A Case Study on the Critical Role of Geometric Regularity in Machine Learning

Jason Gauci and **Kenneth O. Stanley**
Evolutionary Complexity Research Group
School of Electrical Engineering and Computer Science
University of Central Florida, Orlando, FL 32816
{jgauci,kstanley}@eecs.ucf.edu

Abstract

An important feature of many problem domains in machine learning is their geometry. For example, adjacency relationships, symmetries, and Cartesian coordinates are essential to any complete description of board games, visual recognition, or vehicle control. Yet many approaches to learning ignore such information in their representations, instead inputting flat parameter vectors with no indication of how those parameters are situated geometrically. This paper argues that such geometric information is critical to the ability of any machine learning approach to effectively generalize; even a small shift in the configuration of the task in space from what was experienced in training can go wholly unrecognized *unless* the algorithm is able to learn the *regularities* in decision-making across the problem geometry. To demonstrate the importance of learning from geometry, three variants of the same evolutionary learning algorithm (NeuroEvolution of Augmenting Topologies), whose representations vary in their capacity to encode geometry, are compared in checkers. The result is that the variant that can learn geometric regularities produces a significantly more general solution. The conclusion is that it is important to enable machine learning to detect and thereby learn from the geometry of its problems.

Introduction

Among the primary goals of any approach to machine learning is generalization. This paper argues that the ability to represent and thereby discover regularities in the *geometry* of the task domain is essential to generalization. For example, knowing the relative positions of squares in a board game is fundamental to mastering the mechanics of the game. Understanding the implications of adjacency requires recognizing the same adjacency relationships between any two squares on the board. A general understanding of board geometry makes it possible to learn general tactics rather than specific actions tied to only one position. This central role of geometric regularity to generalization extends beyond board games to robot control, in which events at different relative positions often require similar responses, and computer vision tasks, in which the same object may appear at different positions and orientations in the retina.

To appreciate how essential geometry is to learning, imagine learning to play checkers on a board whose squares are each torn from the board and scattered across the living room randomly. The rules are the same and each square still represents the same position on the board as usual. The only problem is that the adjacency relationships among the pieces become entirely opaque to the player. Interestingly, when a board state is input into a machine learning algorithm as a flat vector of position values, the geometry of the board is no less opaque to the learner than in this satirical scenario.

Recognizing that task geometry plays a critical role in many machine learning domains, researchers have introduced a variety of methods ranging from tile coding [16] to specialized neural network topologies [3, 17] that exploit geometric relationships in different ways. However, such approaches typically require the user to specify *a priori* how different regions of the task domain should be broken apart or assorted, which means the learner cannot itself discover the most essential regularities and relationships.

To demonstrate that *learning* geometry is critical to generalization, this paper compares three established evolutionary approaches in checkers. These approaches vary in how they integrate geometry. The baseline approach is NeuroEvolution of Augmenting Topologies (NEAT), which evolves increasingly complex artificial neural networks (ANN's) [12]. In the second approach, NEAT is extended with additional ANN inputs engineered to better capture the geometry of the problem domain. The third approach is Hypercube-based NEAT (HyperNEAT), an extension of NEAT designed to *discover* geometric regularities on its own [4, 6, 15]. By comparing NEAT to extensions of the same approach, it is possible to isolate the effects of geometry on generalization.

Results demonstrate that even among solutions learned by these approaches with *equivalent* training performance, HyperNEAT's solutions are significantly more general. The ability to map regular concepts across the geometry of the board provides HyperNEAT the advantage. An important conclusion is that the ability to detect and learn from geometry is likely important in many machine learning tasks.

Background

Machine learning algorithms often focus on optimizing the search without special attention to geometry. This section surveys several exceptions to this rule and then reviews the NEAT and HyperNEAT methods.

