Comparison of NEAT and HyperNEAT on a Strategic Decision-Making Problem

Jessica Lowell, Kir Birger, and Sergey Grabkovsky

CS 6140 Final Project

Abstract

Neuroevolution is a useful machine learning approach for problems with limited domain knowledge, but it has not done well with strategic decision-making problems, where the correct action varies sharply as the agent moves across states. Two promising neuroevolution algorithms are NeuroEvolution of Augmenting Topologies (NEAT) and its extension, HyperNEAT. We compare the performance of these two algorithms on a benchmark problem, Keepaway Soccer, that requires strategic decision-making. Our results demonstrate that HyperNEAT outperforms NEAT on a simple instance of the problem but that its advantage disappears when the problem is complicated.

1. Introduction

1.1 Genetic Algorithms

Genetic algorithms (De Jong, 1975; Goldberg, 1989) are a class of search heuristics inspired by the process of evolution in nature. In natural selection, a key mechanism of biological evolution, individuals which are more fit for the conditions in which they live are "selected" to procreate, by being more likely than less fit individuals to survive to maturity and find a mate. Genetic algorithms apply this concept to a search space, where the "individuals" are points in the search space representing possible solutions.

The algorithm begins with a first generation of candidate solutions, which is usually randomly generated. A fitness function evaluates the fitness of each candidate, and the algorithm selects a subset of the candidates based on fitness. These candidates "mate" with each other by recombining (exchanging parts), in order to form a new generation of candidate solutions with a higher average fitness. Depending on the particular algorithm, they also may be mutated (have their parameters randomly perturbed) to expand the search process to new areas of the search space. The process repeats itself until a stop condition is reached – depending on the algorithm, this might be after a certain number of generations, or when a certain fitness level is reached.

The two elements of a candidate solution are the genotype and the phenotype. The genotype is the set of parameters that define that particular solution. The genotype can be encoded in any fashion as long as the encodings are searchable - one of the simplest and most common encodings is a string of bits. This encoding is called a phenotype.

Genetic algorithms are useful in many fields, including economics, system design, cryptanalysis, video games, and logistics. A major advantage that genetic algorithms have over other machine learning techniques is that they do not require prior analysis of the problem domain, since they start with a random, non-optimal, set of candidate solutions, and use evolutionary concepts to find an optimal one. This makes them quite helpful for problems with limited domain knowledge. Because mutation ensures that their search covers different parts of solution space, they are also good at problems where the error surface has local minima.

1.2 Artificial Neural Networks

Artificial neural networks are computational models of interconnection between groups of nodes, or neurons, which are inspired by the physical interconnection between neurons in the human brain (Haykin, 1994). They can be represented as graphs, and can approximate continuous functions and search for patterns.

These networks can be thought of in terms of layers. They contain a layer of inputs, a layer of outputs, and hidden neurons in between the inputs and the outputs. These hidden neurons, however, do not have to be rigidly layered. NEAT and HyperNEAT, the algorithms which we will be discussing in this paper, evolve their own network topologies, and so the hidden neurons of their neural networks need not be strict layers.

To perform their computations, neural networks start by taking input values from the input layer. Each connection, or edge in the graph, has a weight assigned to it. Neurons within the network take weighted sums of their inputs, and pass these through a predefined activation function.

1.3 Neuroevolution

Neuroevolution is an evolutionary learning method that combines genetic algorithms and neural networks, in which neural networks are the phenotype of the genetic algorithm, and neural network parameters such as topology or connection weights are the genotype (Stanley, 2004). The algorithm is searching for the neural network that is optimal for some particular problem. Essentially, to use the biological metaphors, the algorithm is evolving the best brain for the problem at hand.

A neuroevolution algorithm can evolve network topology, or connection weights, or both. If it evolves both, it can evolve them separately from each other, or in parallel. They can also use direct encoding, in which every connection and node of a phenotype is specified in its genotype, or indirect encoding, in which the genotypes merely provide rules for constructing phenotypes. NEAT and HyperNEAT evolve both network topology and connection weights. NEAT uses direct encoding, and HyperNEAT uses an indirect encoding scheme which we discuss below.

