims are not completely accurate with regards to adding online learning to HyperNEAT; it is not clear why this is the case.

7.2. Experiments 1.2

Using HyperNEAT's CPPN to produce learning parameters enables some learning techniques to be successful where they failed in the first experiments. This was accomplished by the CPPN learning to "turn-off" online training in some cases. However, in other cases the CPPN produced meaningful values for learning parameters, and these cases produced individuals that were able to achieve performance comparable to the champions of basic HyperNEAT. Overall, the addition of CPPN-generated learning parameters did not increase the success rate of the online learning techniques versus HyperNEAT only. This is likely due to the increase in the size of the search space. Still, the results demonstrate that the use of CPPN-generated learning parameters can lead to successful solutions, even with complex algorithms with multiple parameters such as temporal difference.

7.3. Experiments 1.3

In general, it does not appear that using learning ability as fitness has any positive effect on the frequency of success or the quality of solutions found. Indeed, in most cases, the online learning techniques performed more poorly than in the previous experiments.

A possible explanation is that those individuals that benefit the most from online training tend to be the ones that perform very poorly. While their performance in the various fitness metrics may improve as a result of training, they may still perform poorly at the primary objective. On the other hand, an individual that has a good chromosome and starts off as a good performer would see little benefit to online training. In fact, if the

individual has a strategy that is better than is used in heuristic training techniques, its performance may suffer, and would therefore be ranked lower than it would have otherwise. Thus using learning ability as fitness may not be viable for long term performance.

7.4. Experiments 2.1

The results from experiments 2.1 show that each of the online learning techniques produces a higher performance average of champions across evolution runs than HyperNEAT alone. Further, there tends to be much less variance in the performance of these champions. The addition of rotational backpropagation produces the champions with the highest average performance and with the lowest variance. It also has the highest success rate, 30% higher than the next most successful technique. When compared to repeat training backpropagation, this also suggests that rotational backpropagation benefits HyperNEAT by more than just additional iterations of training.

HyperNEAT alone produced the highest performing champion out of each technique. This suggests that it was capable of finding a champion with a strategy that was superior to the heuristic strategy used in backpropagation training. However, based on the champion's performance in the sparse environment, it appears that performance was optimized for the environment encountered during evolution, and did not generalize as well as those evolved with online learning.

7.5. Experiments 2.2

The bootstrapping technique appears to successfully find a balance between using heuristic backpropagation and pure HyperNEAT. In general, the use of a fitness threshold of 200 allows each technique to produce champions with performance equivalent to HyperNEAT alone, but with a higher cross-run champion average and lower variance than HyperNEAT alone. Where basic backpropagation and repeated backpropagation are used, the success rate is also higher. This threshold appears to be sufficient to allow HyperNEAT to benefit from online learning, but still find strategies that are more optimal than the heuristic.

Interestingly, it appears that basic backpropagation actually outperforms the other backpropagation variants in both training and random environments, suggesting that in this case more training is not necessarily better. Coupled with the fitness threshold of 200, it also surpasses rotational backpropagation without bootstrapping from experiments 2.1 in terms of champion performance average and overall top performer, demonstrating that online learning with bootstrapping is capable of generating top performers at a higher rate than HyperNEAT with continuous online learning.

7.6. Experiments 2.3

While basic backpropagation and repeated backpropagation appear to gain no benefit from being used with the intermittent training strategy used in these experiments, rotational backpropagation excels in certain measures of performance. The use of intermittent training with rotational backpropagation produces better top performing

champions than with rotational backpropagation while maintaining a high rate of success. This technique also shows the highest generalization, with excellent performance in random environments and being the only one to achieve the highest possible score in the sparse environment.

Rotational backpropagation seems to benefit from more training epochs (8 vs. 4) whereas it seems very slightly detrimental to basic and repeated backpropagation.

The results reinforce the viability of rotational backpropagation, furthering the notion that it has some synergistic effect on HyperNEAT substrates beyond the simple increase in the amount of training received. It also shows better generalizability than other techniques.