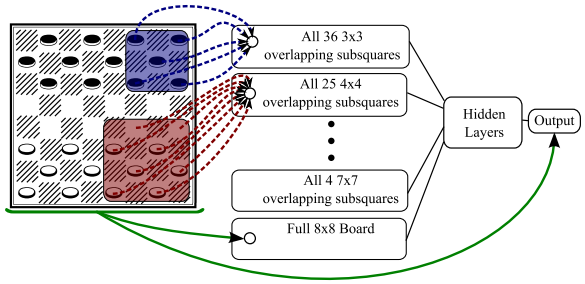


Figure 1: **Blondie24 ANN Topology [3] .** The first hidden layer contains a node for every subsquare of the board of size greater than 2×2. Positions on the board are linked to the corresponding subsquares that contain these positions. This layer then connects to hidden layers that finally connect to the output node. Each valid square on the board connects directly to the output node.

Geometric Regularity in Machine Learning

Tile coding is a common reinforcement learning technique that partitions the state space of a task into small (often overlapping) chunks. Because the state space is often geometric, e.g. in maze navigation [9], the partitions separate different geometric locations. By breaking the geometry into parts, each part can be learned separately as a simple subtask. While advantageous in several problem domains, a downside is that because tile coding breaks the geometry into pieces, it prevents the learner from discovering patterns and regularities that vary across whole dimensions of the geometry. Leffler et al. [7] show how this problem can be alleviated by *a priori* specifying to the learning method which tiles are related, thereby conveying useful regularities. However, they note that an ideal approach would exploit geometric regularities autonomously.

An interesting attempt to integrate geometry into evolutionary computation is Blondie24, an evolved checkers-playing artificial neural network (ANN) [3]. The main idea in Blondie24 is that the ANN topology can be better *engineered* to respect the regularities inherent in the game. In particular, the weights of an ANN topology engineered by hand are evolved. Every subsquare (i.e. set of positions arranged in a square shape) of the board is input to a separate hidden node responsible for only that subsquare (figure 1). Connections are specified from the actual board inputs to their respective subsquares, and also between the inputs and the final output node. The main idea in this engineered structure is that independent local relationships within each subsquare can be learned separately and then combined at a higher level in the network. Through coevolution (i.e. candidates were evaluated by playing against each other), Blondie24 was able to reach expert-level play on a popular internet checkers server [3]. However, as with reinforcement learning, an ideal approach would remove the need for engineering by learning geometric regularities on its own.

NeuroEvolution of Augmenting Topologies (NEAT)

The approaches compared in this paper are variants of the NEAT method [12, 14], which, like the approach in Blondie24 [3], evolves ANNs. In addition to evolving weights of connections, NEAT can build structure and add complexity. NEAT is a leading neuroevolution approach that has shown promise in board games and other challenging control and decision making tasks [13, 14].

NEAT is based on three key ideas. First, to allow network structures to increase in complexity over generations, a method is needed to keep track of which gene is which. Otherwise, it is not clear in later generations which individual is compatible with which, or how their genes should be combined to produce offspring. NEAT solves this problem by assigning a unique *historical marking* to every new piece of network structure. The historical marking is a number assigned to each gene based on its order of appearance over the course of evolution. The numbers are inherited during crossover unchanged, and allow NEAT to perform crossover without the need for topological analysis. That way, networks of different organizations and sizes stay compatible throughout evolution.

Second, NEAT divides the population into species, so that individuals compete primarily within their own niches instead of with the population at large. This way, topological innovations are protected and have time to optimize their structure before competing with other niches in the population. NEAT uses the historical markings on genes to determine to which species different individuals belong.

Third, NEAT begins with a uniform population of simple networks with no hidden nodes, differing only in their initial weights. Speciation protects new innovations, allowing diverse topologies to gradually increase in complexity over evolution. Thus, NEAT can start minimally, and grow the necessary structure over generations. Through increasing complexity, high-level features can be established early in evolution and then elaborated and refined as new genes are added [1]. The next section reviews an extension of NEAT that allows it to learn geometric relationships automatically.