Neuroevolutionary algorithms are particularly useful for problem classes in which the right neural network would be useful for solving the problem, but it is difficult or impossible to define the correctly-labeled input data needed for supervised learning. They are usually classified as a type of reinforcement learning. They have been successfully used in such applications as music composition (Chen, 2001), playing Go (Richards et al, 1998), controlling a robot arm (D'Silva, 2009), and the control of agents in mobile ad-hoc networks (Knoester, 2010).

The next two sections describe the two neuroevolution algorithms whose performance we wish to compare, NEAT and HyperNEAT.

1.4 NEAT

NEAT (NeuroEvolution of Augmenting Topologies) is a neuroevolution algorithm developed at the University of Texas at Austin in 2002 (Stanley & Mikkulainen, 2002). It encodes network structure and connection weights using nodes, and uses the biological principles of increasing complexity and speciation, which we explain below.

The structure of the NEAT genotype contains a list of neuron genes and a list of link genes. Neuron genes assign a neuron an identification number and indicate whether it is an input, output, or hidden node. Link genes contain information about the two neurons to which they are connected, the connection wait of that link, and flags to indicate whether the link is recurrent and whether it is enabled (if another neuron is inserted in the midst of it, it is not enabled). They also have an innovation number to indicate the order in which they were added. Any time a link is added, the algorithm checks to see whether that link has existed before (whether it was there but had been removed, as opposed to being completely novel). If it has, it is assigned the previously-existing innovation number, otherwise it is assigned the next available one.

Below, we depict a neural network phenotype and its NEAT encoding:

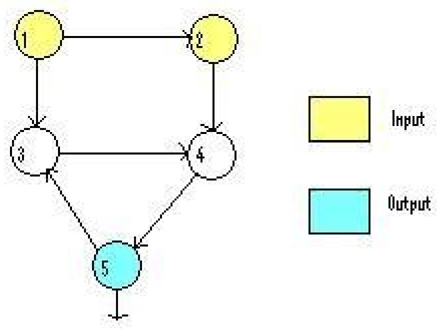

Illustration 1: A simple neural network

ID: 1	ID: 2	ID: 3	ID: 4	ID: 5
Type: Input	Type: Input	Type: Hidden	Type: Hidden	Type: Output

Table 1: Neuron genes for Illustration 1

Weight: 0.6	Weight: -2.1	Weight:1.8	Weight: 1.3	Weight: -0.8	Weight: 2.2
From: 1	From: 1	From: 3	From: 3	From: 4	From: 5
To:3	To: 4	To: 4	To: 5	To: 5	To: 3
Enabled: Y	Enabled: Y	Enabled: Y	Enabled: N	Enabled: Y	Enabled: Y
Recurrent: N	Recurrent: N	Recurrent: N	Recurrent: N	Recurrent: N	Recurrent:Y
Innovation: 1	Innovation: 2	Innovation: 3	Innovation: 4	Innovation: 5	Innovation: 6

Table 2: Link genes for Illustration 1

NEAT allows both mutations to network structure and mutations to connection weights. Connection weights are mutated in the standard way for neuroevolution algorithms, in which they are perturbed with a certain probability by choosing a floating point number from a uniform distribution of numbers, and adding it to the weight. Unlike with other neuroevolution algorithms, however, there is a predefined probability that a connection weight will be completely replaced, and not just perturbed (Buckland, 2002). There are two forms of structural mutations, both of which add genes and thus expand the genotype. The algorithm can add a connection to the network, connecting two nodes that were previously unconnected, or it can add a node, by disabling an old connection, splitting it, and putting a new node in between the two old nodes so that the first old node is now connected to the new one, and the new one is now connected to the second old node.

NEAT follows the philosophy that search should begin in the smallest space possible and gradually expand through the search space. The first generation of networks in an instance of NEAT has the same structure of nodes and links, and differ only in their connection weights. They have no hidden neurons, only an input layer and an output layer. Every input node in these first-generation networks is connected to every output node.

