Comparing Direct and Indirect Encodings Using Both Raw and Hand-Designed Features in Tetris

Lauren E. Gillespie
Dept. of Math and Computer Science
Southwestern University
Georgetown, Texas 78626 USA
gillespl@southwestern.edu

Gabriela R. Gonzalez
Dept. of Math and Computer Science
Southwestern University
Georgetown, Texas 78626 USA
gonzale9@alumni.southwestern.edu

Jacob Schrum
Dept. of Math and Computer Science
Southwestern University
Georgetown, Texas 78626 USA
schrum2@southwestern.edu

ABSTRACT

Intelligent agents have a wide range of applications in robotics, video games, and computer simulations. However, fully general agents should function with as little human guidance as possible. Specifically, agents should learn from large collections of raw state variables instead of small collections of hand-designed features. Learning from raw state variables is difficult, but can be easier when agents are aware of the geometry of the input space. Indirect encodings allow agents to take advantage of the geometry of the task, and scale up to large input spaces. This research demonstrates the relative benefits of a direct and indirect encoding using raw or hand-designed features in Tetris, a challenging video game. Specifically, the direct encoding NEAT is compared against the indirect encoding HyperNEAT. Both algorithms create neural networks to play the game, but HyperNEAT makes better use of raw screen inputs, due to its ability to generate large networks that take advantage of the domain's geometry. However, hand-designed features lead to higher scores with both algorithms. HyperNEAT makes better use of hand-designed features early in evolution, but NEAT eventually overtakes it. Since each method succeeds in different circumstances, approaches combining the strengths of both should be explored.

CCS CONCEPTS

•Computing methodologies → Neural networks; Generative and developmental approaches;

KEYWORDS

Games, Tetris, Indirect encoding, Neural networks

ACM Reference format:
Lauren E. Gillespie, Gabriela R. Gonzalez, and Jacob Schrum. 2017. Comparing Direct and Indirect Encodings Using Both Raw and Hand-Designed Features in Tetris. In *Proceedings of GECCO '17, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3071178.3071195

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GECCO '17, Berlin, Germany
© 2017 ACM. 978-1-4503-4920-8/17/07...$15.00
DOI: http://dx.doi.org/10.1145/3071178.3071195

1 INTRODUCTION

Artificial Intelligence (AI) has been successfully applied to many well-known game domains and has even outperformed human champions in domains like Chess [7] and Go [22]. However, success in many challenging domains depends on the use of intelligent, hand-designed features [11, 16, 21, 32], although there are exceptions to this trend [12, 29], especially thanks to the emergence of Deep Reinforcement Learning [17, 22].

Deep Reinforcement Learning and other methods that work with large neural networks (such as HyperNEAT [25], which is used in this paper) typically use raw or minimally processed visual information as input organized in a manner that matches the domain's layout on the screen, i.e. these networks work with what a human would see. However, such massive feature sets can be difficult to learn from. In contrast, hand-designed features are carefully chosen in order to give useful knowledge to a learning agent, and filter out worthless or distracting information. Though effective, such an approach requires considerable human effort, and must be repeated for each new domain in which a learning algorithm is applied. This restriction has sparked much interest in agents that can learn in multiple domains, such as General Game Playing for board games [13] and General Video Game Playing [18]. There has also been interest in methods that can play all games from the Atari 2600 suite [15, 17].

Like the Atari suite, Tetris is a well-known and popular domain. In Tetris, artificially intelligent agents have had success with approaches such as evolved evaluation functions [3], CMA-ES [4], Temporal Difference Learning [1, 14], and Policy Search [27]. However, these approaches rely solely on intelligent, hand-designed features. There does not seem to be any published work in which Tetris is learned using raw screen inputs.

In this paper, agents learn to play Tetris using both raw screen inputs and intelligent, hand-designed features with two neuroevolution methods: the direct-encoding NEAT [26] and the indirect-encoding HyperNEAT [25]. NEAT is a popular means of evolving artificial neural networks (ANNs) with arbitrary topologies and has been applied in many game domains in the past [8, 20, 21, 24]. However, NEAT genotypes are directly encoded, meaning that each gene corresponds to one component of the network, which creates difficulties when scaling up to very large input spaces. This limitation was part of the motivation behind the development of HyperNEAT, an indirect encoding that can easily create massive neural networks through a generative process that derives phenotype networks of arbitrary size from compact genotypes. HyperNEAT has also been applied to game domains and has been successful using visual inputs in these domains [12, 15, 33].