CPPNs and HyperNEAT

Like many methods in machine learning, the reason that regular NEAT cannot explicitly learn geometric regularities is that when it learns to represent important local relationships (e.g. how a checkers piece in one square can be threatened by another in an adjacent square), it cannot extend that relationship as a pattern of connectivity across the entire neural structure connecting to the board. In other words, it needs to rediscover similar concepts multiple times.

The main idea in HyperNEAT is that it is possible to learn such relationships if the solution is represented *indirectly*, which means that it is a *generative description* of the connectivity of the ANN rather than embodying the connection weights of ANN itself. Unlike a *direct* representation, wherein every dimension in the solution space (i.e. the phenotype in evolutionary computation) is described individually (i.e. by its own gene), an indirect representation can describe a pattern of values in the solution space without explicitly enumerating every such value. That is, information is reused in such an *indirect encoding*, which is a major focus in the field of *generative and developmental systems*, the subfield of evolutionary computation from which HyperNEAT originates [2, 8, 11, 18]

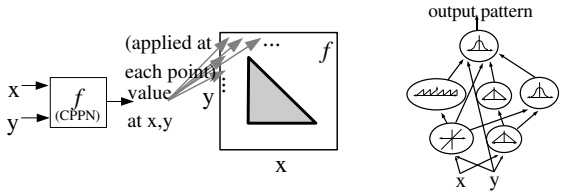


Figure 2: **CPPN Encoding.** (a) A CPPN is a function f that takes arguments x and y, which are coordinates in a two-dimensional space. When all the coordinates are drawn with an intensity corresponding to the output of f, the result is a spatial pattern. (b) Internally, the function produced by the CPPN is encoded as a graph that specifies how a set of simpler canonical functions (e.g. Gaussian, sigmoid, and sine) are connected. Like in an ANN, the connections are weighted such that the output of a function is multiplied by the weight of its outgoing connection.

HyperNEAT is based on an indirect encoding called Compositional Pattern Producing Networks (CPPNs) [10]. The idea behind CPPNs is that patterns such as those seen in nature can be described at a high level as a *composition of functions* that are chosen to represent several common motifs in patterns. For example, because the Gaussian function is symmetric, when it is composed with any other function, the result is a symmetric pattern. The appeal of this encoding is that it allows patterns with regularities such as symmetry (e.g. with Gaussians), repetition (e.g. with periodic functions such as sine), and repetition with variation (e.g. by summing periodic and aperiodic functions) to be represented as networks of simple functions, which means that NEAT can evolve CPPNs just as it evolves ANNs. While CPPNs are similar to ANNs, the distinction in terminology is particularly important for explicative purposes because in HyperNEAT, CPPNs describe ANNs. Formally, CPPNs produce a phenotype that is a function of n dimensions, where n is the number of dimensions in a geometric space. For each coordinate in that space, its level of expression is an output of the function that encodes the phenotype. Figure 2 shows how a two-dimensional pattern can be generated by a CPPN that takes two inputs.

The main idea in HyperNEAT is to extend CPPNs, which encode two-dimensional spatial patterns, to also represent *connectivity patterns* [4, 6, 15]. That way, NEAT can evolve CPPNs that represent ANNs with symmetries and regularities that are computed directly from the geometry of the task inputs. The key insight is that $2n$-dimensional spatial patterns are *isomorphic* to connectivity patterns in n dimensions, i.e. in which the coordinate of each endpoint is specified by n parameters.