NEAT uses a bio-inspired method, speciation, to protect new innovations in the early stages of their evolution. In the biological world, organisms evolve into different species, and these species fill different niches, or roles in the environment. In a neuroevolution algorithm, when a new connection or neuron is added to a network, it is likely that the new individual will do poorly, and without measures to protect it, it could die out before evolving any potentially useful behaviors. NEAT prevents the extinction of these premature innovations by simulating biological speciation. At the start of the algorithm's run, there are no pre-existing species, so NEAT creates the first species and puts the first genotypes into it. As new innovations happen, new species are created, and the system stores them in a list. Individuals are tested against the first member of each species to calculate a compatibility distance. If the compatibility distance is small enough, according to defined boundaries, the individual is added to that species, and shares in the fitness of their species' niche (which also prevents any one species from taking over the population). If the individual is not compatible with any existing species, it becomes the first member of a new species, which is added to the list.

The innovation numbers, depicted in Illustration 2 above, provide historical markings that allow the system to track the origins of different genes, and remember which genes are compatible with which during recombination. This allows NEAT to perform recombination more efficiently, because it does not have to analyze the topologies of the neural networks in orderer to recombine them.

1.5 HyperNEAT

HyperNEAT (Hypercube-based NeuroEvolution of Augmenting Topologies) is an extension of NEAT that uses a form of indirect encoding called Compositional Pattern-Producing Networks (CPPNs). It was created as an attempt to narrow the gap between the results produced by neuroevolution algorithms and the scale of natural brains (Stanley et al, 2009).

In a CPPN, the inputs are Cartesian coordinates in *n*-dimensional space, where *n* is the number of inputs, and are passed into a function *f*. The value of *f* evaluated at each coordinate is a provides the presence, absence, or intensity of expression of a point in n-dimensional space, and so the phenotype drawn by the evaluation of *f* (which can be seen as the genotype), is a pattern (Stanley, 2007). A difference between ordinary artificial neural networks (ANNs) and CPPNs is that ANNs usually only contain sigmoid functions, while CPPNs can contain many different types of functions (Stanley et al, 2009).

While CPPNs can be evolved through genetic algorithms just as ANNs can, in HyperNEAT, CPPNs are not being evolved as an end product but are being used to encode ANNs. Since a CPPN produces a spatial pattern, and an ANN is a connectivity pattern, the algorithm requires a mapping between spatial and connectivity patterns. The input into the CPPN is two points rather than one, and the two points represent two nodes in the ANN, with the output of the CPPN's function being the weight of the link between the two nodes. Thus, the pattern being produced by the CPPN is a neural network topology. This topology-producing CPPN is called a connective CPPN, as opposed to spatial CPPNs, which produce spatial patterns. The connective CPPN takes a grid of nodes, called the substrate, and queries every potential connection to determine whether there is actually a connection there and what its weight is, by taking the positions of the two nodes and outputting the weight of the connection between them. This produces a pattern of connections between nodes in space that is derived from the substrate's configuration.

HyperNEAT evolves CPPNs using the NEAT strategy. The CPPNs are representations of patterns in hyperspace, with every point bounded by a four-dimensional hypercube. So the pattern in a four-dimensional hypercube can be mapped to a two-dimensional neural network. The first step of HyperNEAT is to choose the substrate layout and assignments between inputs and outputs. The algorithm initializes a first generation of minimal CPPNs with random weights, just as NEAT initializes a first generation of minimal ANNs with random weights. Then, it queries each CPPN as described in the previous paragraph, and comes up with an ANN-like connectivity pattern. The algorithm determines this ANN's fitness just as NEAT does with its ANNs, and evolves the second generation of CPPNs according to the NEAT strategy. This repeats as any neuroevolution algorithm would until a solution is found.

We selected HyperNEAT for this study because it is particularly well-suited for problems where geometric knowledge is important (Stanley et al, 2009). Using the substrate, it “sees” the coordinates of the neurons. It can “see” which pixel is next to which in an image, or which square is next to which on a chessboard, rather than having to learn them through examples as a traditional neural network would do.

1.6 “Fractured” Problems

In this paper, we are examining the performance of NEAT and HyperNEAT on a class of problems which are referred to in the literature as “fractured” problems (Kohl & Mikkulainen, 2009), by testing them on the example benchmark problem of keepaway soccer. In a fractured problem, the correct decision for the problem-solving agent changes abruptly and often, rather than slowly and continuously, as the agent moves between states.