Considering the success of HyperNEAT in previous domains using visual inputs, Tetris is a prime candidate in which to apply HyperNEAT using raw screen inputs. When compared with NEAT, HyperNEAT performs much better using raw screen inputs, though still poorly in comparison with previous attempts using hand-designed features. Therefore, NEAT and HyperNEAT are also compared using hand-designed features, which increases the performance of both methods. Applying hand-designed features with HyperNEAT is not straightforward, because not all features are geometrically related. Nevertheless, it can be done and results in reasonably good play. However, the performance of regular NEAT eventually reaches the same median level of performance, and the overall distribution of champions has more higher scores. These comparisons reinforce the notion that indirect encodings have clear strengths, but are not universally superior to direct encodings. In particular, this paper gives an example of how indirect encodings can be harder to apply and less effective with hand-designed features.

This paper proceeds by first detailing the domain of Tetris followed by an in-depth look at pertinent previous research, a brief overview of the relevant evolutionary algorithms, experimental setup and finally a discussion of the final results.

2 TETRIS

Tetris was created in 1989 by Russian game designer Alexey Pajitnov. It is now one of the most ubiquitous games on the planet and is available on nearly all devices that support video games. Its relatively simple game mechanics combined with a large state space are what make it such an interesting and challenging domain. This section discusses the gameplay of Tetris, followed by a review of previous research in this domain.

2.1 Gameplay

In the game of Tetris, pieces called tetrominoes (Figure 1), that are various configurations of four blocks, are randomly selected to slowly fall one by one from the top of the screen. As a piece falls, the player can move it from side to side and rotate it in order to move it into a desirable position. The player seeks to completely fill horizontal rows of a 10 block wide and 20 block high board with blocks from the tetrominoes. Sometimes, placement of a piece can create holes, which are open spaces with at least one block above them. Some holes can be filled by moving in a piece from the side as it falls, but holes can also become completely covered by falling pieces. The accumulation of holes should be avoided since they prevent rows from being filled. When a row is completely filled, the row disappears and all blocks above it are shifted down one space. It is possible to clear multiple rows at once, and the clearing of rows earns points. In the implementation used in this paper (part of RL-Glue [28]), one, two, four or eight points are earned for simultaneously clearing one, two, three or four lines, respectively. Clearing four lines simultaneously is called a *Tetris*. Play continues until the blocks reach the top of the screen, at which point the game is lost. The goal is to maximize the score, but this goal is tied to the goal of playing for as long as possible.

Part of what makes Tetris so hard and therefore so interesting is the wide variety of pieces and how it is impossible to place every sequence of blocks in a way that the player will not lose, as research

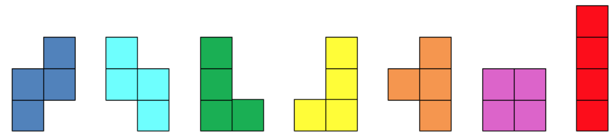

Figure 1: Tetrominoes. The seven available tetromino pieces in the Tetris domain. Each is referred to using the letter it most closely resembles, from left to right: Z, S, L, J, T, O, I.

indicates that the presence of Z and S shaped tetrominoes assures the eventual termination of every game [6]. Furthermore, Tetris is an NP-complete problem, even when the player knows the identity and order of all the pieces [5]. However, players can plan better if they are aware not only of the currently falling piece, but of the identity of the piece that will follow, as is allowed in some implementations of the game. Controllers that use this extra information are known as two-piece controllers, while those that are only aware of the current piece are one-piece controllers [30]. The RL-Glue [28] implementation of Tetris used in this paper uses one-piece controllers.

Much work has been done in developing intelligent controllers to play Tetris using the RL-Glue simulator and other custom implementations. The next section details much of this work.