Consider a CPPN that takes *four* inputs labeled x_1, y_1, x_2, and y_2; this point in four-dimensional space can *also* denote the connection between the two-dimensional points (x_1, y_1) and (x_2, y_2), and the output of the CPPN for that input thereby represents the weight of that connection (figure 3). By querying every possible connection among a set of points in this manner, a CPPN can produce an ANN, wherein each queried point is the position of a neuron. Because the connection weights are produced as a function of their endpoints, the final structure is produced with *knowledge* of its

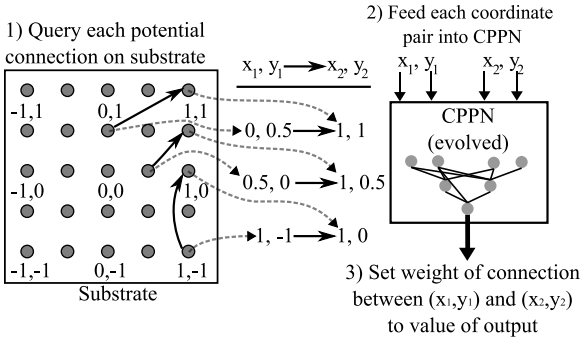


Figure 3: **Hypercube-based Geometric Connectivity Pattern Interpretation.** A grid of nodes, called the *substrate*, is assigned coordinates such that the center node is at the origin. (1) Every potential connection is queried to determine its presence and weight; the dark directed lines in the substrate in the figure represent a sample of connections that are queried. (2) For each query, the CPPN takes as input the positions of the two endpoints and (3) outputs the weight of the connection between them. Thus, ***CPPNs*** can produce regular patterns of connections in space.

geometry. In effect, the CPPN paints a pattern on the inside of a four-dimensional hypercube that is interpreted as an isomorphic connectivity pattern, which explains the origin of the name *Hypercube-based NEAT* (HyperNEAT). Connectivity patterns produced by a CPPN in this way are called *substrates* so that they can be verbally distinguished from the CPPN itself, which has its own internal topology.

Recall that each queried point in the substrate is a node in an ANN. The experimenter defines both the location and role (i.e. hidden, input, or output) of each such node. As a rule of thumb, nodes are placed on the substrate to reflect the geometry of the task, which makes the setup straightforward [4, 6, 15]. This way, the connectivity of the substrate becomes a direct function of the task structure.

For example, in a board game, the inputs can be placed on the substrate in a two-dimensional plane just as their corresponding squares are arranged on the board. In this way, knowledge about the problem can be injected into the search and HyperNEAT can exploit the regularities (e.g. adjacency, or symmetry) of a problem that are invisible to traditional encodings. For full descriptions of HyperNEAT see [4, 6, 15]. The next section explains in detail how checkers is represented and learned by HyperNEAT.

Approach: Learning Regularities in Checkers

This paper focuses on the need to learn from geometry. While approaches like Blondie24 engineer geometry into the ANN topology to take advantage of it, the idea in HyperNEAT is to *learn* from geometry by generating the policy network as a direct function of task geometry. This section explains how that is done in the game of checkers.

To apply HyperNEAT to checkers, the substrate input layer is arranged in two dimensions to match the geometry of the checkers board (figure 4). To distinguish the flow of information through the policy network from the geometry of the game, a *third* dimension in the substrate represents

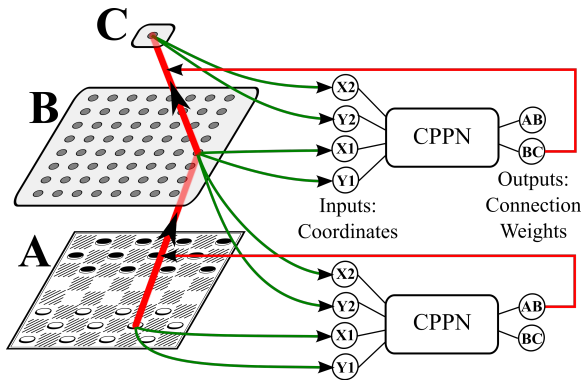


Figure 4: **Checkers Substrate.** The substrate (at left) contains a two-dimensional input layer (A) that corresponds to the geometry of a game board, an analogous two-dimensional hidden layer (B), and a single-node output layer (C) that returns a board evaluation. The two CPPNs (at right) are depictions of the *same* CPPN being queried to determine the weights of two different substrate connections. The bottom CPPN depiction receives as input the x and y coordinates of a node in A and a node in B and returns the weight of this connection from its AB output node. Similarly, the top depiction of the same CPPN is being queried for the weight of a connection between B and C and therefore returns this weight from its BC output. In this way, a four-input CPPN can specify the connection weights of a two-layer network structure as a function of the positions, and hence the geometry, of each node.