This is an important class of problem because there are many fractured problems in the real world, particularly ones that involve high-level strategic thinking. For example, auto-racing strategy (Kohl & Mikkulainen, 2009), division of labor of robots cleaning up a hazardous waste site, and autonomous vehicle navigation in crowded terrain (Kohl, 2009) are fractured problems.

Fractured problems such as keepaway soccer have decision spaces where decision boundaries are abrupt and decision space can often be mapped onto real-world space, and we felt that an algorithm such as HyperNEAT that can exploit geometry as described in the previous section, and “see” areas that are adjacent to other areas in a model, would be well-suited for such a problem.

2. Methodology

2.1 SharpNEAT

The implementations of NEAT and HyperNEAT that we studied are part of a package called SharpNEAT (Green, 2004). This package contains C# implementations of NEAT and HyperNEAT using the Microsoft .NET framework. It also contains GUIs that allow the user to easily start and track experiments. We chose these implementations because the keepaway soccer program (see below) that we had was also written in C# using .NET. They are new implementations of the algorithms, written from scratch, rather than simply ported code from the original C++, but they were written using the original NEAT and HyperNEAT source as a reference point, and with all the same parts of the original algorithms.

2.2 Keepaway Soccer

Keepaway soccer (Whiteson et al, 2005; Stone et al, 2006) is a machine learning benchmark problem that requires high-level strategic decision-making and has a fractured decision space as discussed in section 1.6 (Kohl & Mikkulainen, 2009). The "keepers" are charged with preventing the "takers" from taking the ball, for as long as possible. In the implementation of the game that we used (Verbancsics & Stanley, 2010), the keepers and the takers move at the same speed, and so the strategy of the keepers mixes passing and running with the ball.

The two teams of players are controlled by different algorithms. The keepers are controlled by both fixed code and a learning algorithm of the user's choice. When the game starts, the keeper nearest the ball attempts to get control of the ball (defined as being within a certain distance of the ball), while the other keepers try to get into good positions to receive a pass. Once the first keeper has the ball, it has to choose between holding the ball, which is a pre-coded maneuver, or passing the ball to a teammate of its choice. This is where the learning algorithm comes in. Through the training of the algorithm, the keepers learn which decision to make – whether to pass, and to whom to pass – given the game state at that time. If the controlling keeper passes the ball, and its teammate gets control of the ball, then the teammate starts going through the same decision process, while the original controlling keeper tries to get into a good position to receive a pass.

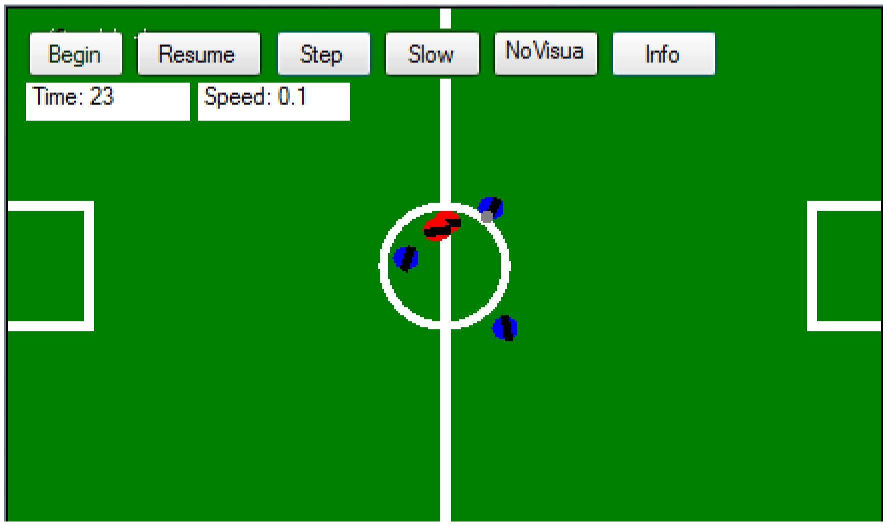

Illustration 2: A screenshot of keepaway soccer

The field is split into a 20 x 20 grid of inputs, and the inputs to each player are +1 for a teammate and -1 for an opponent. The neural network produces a 20 x 20 array of outputs, which assign a value to each of the 400 grid spaces. If the keeper in control of the ball finds a teammate whose current grid space has a value higher than that of its own, it will pass to that teammate. Otherwise, it will pass to itself, maintaining control of the ball.