2.2 Previous Work

Beyond becoming a widely popular game, Tetris has also become a popular benchmark for AI study and competitions. Many different approaches have been applied to Tetris. However, these approaches to Tetris often make use of a simplified version of the game that reduces the domain to choosing which column and rotation in which to drop the piece. This version ignores the restriction in the original game of potentially being unable to move and rotate a piece quickly enough to either side in order to place it in the desired manner due to highly stacked blocks obstructing movement. It has been shown that this simplified version improves the number of lines an agent is able to clear [30]. When researchers apply their methods to the full version of the game, they generally learn an afterstate evaluator, which only considers game states that can possibly be reached.

An early and influential approach to Tetris was that of Bertsekas et al. [1, 2], known as λ-Policy Iteration. However, this work is most influential for the hand-designed features it introduced, which have been used in many other approaches. These features will be referred to as the Bertsekas features and are described in Section 4.2.

The reinforcement learning community has been especially active in applying various algorithms to Tetris, frequently making use of the Bertsekas features. For example, the noisy cross-entropy method of policy search [27] performed one hundred times better than the original results of Bertsekas et al. Later work using Approximate Dynamic Programming was able to roughly match this performance with fewer samples [11]. Tetris was even a test domain included in the 2008 Reinforcement Learning Competition [34]. The winning approach was an extension of the cross entropy method that also introduced some additional new features [31].

Comparatively little work has been done with evolutionary computation in Tetris. Some examples include a simple evolutionary

algorithm that assigned weights to a linear function approximator [3] and an application of the Covariance Matrix Adaptation Evolution Strategy [4]. Neuroevolution in general and NEAT in particular have also not been applied, nor has HyperNEAT.

None of the work mentioned above made use of raw screen inputs from the 10 by 20 screen grid. Instead, all made use of some collection of hand-designed features. Regardless, the state of the art in Tetris is quite skilled. The purpose of this paper is not to challenge the state of the art, but to explore the comparative success of direct and indirect encodings using both raw and hand-designed features, using Tetris as an example domain. The next section describes the specific evolutionary approaches being compared.

3 EVOLUTIONARY ALGORITHMS

Directly and indirectly encoded neural network afterstate evaluators for Tetris are evolved using NEAT (Neruo-Evolution of Augmenting Topologies [26]) and HyperNEAT (Hypercube-based NEAT [25]), respectively. In addition to using the standard game score as a fitness function, an additional shaping objective is used (number of time steps), necessitating the use of the multiobjective evolutionary algorithm NSGA-II (Non-dominated Sorting Genetic Algorithm II [10]). Code from all experiments is available as part of MM-NEAT[1].

3.1 NSGA-II

The Non-Dominated Sorting Genetic Algorithm II (NSGA-II [10]) is a Pareto-based multiobjective evolutionary optimization algorithm. Its use makes the inclusion of an additional shaping objective described in Section 4.5 straight-forward, because no weighting of different objectives is necessary. The final results are still evaluated entirely in terms of game score, which is the main objective of interest. However, because multiple objectives are used during evolution, a principled way of dealing with them is needed.

NSGA-II uses the concepts of Pareto Dominance and Pareto Optimality to sort a population into Pareto layers according to their objective scores. Each layer consists of agents whose scores do not Pareto dominate the scores of others in the same layer. One score only dominates another if it is at least as good in all objectives, and strictly better in at least one objective. Thus, in a multiobjective sense, the layer whose scores are not dominated by any scores in the population (the Pareto front) consists of the best individuals, who are therefore most worthy of selection and reproduction. Individuals in the layer beneath this one are second-best, and so on.

NSGA-II uses $(\mu + \lambda)$ elitist selection favoring individuals in higher layers over those in lower fronts. In the $(\mu + \lambda)$ paradigm, a parent population of size μ is evaluated, and then used to produce a child population of size λ. Selection is performed on the combined parent and child population to give rise to a new parent population of size μ. NSGA-II typically uses $\mu = \lambda$.

When performing selection based on which Pareto layer an individual occupies, a cutoff is often reached such that the layer under consideration holds more individuals than there are remaining slots in the next parent population. These slots are filled by selecting individuals from the current layer based on a metric called *crowding distance*, which encourages the selection of individuals in less-explored areas of the trade-off surface between objectives. A combination of dominance criteria and crowding distance is used to derive each new child population from the preceding parent population.