information flow from one layer to the next. Along this third dimension, the two-dimensional input layer connects to an analogous two-dimensional hidden layer so that the hidden layer can learn to process localized geometric configurations. The hidden layer then connects to a single output node, whose role is to evaluate board positions. The CPPN distinguishes the set of connections between the inputs and the hidden layer from those between the hidden layer and the output node by querying the weights of each set of connections from a *separate* output on the CPPN (note the two outputs in the CPPN depiction in figure 4). That way, the x and y positions of each node are sufficient to identify the queried connection and the outputs differentiate one connection layer from the next. Because the CPPN can effectively compute connection weights as a function of the *difference* in positions of two nodes, it can easily map a repeating concept across the whole board.

In this way, the substrate is a board evaluation function. The function inputs a board position and outputs its value for black. To evaluate the board when it is white's turn to move, the color of the pieces can be reversed and then the sign of the result inverted. To decide which move to make, a minimax search algorithm runs to a fixed ply depth. Alpha-beta pruning and iterative deepening techniques increase performance without changing the output. The output of the substrate is the heuristic score for the minimax algorithm.

This approach allows HyperNEAT to discover geometric regularities on the board by expressing connection weights as a function of geometry. It is therefore unnecessary to manually engineer the network topology, or divide the input space into subsections in an attempt to inject a priori theories about the key regularities in the game into the representation. Because HyperNEAT discovers geometric relationships on its own, an *identical substrate* can be applied to other board games even without knowledge of the game rules, making the approach highly general.

Experiment

The experiment aims to determine whether encoding geometry helps machine learning to generalize. The idea is to learn to defeat a single fixed training opponent and then test for generalization against variations of this opponent.

Board games are an effective platform to discern the importance of geometry because they depend heavily on geometric relationships that often repeat across the board. Therefore, this paper compares three evolutionary approaches that take geometry into account to varying degrees in the domain of checkers. Each approach is trained against the same hand-engineered deterministic opponent [5]. The opponent is a linear combination of several heuristics, including material possession, positional bias, whether pieces on the back row have been moved (which would lower the score), whether a double corner is intact, and who controls the center and the edge of a board. Thus, the deterministic opponent is nontrivial, i.e. not just a simple piece counter. During evolution, each candidate plays a single game as black against the opponent to determine its fitness. Fitness is computed as a function of both the final game state and intermediate board states. At each turn, fitness is awarded based on the current board state according to the equation:

$$100 + 2m_s + 3k_s + 2(12 - m_o) + 3(12 - k_o), \quad (1)$$

where m_s and m_o are the number of regular pieces possessed by the learner and the opponent, respectively, and k_s and k_o are the number of kings. This function rewards incremental progress and provides a smoother learning gradient than simply awarding fitness based on the final score. Fitness is always awarded over 100 turns, even if the game ends earlier. That way, winning early is not penalized. If the candidate wins against the training opponent, an additional 30,000 is added to the total fitness.

The learned strategies are then tested against a non-deterministic variant of the same opponent. This variant has a 10% chance of choosing the second-highest scoring move instead of the optimal move found in minimax search. Methods that evolve more general solutions should produce policies that win more such games.