2.3 Performance Metrics

In order to assess the performance of NEAT and HyperNEAT on keepaway soccer, we looked at the number of timesteps for which the keepers were able to control the ball (the keepers' "score"). We trained the algorithms using different configurations (numbers of keepers and takers), and compared the scores achieved by their best-performing candidate solutions.

3. Results and Discussion

3.1 Using One Configuration

Our first experiment was to compare the scores of NEAT and HyperNEAT using a 2 vs 3 configuration (2 takers and 3 keepers). NEAT achieved a score of 41 and HyperNEAT achieved a score of 48.

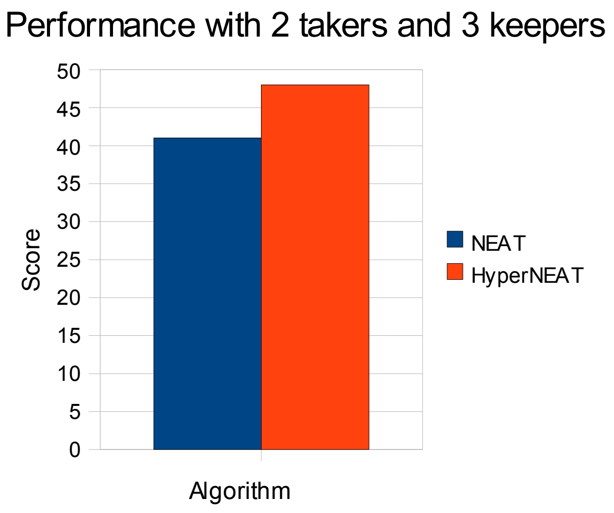

Illustration 3: Comparison of NEAT and HyperNEAT with the configuration of 2 takers and 3 keepers

Illustration 3 shows that with the configuration of 2 takers vs 3 keepers, HyperNEAT outperforms NEAT by 17%, which lends support to our hypothesis that HyperNEAT's ability to exploit geometric information improves its performance on this problem.

Illustrations 4 and 5 show us the scores achieved by individual genotypes ("genotype" and "genome" are used interchangeably in genetic algorithm literature) produced by NEAT and HyperNEAT. In both cases, the worst candidates achieved scores of about 35, and only one to two candidates scored above 40. This suggests that the algorithms are performing similarly for the most part, but HyperNEAT's top candidates are able to break away from the pack to a greater degree than NEAT's.

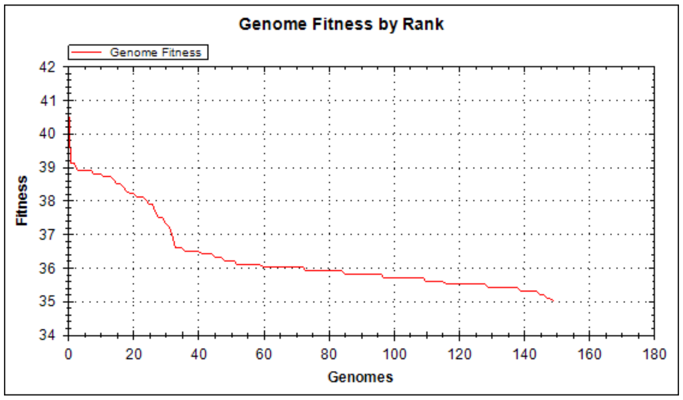

Illustration 4: Scores achieved by NEAT's candidate solutions ("rank" refers to rank among the candidates)

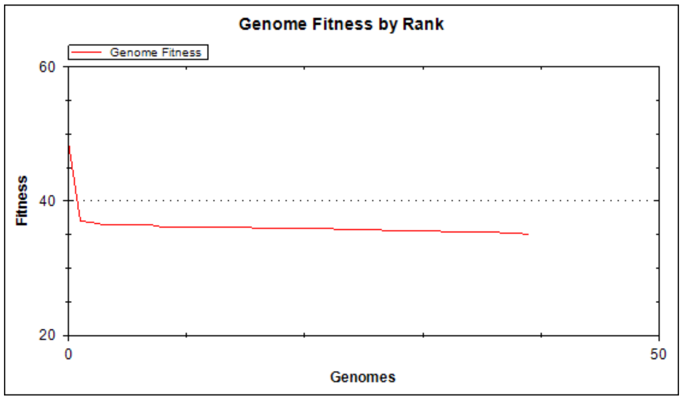

Illustration 5: Scores achieved by HyperNEAT's candidate solutions ("rank" refers to rank among the candidates)