NSGA-II provides a way to select the best solutions based on multiple objectives, but it is indifferent as to how these solutions are represented. In this paper, NSGA-II was used to evolve artificial neural networks using either the direct encoding NEAT, or the indirect encoding HyperNEAT.

3.2 NEAT

Neuro-Evolution of Augmenting Topologies (NEAT [26]) has been a benchmark neuroevolutionary algorithm for years. The main attraction of NEAT is that it can evolve a network's topology in addition to its weights. NEAT starts with an initial population of simple, fully connected networks with no hidden nodes. Throughout evolution, NEAT gradually complexifies the networks by augmenting the topology through mutations that add new links and nodes. The weights of existing network links can also be modified by mutation.

NEAT also allows for crossover between networks during reproduction. In order to account for competing conventions resulting from different topological lineages, NEAT assigns historical markers to each link and node within the genome, which allows for efficient alignment of network components with shared origin.

However, each of these components in the genome directly corresponds to a component of the network phenotype, which is why NEAT is classified as a direct encoding. A consequence of this encoding is that the size of an evolved network is proportional to the size of its genome. Despite this drawback, NEAT has been successful in many video game domains [8, 20, 21, 24].

However, these applications each use less than 40 carefully chosen features. When scaling up to larger numbers of features that are simpler and less informative, direct encodings like NEAT struggle to make progress. The reason for this struggle is that a single structural mutation (i.e. the addition of one new link or node) does not do much to significantly modify the behavior of an agent, yet many such mutations are needed in order for a large network to optimize its behavior. Furthermore, large random changes to a genotype are not likely to be beneficial anyway, so mutations whose impact across the phenotype is large but regular are needed. Finally, NEAT networks have no way of leveraging information about the geometric organization of various inputs if such information is available. All of these shortcomings of NEAT are addressed by HyperNEAT, an extension of NEAT described next.

3.3 HyperNEAT

Hypercube-based NEAT [25] is an indirect encoding that extends NEAT by evolving networks that encode the connectivity patterns of typically larger *substrate* networks, which are the evaluated networks in a given domain. Specifically, HyperNEAT genotypes are Compositional Pattern-Producing Networks (CPPNs [23]), which differ from standard ANNs in that their activation functions are not limited to a single type, such as the standard sigmoid or hyperbolic tangent functions. Instead, CPPN activation functions can be any of collection of functions that produces useful patterns, such as symmetry and repetition. As such, the specific activation functions used in this work are: sigmoid, Gaussian, sine, sawtooth wave, absolute value, and identity clamped to the range [0,1]. This wide selection

[1] Download at http://nn.cs.utexas.edu/?mm-neat

of activation functions also necessitates the addition of a mutation operation that randomly replaces the activation function of a given node with another one from the set of possibilities.

However, what makes CPPNs distinct from typical neural networks, and what makes HyperNEAT so powerful, is how these CPPNs are *used*. The CPPNs are repeatedly queried across a neural substrate, and the outputs of the CPPN are used to construct a neural network within that substrate, thus making the substrate network indirectly encoded by the CPPN. These substrates are collections of layered neurons assigned explicit geometric locations with predetermined potential connections between neurons in different layers. The layout of the substrate is defined by the experimenter and is domain-specific. It specifies how many substrate layers are needed, how many neurons are in each layer, which are input, hidden and output neurons, and where the neurons are located.

When the CPPN is queried across these substrates, it is determining whether a potential link between layers will exist, and if so, what its weight will be. Since this paper extracts Tetris inputs from a 2D screen, the substrate layers are 2D as well. Thus, all CPPNs have five inputs: The x and y coordinates of both source and target neurons in different substrate layers, and a constant bias of 1.0. The manner in which a CPPN is queried to create a substrate network for processing raw Tetris screen inputs is shown in Figure 2.