The three compared approaches are carefully chosen to isolate the issue of geometric processing. Therefore, they are all variants of the same NeuroEvolution of Augmenting Topologies (NEAT) approach. This shared basis means that differences in performance are attributable to the way each approach processes its inputs. For all three approaches, input values of 0.5 and -0.5 encode black and white pieces, respectively. Kings are represented by a magnitude of 0.75 (Chellapilla and Fogel [3] showed that multiplying the standard piece input magnitude by 1.3 produces a good magnitude for kings). A single output expresses the value of the current board state for black.

Regular NEAT inputs a vector of length 32 in which each parameter represents a square on the board that can poten-

tially hold a piece. NEAT evolves the topology and weights between the input and output nodes.

NEAT-EI is an attempt to enhance NEAT's ability to take into account geometric regularities across the board by supplying additional engineered inputs (EI). It has the same inputs as NEAT; however, the starting network topology is engineered as in Blondie24 [3] (figure 1). The result of training NEAT-EI in this paper cannot be compared directly to Blondie24 because Blondie24 is the result of *coevolution* while the policies in this paper are evolved against a fixed opponent. While the goal for Blondie24 was to evolve the best possible player, the goal in this paper is to fairly compare the generalization of different representations, thereby isolating the issue of generalization.

HyperNEAT inputs are arranged in a two-dimensional 8×8 grid that forms the first layer of a three-layer substrate (figure 4). For HyperNEAT, NEAT evolves the CPPN that computes the connection weights of the substrate.

If geometry is indeed essential to effective play, the better an approach can represent geometric relationships (either through learning or a priori engineering), the better that method should generalize.

Experimental Parameters

Because both HyperNEAT and NEAT-EI extend NEAT, they all use the same parameters [12]. The population size was 120 and each run lasted 200 generations. The compatibility threshold was 6.0 and the compatibility modifier was 0.3. Available CPPN activation functions were sigmoid, Gaussian, sine, and linear functions. Recurrent connections within the CPPN were not enabled. Signed activation was used, resulting in a node output range of $[-1, 1]$. By convention, a connection is not expressed if the magnitude of its weight is below a minimal threshold of 0.2 [6]. These parameters were found to be robust to moderate variation in preliminary experimentation.

Results

Performance in this section is measured in two ways. First, the fitness of each approach is tracked during training over generations, which gives a sense of relative *training* performance. Second, after training is complete, the best solutions from each run play 100 games against the randomized opponent, yielding generalization. The main question is whether HyperNEAT's ability to learn from geometry benefits its performance and generalization.

Training Performance

Figure 5 shows the average generation champion fitness over evolution averaged over 20 runs. While *none* of the runs of regular NEAT were able to defeat the opponent within 200 generations, both HyperNEAT and NEAT-EI learned to defeat it in all runs. On average, it took NEAT-EI 57.85 generations to find a winning solution. HyperNEAT succeeds much more quickly, finding a winner in 8.2 generations. These differences are statistically significant ($p < 0.001$). This disparity highlights the critical importance of learning from geometry. While defeating the heuristic appears challenging with direct representations, it becomes easy if the solution is learned as a function of the board geometry.

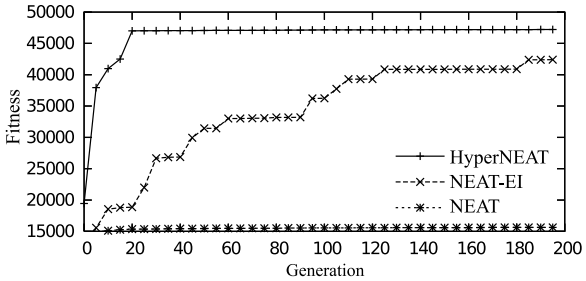


Figure 5: **Fitness During Training.** The fitness of the generation champions of each approach is shown, averaged over 20 runs. HyperNEAT generation champions perform significantly better than NEAT-EI and regular NEAT between generations 1 and 108 ($p < .01$). HyperNEAT learns faster than NEAT-EI because its CPPN solutions require fewer dimensions to represent.