7.7. Future Work

The breadth of techniques explored in this research offer a number of opportunities for future research.

The results of experiment 1.2 show that the HyperNEAT CPPN can be extended to output learning parameters for use with online learning. This technique is successful, however it does not improve upon the base performance of HyperNEAT alone. It does encourage further research into CPPN extension. The result that learning parameters were set to zero in some cases doesn't support the use of online learning with HyperNEAT, but it does show that extending how the CPPN is used can enable HyperNEAT to overcome certain problems during evolution. A further research might uncover other ways the CPPN can be extended.

Experiments 2.1-2.3 show that rotational backpropagation performs quite well in many cases, with good generalization. Future research might delve into how other geometric translation strategies could impact training on HyperNEAT substrates, or even other types of networks. Further research in this area could help prove out the assumption such techniques synergize with substrate symmetry.

Rotational backpropagation performed particularly well when used with intermittent training, producing individuals with superior search strategies. Further research might explore why the other techniques did not respond as well to intermittent training, or what factors allow rotational backpropagation to succeed in this scenario.

Because intermittent training made use of a training bank of samples gathered during the lifetime of an individual, it might be suited to providing ongoing training to evolved individuals that are used in environments that are different than the ones they trained in, or environments that are prone to periodic shifting.

Despite its success in experiments 2.1 and 2.3, rotational backpropagation performed more poorly than basic backpropagation when it was used with bootstrapping. Given that basic backpropagation was the clear winner in that experiment when a bootstrapping fitness threshold of 200 was used, it raises the question of exactly what mechanism allowed a smaller amount of online training, but for a longer period of time to be the most successful.

The difference in performance between HyperNEAT only and with online learning techniques differs between experiment set 1 and set 2. The differences in configuration

between the sets of experiments are relatively small, yet have significant impact. As noted in 5.6, 5.7, and 6.1, the fitness calculation, the number of parents from each generation, and whether to stop or continue evolution when a performance threshold is reached are the only real differences in how the algorithm executes evolution. However, the performance of basic HyperNEAT differs significantly between the two sets. This is notable, because it also likely impacts the performance of the online learning techniques. Viewing the results of set 1, it might be easy to conclude that online learning was of no value to HyperNEAT. Yet, set 2 shows promising results when online or intermittent training are applied in certain ways. This leads to the question: under what circumstances does online learning benefit or inhibit HyperNEAT? It is possible that online learning may be of benefit when the task to be performed is complex, and/or the fitness calculation does not guide evolution very progressively. Consider that for set 1, the fitness calculation incorporated a number of metrics, it may be the case that it provided a very smooth performance gradient for HyperNEAT to follow. Similarly, in set 2, the fitness calculation was much simpler, but may not have provided a smooth performance gradient; the use of online learning may have aided HyperNEAT in overcoming this difficulty. This effect might be especially pronounced in environments involving complex tasks, where it takes a certain sequence of steps to succeed. This seems a reasonable possibility, however it will require additional research to establish if this is actually the case.

APPENDIX A: CODE FOR EXPERIMENTS

The experiments in this research utilized various extensions to the HyperNEAT algorithm.

The original implementation, called “Another HyperNEAT Implementation” (AHNI), was written by Oliver Coleman and can be found at:

https://github.com/OliverColeman/ahni

This code was extended to support the following functionality:

1) Substrates with arbitrary layer dimensions
2) Substrates that store learning rate parameters, for use in other training algorithms
3) CPPNs that transcribe learning rate parameters to the substrates
4) CPPNs that may transcribe multiple substrates
5) Support for online training algorithms: backpropagation, Hebbian learning and variant, temporal difference learning.

These extensions, collectively referred to as AHNI-ND were written by Shaun Lusk, the author of this research, and are available at:

https://code.google.com/p/ahni-nd/

The code for the gathering environment and the experimental setup and configurations was also written by Shaun Lusk and is available at:

https://code.google.com/p/thynwor/

The projects for AHNI-ND and the gathering environments are licensed for use under the GNU GPL v2.