The CPPN has a separate output for each pair of substrate layers between which connections can exist. For any pairing of x/y-coordinates that are input to the CPPN, the output corresponding to the given substrate layer pairing is the one that defines the link. Specifically, links are only expressed if the magnitude of the relevant CPPN output exceeds 0.2. For links that are expressed, the output value is scaled toward 0 to eliminate the region between -0.2 and 0.2. The resulting value then becomes the link weight. The use of these separate outputs avoids the need for a z-coordinate input to the CPPN to account for how the substrate layers are stacked and allows the connection patterns between different pairs of layers to be drastically different rather than simply varying in accordance with z.

Finally, CPPNs also have outputs that define a constant bias associated with each neuron in a non-input substrate. In a typical neural network, such a bias could easily be applied with the use of an additional input with a constant value. However, in a substrate network, all neurons require geometric coordinates. Because a bias is detached from the actual inputs, it needs to effectively occupy its own substrate. Having additional CPPN outputs effectively accomplishes this, and allows each neuron to store its bias value and add it to subsequently received inputs, which is equivalent to actually having a separate bias input.

The geometric mapping of inputs onto substrate layers is a novel innovation of HyperNEAT, but it has both advantages and disadvantages. The geometric nature of the substrate allows for the agent network to take advantage of task-relevant geometry of the domain, because typically there is an important relationship between the geometry of a state space and how an agent should evaluate states from that space. The geometric nature of the substrate allows for the agent network to access information about the geometric regularities of the domain. However, intelligent, hand-designed features are less likely to have natural geometric associations with each other, which makes embedding them into geometrically organized substrates problematic. The most straight-forward solution to this

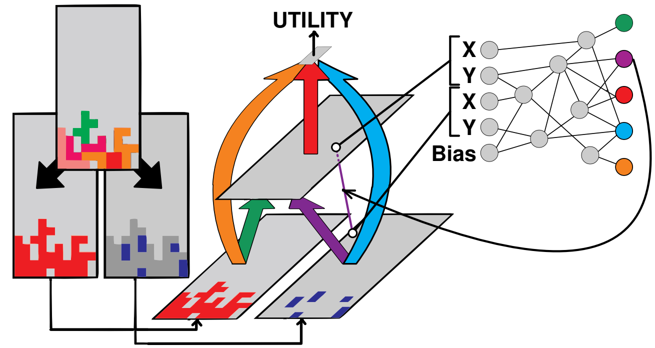

Figure 2: Indirect Encoding of Tetris State Evaluators with HyperNEAT using Raw Screen Inputs. Raw screen inputs are fed to the network as two 10 by 20 substrate grids, one identifying all current blocks, and another identifying all current holes. These grids are then mapped to the corresponding input substrate layers in the large neural network created by the CPPN on the right. Each colored arrow between substrate layers indicates that every possible feed-forward link between these layers is queried by the CPPN. The inputs that define a link are the x/y-coordinates within the plane of each layer of the source and target nodes. There is also a constant bias input to the CPPN. The weight of a link depends on the output of the neuron in the CPPN corresponding to the arrow color. The querying and creation of one example link is shown. The CPPN also has special outputs defining internal bias values within each non-input neuron, but these outputs are left out of the figure for clarity. The output layer of the substrate network consists of a single neuron that defines the utility of the given Tetris board game state. This architecture is aware of the geometry of the input space and allows HyperNEAT to learn how to play Tetris using raw screen inputs.

problem is to embed groups of geometrically related features together on the same substrate, while having separate input substrates for each such group [19]. However, when there are features that are geometrically distinct from all other features, they need to be embedded on small substrate layers with a single neuron. Because each pairing of connected substrate layers necessitates an additional CPPN output, a proliferation of geometrically distinct groups of features can considerably bloat a CPPN, as can be seen in Figure 3.

The consequences of how HyperNEAT genotypes and phenotypes are configured in contrast to those of NEAT are explored in the experiments discussed next.

4 EXPERIMENTAL SETUP

The following sections describe how Tetris agents used neural networks to determine their actions, the hand-designed and raw screen inputs used by these networks, the specific configuration of HyperNEAT substrate layers, and the general evaluation parameters used in all experiments.