One way to increase the level of fracture in a fractured problem is to change the number of starting states – in this case, configurations of players and keepers – for the problem, which changes the amount of variation in the problem (Kohl & Mikkulainen, 2009). Our next experiment used this principle. We ran NEAT and HyperNEAT using a set of 16 different game configurations – all possible combinations of 2-5 takers and 3-6 keepers. The score achieved by each algorithm was the average of the scores for the 16 configurations. In this experiment, NEAT achieved a score of 58 and HyperNEAT achieved a score of 52 – NEAT performed 11.5% better than HyperNEAT.

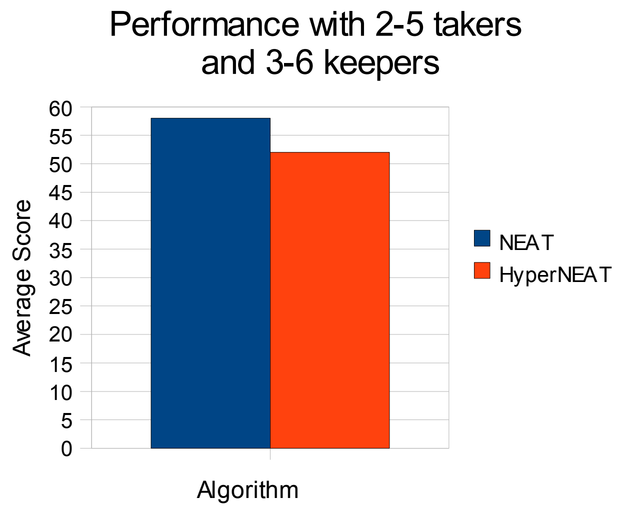

Illustration 6: Comparison of NEAT and HyperNEAT with the configurations of 2-5 takers and 3-6 keepers

Illustration 6 shows that with this set of configurations, NEAT outperforms HyperNEAT on average. This suggests that, while HyperNEAT was an improvement over NEAT for a moderately fractured problem, the benefit disappeared as the level of fracture increased.

A peculiar aspect of this finding is that neuroevolutionary algorithms, including NEAT, normally perform less well on a problem with more fracture (Kohl & Mikkulainen, 2009), but in this case, both NEAT and HyperNEAT did better with more starting states. It is possible that with only one starting state to train on, the keepers were not trained as well, and so performed less well even though the problem should have been easier to solve.

We observed, while running the algorithms, that HyperNEAT can be very slow on the keepaway soccer problem, and that even on a multi-core processor it took hours to produce a single generation of candidate solutions. This could make HyperNEAT's use on certain problems infeasible even if its learning was ultimately very good.

4. Conclusions and Future Work

Based on our experiments, we concluded that HyperNEAT improves upon NEAT for a relatively simple fractured problem, but that the benefit disappeared as the problem was complicated.

An obvious avenue for future work would be to test both NEAT and HyperNEAT on other fractured problems and see if the results are similar. It is possible that there was a factor specific to the keepaway soccer problem that affected the results, that would not be the case in other fractured problems.

Some work has been done extending NEAT to improve its performance on fractured problems (Kohl & Mikkulainen, 2009; Kohl, 2009). Kohl and Mikkulainen have demonstrated that NEAT's performance on several fractured problems can be improved if a mutation is added to the algorithm that allows the addition of radial basis function nodes (RBF-NEAT). They also demonstrated that NEAT's performance on fractured problems could be improved if its search space were constrained to cascade architectures, a network architecture in which each hidden node is connected to all hidden nodes to its left in addition to inputs and outputs (Cascade-NEAT). Kohl combined these two improvements into an algorithm called SNAP-NEAT, which was able to perform well on both fractured and non-fractured problems.