APPENDIX B: CONFIGURATION VALUES

The configuration settings used for running ANHI and the simulation environment are shown below, in the format of a standard Java properties file.

```
###########
# evolution
###########
num.runs=20
num.generations=1000
popul.size=100
performance.target=2.0
performance.target.type=higher

#topology mutation scheme
#false means mutation probabilities are applied to all possible places a
mutation could occur
#true means probabilities apply to individual as a whole; only one
topological mutation can occur per individual
#note that this applies only to topological mutations, not weight mutations
topology.mutation.classic=true
#topology.mutation.classic=false

#classic=[0.01, 0.5], not classic=[0.0001,] dependent on pop size. 0.03
add.neuron.mutation.rate=0.25
#add.neuron.mutation.rate=0.01
#classic=[0.01, 0.5], not classic=[0.0001,] dependent on pop size. 0.4
add.connection.mutation.rate=0.5
#add.connection.mutation.rate=0.010
#[0.01, 0.3]
remove.connection.mutation.rate=0.02
```

```
#only remove weights with magnitude smaller than this
remove.connection.max.weight=50


#should be 1.0
prune.mutation.rate=1.0


#[0.1, 0.8]. 0.5, 0.6
weight.mutation.rate=0.1
#[1.0, 2.0] dependent on weight.max/min?
weight.mutation.std.dev=1.0


#percent of individuals used as parents
survival.rate=0.92 # use 0.5 for experiments 1
#proportion of sexual (crossover) versus asexual reproduction
crossover.proportion=0.5


#selector.elitism=true
#[1, 5]
selector.elitism.min.specie.size=1
#percent of individuals from each species copied to next generation
unchanged
selector.elitism.proportion=0.1
#min number to select from a species (if it has size >=
selector.elitism.min.specie.size)
selector.elitism.min.to.select=1
selector.roulette=false
selector.max.stagnant.generations=99999
selector.speciated.fitness=true


############
```

```
# speciation
############
#species distance factors
#c1, excess genes factor [1.0, 2.0]
chrom.compat.excess.coeff=2.0
#c2, disjoint genes factor [1.0, 2.0]
chrom.compat.disjoint.coeff=2.0
#c3, Weight difference factor [0.2, 3.0]
chrom.compat.common.coeff=1.0

#compatability threshold [0.1, 4.0], relative to c#
speciation.threshold=2.1
#target number of species to try and maintain by altering speciation
threshold.
speciation.target=15
speciation.threshold.change=0.08

##################
# fitness function
##################
fitness_function.class=org.slusk.thynwor.neat.BetterSasas
fitness.target=1
#max threads to use for fitness evaluation (including transcription of
genotype/cppn to phenotype/substrate)
#if value is <= 0 then the detected number of processor cores will be used
fitness.hyperneat.max_threads=8
#if scale.factor > 1 then the substrate height, width and connection.range
#will be multiplied by scale.factor every time scale.fitness is reached, at
#most scale.times times. If the
fitness.hyperneat.scale.factor=1
```

```
fitness.hyperneat.scale.times=0
fitness.hyperneat.scale.fitness=0.95
fitness.hyperneat.scale.recordintermediateperformance=false


################
# CPPN/AnjiNet #
################
#input and output size determined by hyperneat settings
#stimulus.size=7