4.1 Afterstate Evaluation

To determine where each piece will go, the agent uses the evolved network as an afterstate evaluator to make decisions about piece placement. Before a new piece begins to fall, a search algorithm considers every possible location in which the piece could be placed and remembers the sequence of moves leading to each placement. Placements that lead to an immediate loss are not considered. Ruling

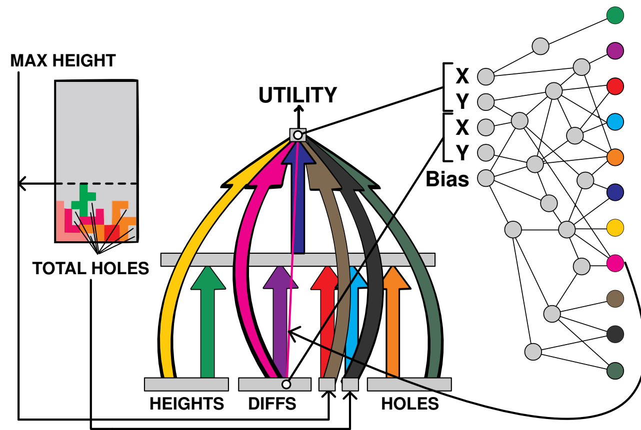

Figure 3: Indirect Encoding of Tetris State Evaluators with HyperNEAT using Hand-Designed Features. Input substrate layers for hand-designed inputs are grouped in terms of shared geometric relations. Specifically, the ten column heights, nine column differences and ten column hole counts are grouped in 1D substrates from left to right. The max height and total hole count features each have their own 1 by 1 substrates. Each of these layers has links to a common hidden layer and to the output neuron. These links are defined by the CPPN outputs shown on the right. The presence of many more substrate layers means that more CPPN outputs are needed. The y-coordinate inputs of the CPPN are not actually relevant because these substrate layers are 1D (y will always equal 0), but the extra inputs are kept for consistency with the raw input experiments.

out loss states in this way allows even a poor agent to survive slightly longer than it would otherwise. For both hand-designed and raw features, inputs for the network are taken from the states that result from each valid placement. Note that if a placement fills a row, that row will be cleared before the state's feature values are calculated. For each of these afterstates, the network will produce a single output, which is a utility score between -1 and 1. The piece placement with the highest utility score across all possible placements is selected, and the Tetris agent then carries out the remembered sequence of actions that lead to the desired placement. After the placement, a new piece appears and the process repeats. Since placement evaluation occurs before the piece begins to fall, evaluation time does not in any way exclude available piece placements.

Because these experiments use the full version of Tetris and not the simplified version mentioned in Section 2.1, some placements are not possible and are therefore not considered for evaluation. In particular, it may not be possible to move a piece to the side quickly enough to stack it on top of blocks piling up on the sides, because the piece may have fallen beneath the top of the stack by the time it has moved far enough to the side. However, moves that involve waiting for a piece to fall for a while before moving it sideways into an exposed hole are possible.

4.2 Hand-Designed Features

The hand-designed features used in these experiments are the Bertsekas features [1] mentioned in Section 2.2, plus additional features described below. The Bertsekas features include the heights of each of the ten columns, the height differences between all nine pairs of adjacent columns, the height of the tallest column and the total number of holes. These features have already proven themselves in many other Tetris experiments [11, 14, 27, 30, 31]. However, additional features were added to provide the agent with more fine-grained information about the holes. Specifically, the number of holes present per column was included to allow a more fair comparison with the raw input setup, which also includes more specific information about the locations of holes (Section 4.3).

All of these features are scaled to the range [0,1]. In this paper, standard NEAT networks have an additional constant bias input of 1, whereas HyperNEAT networks need no bias input due to the biases being stored in each individual neuron, as discussed in Section 3.3. As a result, NEAT uses 32 hand-designed feature inputs, whereas HyperNEAT only uses 31. Figure 3 shows how these inputs would be fed into a network produced by HyperNEAT.

4.3 Raw Screen Inputs

For all networks evolved with raw screen inputs, the current board configuration was split into two sets of inputs. The first set identified the locations of all blocks, using 1 to represent the presence of a block and 0 the absence of a block. The second set identified only the locations of holes, which had a value of -1, while all other locations had a value of 0. Including the locations of holes made it easier to distinguish holes from empty non-hole locations. Both types of networks used these inputs, but NEAT networks also received a constant bias input of 1, making for a total of 401 inputs for NEAT networks and 400 for HyperNEAT networks. Once again, HyperNEAT does not need a distinct bias input for reasons described in Section 3.3. Figure 2 shows how the screen inputs are split into two sets and fed to a substrate network produced by HyperNEAT.