Following from these results, a possible avenue of future work might be to add these improvements to HyperNEAT – to create an RBF-HyperNEAT, Cascade-HyperNEAT, and/or SNAP-HyperNEAT, and test whether they improve HyperNEAT's performance on fractured problems as they improved NEAT's.

Finally, one major problem with HyperNEAT is that it is very slow, even on a multi-core processor. A possible solution to this could be an implementation of HyperNEAT that takes advantage of other high-performance computing techniques (such as a GPU implementation) that would speed up the runtime. Along other lines, it has been demonstrated that machine learning techniques such as random decision forests can be made to determine for themselves when they have had enough training through the use of statistical hypothesis tests, allowing them to learn more quickly using sparse data (McBride et al, 2009). It is possible that this could be adapted to use with HyperNEAT in order to increase its speed.

Bibliography

Buckland, M., 2002. AI Techniques for Game Programming. Cincinnati, OH: Premier Press.

Chen, C.-C.J., Mikkulainen, R. Creating melodies with evolving recurrent neural networks. Proceedings of the International Joint Conference on Neural Networks (IJCNN '01). Vol. 3. pp 2241-2246.

De Jong, K.A., 1975. An analysis of the behavior of a class of genetic adaptive systems. Doctoral dissertation. University of Michigan, Department of Computer and Communications Science.

D'Silva, T., Mikkulainen, R. Learning dynamic obstacle avoidance for a robot arm using neuroevolution. Neural Processing Letters. Vol. 30:1. pp 59-69.

Goldberg, D.E., 1989. Genetic Algorithms in Search, Optimization, and Machine Learning. Reading, MA: Addison-Wesley.

Green, C. 2004. SharpNEAT software package. http://sharpneat.sourceforge.net

Haykin, S., 1994. Neural Networks: A Comprehensive Foundation. New York: Macmillan.

Knoester, D.B, Goldsby, H.J., McKinley, P.K. 2010. Neuroevolution of mobile ad-hoc networks. Proceedings of the 12th Genetic and Evolutionary Computation Conference (GECCO '10). pp 603-610.

Kohl, N. 2009. Learning in fractured problems with constructive neural network algorithms. Doctoral dissertation. University of Texas at Austin, Department of Computer Science.

Kohl, N., Mikkulainen, R., 2008. Evolving neural networks for fractured domains. Proceedings of the 10th Genetic and Evolutionary Computation Conference (GECCO '08). pp 1405-1412.

Kohl, N., Mikkulainen, R., 2009. Evolving neural networks for strategic decision-making problems. Neural Networks. Vol. 22:3. pp 326-327.

McBride, J., Lowell, J., Snorrason, M., Eaton, R., Irvine, J. Automated rapid training of ATR algorithms. Proceedings of SPIE. Vol. 7335.

Richards, N., Moriarty, D.E, Mikkulainen, R. Evolving neural networks to play Go. Applied Intelligence. Vol. 8:1. pp 85-96.

Stanley, K.O. 2004. Efficient evolution of neural networks through complexification. Doctoral dissertation. University of Texas at Austin, Department of Computer Science.

Stanley, K.O., 2007. Compositional pattern producing networks: A novel abstraction of development. Genetic Programming and Evolvable Machines: Special Issue on Developmental Systems. Vol. 8:2. pp 131-162.

Stanley, K.O., D'Ambrosio, D., Gauci, J., 2009. A Hypercube-based indirect encoding for evolving large-scale neural networks. Artificial Life. Vol. 15:2. pp 185-212.

Stanley, K.O., Mikkulainen, R., 2002. Evolving neural networks through augmenting topologies. Evolutionary Computation. Vol. 10:2. pp 99-127.

Stone, P., Kuhlmann, G., Taylor, M.E., Liu, Y., 2006. Keepaway soccer: From machine learning testbed to benchmark. Lecture Notes in Computer Science. Vol. 4020. pp 93-105.

Verbancsics, P., Stanley, K.O., 2010. Evolving static representations for task transfer. Journal of Machine Learning Research. Vol. 11. pp 1737-1769.

Whiteson, S., Kohl, N., Mikkulainen, R., Stone, P., 2005. Evolving soccer keepaway players through task decomposition. Machine Learning. Vol. 59:1-2. pp 5-30.