#response.size=1
initial.topology.activation=random
initial.topology.fully.connected=true
initial.topology.num.hidden.neurons=0
initial.topology.activation.input=linear
initial.topology.activation.output=linear
recurrent=disallowed
recurrent.cycles=1
#[1, 500]
weight.max=3
weight.min=-3


####################
# HyperNEAT/GridNet #
####################
ann.type=modularHyperNeat
ann.hyperneat.activation.function=sigmoid
ann.hyperneat.feedforward=true
#ann.hyperneat.cyclesperstep=4  not required for feed forward
ann.hyperneat.enablebias=true
ann.hyperneat.includedelta=true
```

```
ann.hyperneat.includeangle=true
ann.hyperneat.useinputlayerencoding=true

ann.hyperneat.connection.expression.threshold=0.2
ann.hyperneat.connection.range=999
ann.hyperneat.connection.weight.min=-2
ann.hyperneat.connection.weight.max=2

ann.hyperneat.depth=3
#ann.hyperneat.height=21,6,1    # now loads arbitrary dimensions from
descriptor, below
#ann.hyperneat.width=21,6,1

ann.hyperneat.use.ndimensional.layers=true
ann.hyperneat.topology.descriptor.file=configs/sasas.xml
ann.hyperneat.topology.max.dimensions=3

ann.hyperneat.enable.learning.rate=false            # Most techniques will
use true

# learning.type = NONE | BACKPROPAGATION | HEBBIAN | HEBBIANABC
#     | TEMPORALDIFFERENCE | MULTIBP | ROTATIONALBP3D
ann.hyperneat.online.learning.type=NONE

ann.hyperneat.enable.learning.rate.decay=false

# For modular transcription
ann.hyperneat.modular.transcriber.class=com.anji.hyperneat.modular.ModularHy
perNeatMultiNetTranscriber
```

```
#####################
# Environment Props     #
#####################
MapWidth=15
MapHeight=15
SimSpeed=400
Environment.initialFood=25
Environment.maintainFood=10
Environment.numDrones=6
Environment.singleAgentTowMode=false          #overridden depending on
current fitness
Environment.obstacleCount=0
Environment.useOnlineReinforcement=true#overridden with false for basic
HyperNEAT
Environment.multiNetMode=true
Environment.useRandomLayout=false
Environment.layoutName=configs/layout.xml
#Environment.onlineLearningType = NONE | SUPERVISED | REINFORCEMENT
Environment.onlineLearningType=NONE    # Varies by technique used.
#Whether to use separate nets depending on type of task
Environment.useTaskNet=true
Drone.sightRange=4
Drone.actionWaitLimit=4
Drone.strength=1
Drone.memoryLimit=6
Food.weight=2


#############################################################
# Single Agent Sensory Array Substrate Fitness Function fitness shaping
factors
```

```
#############################################################
#For Experiments 1 only
sasasFF.foodCollectedFactorStart=0.4
sasasFF.foodCollectedFactorEnd=0.7
sasasFF.moveFoodTowardGoalFactorStart=0.25
sasasFF.moveFoodTowardGoalFactorEnd=0.5
sasasFF.moveTowardSignalFactorStart=0.4
sasasFF.moveTowardSignalFactorEnd=0.25
sasasFF.assistWithFoodFactorStart=0.6
sasasFF.assistWithFoodFactorEnd=0.4
sasasFF.attachedToFoodFactorStart=0.9
sasasFF.attachedToFoodFactorEnd=0.05
sasasFF.handsFullFactorStart=0.001
sasasFF.handsFullFactorEnd=0.2
sasasFF.hitWallFactorStart=0.1
sasasFF.hitWallFactorEnd=0.4
sasasFF.droppedFoodFactorStart=0.05
sasasFF.droppedFoodFactorEnd=0.4
```