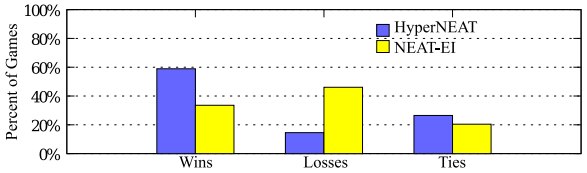


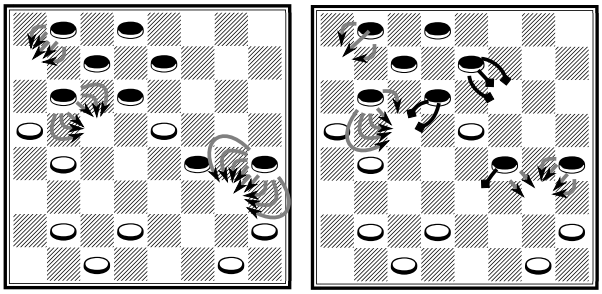
Figure 6: **Generalization Results.** Average wins, losses, and ties in 100 games against the randomized opponent are shown for HyperNEAT and NEAT-EI, averaged over 20 runs of each. Only the most general solutions of each run are included in the test. HyperNEAT solutions win significantly more games ($p < 0.001$) and lose significantly fewer games ($p < 0.001$) than NEAT-EI. The difference in ties between the two methods is not significant.

Generalization

Every generation champion that defeats the deterministic opponent plays 100 games against the randomized opponent. Because regular NEAT could never defeat this opponent, it cannot compete in this test. To make the comparison fair, only the *most general* solutions of each run are compared. That way, the generalization results focus on the *best possible* generalization for both methods when they learn to defeat an identical opponent. The best possible generalization represents what would result from an ideal validation of the trained opponents. While in the real world such idealized validation may not always be possible, assuming reasonable effort on the part of the experimenter, it is a yardstick for how well a system can be expected to perform in a reinforcement learning task. Figure 6 shows the results of these solutions against the randomized opponent. HyperNEAT wins significantly more and loses significantly less than NEAT-EI. The geometric encoding allows HyperNEAT to generalize across the board.

Typical Solutions

HyperNEAT's advantage is most evident in the middle-game and later. As the game tree branches, deviation from the training opponent increases. Because HyperNEAT performs better in such novel situations, it is more general. For example, figure 7 contrasts moves chosen by NEAT-EI solutions with those from HyperNEAT from the *same* unfamiliar position. NEAT-EI players unnecessarily sacrifice pieces, while HyperNEAT players rarely do from this position.



(a) HyperNEAT Moves (b) NEAT-EI Moves

Figure 7: **Requested moves from the same board position by HyperNEAT and NEAT-EI.** This figure depicts a position several moves into a game. Twenty moves requested by the champions of all NEAT-EI runs are contrasted with twenty from HyperNEAT runs. All of the HyperNEAT runs suggest neutral or positive moves. Six of the NEAT-EI runs make moves that lead to immediate, uncompensated loss. These moves are denoted with a darker line and a square endpoint.

The most general solution in all runs of NEAT-EI has 126 nodes and 1,106 connections. In contrast, the most general solution of HyperNEAT is a CPPN with only 23 nodes and 84 connections, which generates an ANN with 129 nodes and 3,979 connections. In this way, HyperNEAT is able to explore a significantly smaller search space (i.e. CPPNs) while still creating complex structures (i.e. substrates).

Discussion and Future Work

An important lesson is that a representation hand-engineered to respect the task geometry (e.g. NEAT-EI), while better than nothing (e.g. regular NEAT), may not be the best possible representation. It follows naturally that machine learning should, ideally, learn the most critical geometric regularities on its own, just as it learns other aspects of the problem.