REFERENCES

[1] U. Neisser et al., "Intelligence: Knowns and Unknowns," *American Psychologist*, vol. 51, no. 2, pp. 77-101, Feb. 1996.

[2] E. Rich, and K. Knight, *Artificial Intelligence*, 2nd ed. New Delhi, India: Tata McGraw-Hill, 1991.

[3] N. R. Carlson, *Foundations of Physiological Psychology*, 6th ed. Auckland, New Zealand: Pearson Education New Zealand, 2005.

[4] K. Mehrotra et al., *Elements of Artificial Neural Networks*. Cambridge, MA: The MIT Press, 2000.

[5] K. Hornik "Approximation Capabilities of Multilayer Feedforward Networks," *Neural Networks*, vol. 4, no. 2, pp. 251–257, 1991.

[6] C. Bishop, *Pattern Recognition and Machine Learning*. New York, NY: Springer, 2006.

[7] A. Grüning, “Elman backpropagation as reinforcement for simple recurrent networks,” *Neural Computation*, vol. 19, no. 11, pp. 3108-3131, November 2007.

[8] R. Sutton, “Learning to predict by the methods of temporal differences,” *Machine Learning*, vol. 3, no. 1, pp. 9-34, Aug. 1988.

[9] J. McClelland, “Temporal-Difference Learning", in *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises*, 2013. Available: http://www.stanford.edu/group/pdplab/pdphandbook/

[10] R. Sutton and A. Barto, Reinforcement Learning: An Introduction, Cambridge, MA: MIT Press, 1998. Available: http://webdocs.cs.ualberta.ca/~sutton/book/the-book.html

[11] J. Koza, *Genetic Programming*, Cambridge, MA: The MIT Press, 1998.

[12] K. Stanley and R. Miikkulainen, “Evolving Neural Networks through Augmenting Topologies,” *Evolutionary Computation*, vol. 10, no. 2, pp. 99-127, Summer 2002.

[13] K. Stanley and R. Miikkulainen, "Competitive Coevolution through Evolutionary Complexification," *J. Artificial Intelligence Research*, vol. 21, pp. 63-100, Feb. 2004.

[14] J. Gauci and K. Stanley, "Autonomous Evolution of Topographic Regularities in Artificial Neural Networks," *Neural Computation*, vol. 22, no. 7, pp. 1860-1898, May 2010.

[15] J. Clune et al., "The sensitivity of HyperNEAT to different geometric representations of a problem," in *Proc. Genetic and Evolutionary Computation Conf.*, Montreal, Quebec, Canada, 2009, pp. 675-682.

[16] D. Parisi and S. Nolfi, “The Influence of Learning on Evolution,” in *Adaptive Individuals in Evolving Populations: Models and Algorithms*, R. K. Belew and M. Mitchell Eds. Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 1996, pp. 419-428.

[17] P. McQuesten and R. Miikkulainen, “Culling and teaching in neuro-evolution,” in *Proc. 7th Int. Conf. on Genetic Algorithms*, East Lansing, MI, 1997, pp. 760–767.

[18] K. Stanley et al., “Evolving Adaptive Neural Networks with and without Adaptive Synapses,” in *Proc 2003 IEEE Congr. on Evolutionary Computation*, 2003, pp. 2557-2564.

[19] D. D’Ambrosio and K. Stanley, “Generative Encoding for Multiagent Learning,” in *Proc. Genetic and Evolutionary Computation Conf.*, Atlanta, GA, 2008, pp. 819-826.

[20] D. D’Ambrosio et al., "Task Switching in Multirobot Learning through Indirect Encoding," in *Proc. of the Int. Conf. on Intelligent Robots and Systems*, San Francisco, CA, 2011, pp. 2802-2809.

[21] J. Pugh et al., “Directional Communication in Evolved Multiagent Teams ” Univ. Central Florida Dept. EECS, Tech. Rep. CS-TR-13-04, June 10, 2013.

[22] S. Risi, K. Stanley, “Indirectly Encoding Neural Plasticity as a Pattern of Local Rules,” in *Proc. 11th Int. Conf. on Simulation of Adaptive Behavior*, Paris, France, 2010, pp. 533-543.

[23] C. Christenson, "Evolving Learning Neural Networks," M.S. Thesis, Dept. Computer Science, Texas State Univ., San Marcos, TX, 2004.

[24] E. Kumar, "A New Hybrid Learning Algorithm for Drifting Environments," M.S. Thesis, Dept. Computer Science, Texas State Univ., San Marcos, TX, 2005.

[25] A. Dziuk and R. Miikkulainen, "Creating Intelligent Agents through Shaping of Coevolution," in *Proc. Congr. on Evolutionary Computation*, New Orleans, LA, 2011, pp. 1077-1083.

[26] J. Schrum and R. Miikkulainen, "Evolving Agent Behavior In Multiobjective Domains Using Fitness-Based Shaping," in *Proc. Genetic and Evolutionary Computation Conf.*, Portland, Oregon, 2010, pp. 439-446.

[27] G. E. Hinton and S. J. Nowlan, "How Learning Guides Evolution," *Complex Systems*, 1, pp. 495-502, 1987.