The results in this paper reveal the kind of advantage that is gained from letting it do so. HyperNEAT indeed learns significantly faster than NEAT-EI, but more interestingly, even when comparing *equally* successful solutions evolved by both methods to defeat the same opponent, those evolved by HyperNEAT *generalize* significantly more effectively against variants of that opponent. The implication is that HyperNEAT discovers *regularities* in the geometry of the game, which is essential for general performance. General patterns are easier to represent than policies that separately analyze every individual location on the board.

Another advantage of learning from geometry is simply that it removes the need for humans to decide how to engineer the representation. Thus, the primary lesson of this study is that machine learning should begin to integrate a capacity to learn geometric correlations and regularities in the task domain into its algorithms by providing them explicit access to the domain geometry. The reward will be significantly more general solutions to real world problems.

Conclusion

This paper argued that learning regularities across the geometry of the problem domain is critical to effective generalization. To demonstrate this point, three variants of the same learning approach with differing degrees of access to geometric information were trained against a single deterministic checkers heuristic. The main result is that the quality of learning and generalization is directly proportional to the representation's ability to learn from geometry. In fact, the most powerful representation, HyperNEAT, learns geometric regularities on its own, leading to significantly improved generalization against a randomized version of the training opponent. The main conclusion is that, to reach its full potential, machine learning should include the ability to learn from the geometry of the task.

References

[1] L. Altenberg. Evolving better representations through selective genome growth. In *Proceedings of the IEEE World Congress on Computational Intelligence*, pages 182–187, Piscataway, NJ, 1994. IEEE Press.

[2] P. J. Bentley and S. Kumar. The ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In *Proc. of the Genetic and Evo. Computation Conference*, pages 35–43, San Francisco, 1999. Kaufmann.

[3] K. Chellapilla and D. B. Fogel. Evolving an expert checkers playing program without using human expertise. *IEEE Trans. Evolutionary Computation*, 5(4):422–428, 2001.

[4] D. D'Ambrosio and K. O. Stanley. A novel generative encoding for exploiting neural network sensor and output geometry. In *Proc. of the Genetic and Evolutionary Computation Conference 2007*, New York, NY, 2007. ACM Press.

[5] M. Fierz. Simplech. http://arton.cunst.net/xcheckers/, September 22, 2002.

[6] J. Gauci and K. O. Stanley. Generating large-scale neural networks through discovering geometric regularities. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, New York, NY, 2007. ACM Press.

[7] B. R. Leffler, M. L. Littman, and T. Edmunds. Efficient reinforcement learning with relocatable action models. In *AAAI*, pages 572–577. AAAI Press, 2007.

[8] A. Lindenmayer. Adding continuous components to L-systems. In G. Rozenberg and A. Salomaa, editors, *L Systems, Lecture Notes in Computer Science 15*, pages 53–68. Springer-Verlag, Heidelberg, Germany, 1974.

[9] A. A. Sherstov and P. Stone. Function approximation via tile coding: Automating parameter choice. In J.-D. Zucker and L. Saitta, editors, *SARA*, volume 3607 of *Lecture Notes in Computer Science*, pages 194–205. Springer, 2005.

[10] K. O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines Spec. Issue on Dev. Sys.*, 2007.

[11] K. O. Stanley and R. Miikkulainen. A taxonomy for artificial embryogeny. *Artificial Life*, 9(2):93–130, 2003.

[12] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:99–127, 2002.

[13] K. O. Stanley and R. Miikkulainen. Evolving a roving eye for Go. In *Proc. of the Genetic and Evolutionary Computation Conference (GECCO-2004)*, Berlin, 2004. Springer Verlag.

[14] K. O. Stanley and R. Miikkulainen. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21:63–100, 2004.

[15] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci. A hypercube-based indirect encoding for evolving large-scale neural networks. *Artificial Life*, 2008. To appear.

[16] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[17] J. Togelius and S. M. Lucas. Forcing neurocontrollers to exploit sensory symmetry through hard-wired modularity in the game of cellz. In *CIG*. IEEE, 2005.

[18] A. Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society B*, 237:37–72, 1952.