4.4 HyperNEAT Substrates

For HyperNEAT substrates, only one hidden layer was used for all experiments. Some preliminary experiments were conducted with more hidden layers, and although the added neurons generally did not hurt performance, they also did not help it. Therefore, large experiment batches were conducted only with the smaller networks. As is standard when configuring neural networks, each input layer was fully connected to the hidden layer, and the hidden layer was fully connected to the output layer. Additionally, each input layer was also fully connected to the output layer, since much previous successful work in Tetris was done with simple linear function approximators. Since signals could reach the output neuron both through the hidden layer and directly from the inputs, networks had the option of bypassing the hidden layer if it suited them.

HyperNEAT networks evolved with raw screen inputs had two separate input substrates. The first substrate was for the block locations and second was for the hole locations. The 2D organization of this data contrasts with the input organization for NEAT networks, which was arranged linearly from left to right and top to bottom with the block location inputs preceding the hole location inputs. The size of the hidden layer was chosen to equal the size of one input board, and is thus 10 by 20, containing 200 hidden neurons.

The x/y-coordinates sent as CPPN inputs were always normalized so that the CPPN had the ability to represent patterns at arbitrary resolution. Such normalization usually places the origin in the center

of the substrate, but there is no geometric relevance about the vertical center of the Tetris board. Rather, distance from the bottom of the board is important. Horizontal distance from the vertical midline to the sides is also relevant. Therefore, the Tetris board had its x-coordinates mapped to [-1,1] and its y-coordinates mapped to [0,1] (origin at bottom center). The way the CPPN creates the substrate network and accepts raw screen inputs is shown in Figure 2.

CPPNs evolved with hand-designed features were actually more complicated than their raw screen counterparts due to the nature of the hand-designed features. The ten height features have a 1D geometric relationship to each other, as do the nine column difference and ten column hole features. However, none of these subsets of features have any geometric relationship to the others. The maximum height and total holes features are also stand-alone features that, despite having a relation to the column heights and holes respectively, do not have a clear *geometric* relationship to these feature subsets. Therefore, each of these five categories of features resides on its own input substrate layer. Because the total number of these features is 31, this is also the number of neurons in the 1D hidden layer.

Note that because each of these substrate layers is linear, the y-coordinate inputs of the CPPN are superfluous. The y-coordinate input is 0 in all cases, and only x varies, with the middle of each linear substrate having an x-coordinate of 0. The y-coordinate inputs were kept for consistency across runs, but removing them should have no negative effect, and could even improve performance slightly.

4.5 Evaluation Setup

Each experiment consisted of 30 runs per approach lasting 500 generations with a population size of $\mu = \lambda = 50$. Tetris is a noisy domain, meaning that repeated evaluations are likely to yield wildly different scores due to randomness in the sequence of tetrominoes that fall in each game. To mitigate the noisiness slightly, the fitness scores for each agent were objective scores averaged across three trials. The specific objectives used when evaluating each generation with NSGA-II were the game score and the number of time steps the agent survived. The time steps objective was included because survival is always good, since it provides more time to score points and is particularly useful in the early stages of evolution when agents may not be able to clear any rows at all.

When creating the next generation of networks, there was a 50% chance an offspring network would be the product of crossover. Each offspring network also had a 5% chance per link of Gaussian perturbation. There was a 40% chance of adding a new link and a 20% chance of adding a new node. HyperNEAT CPPNs had a 30% chance of a randomly chosen node having its activation swapped with another random function from the available set (Section 3.3).

5 RESULTS

HyperNEAT and NEAT reached comparable levels of performance when using the intelligent hand-designed features (Figure 4), but there are important differences. From generation 0 to 200, HyperNEAT outperformed NEAT, with a median score of nearly 5,000 reached by generation 200 and outliers reaching up to 15,000. In contrast, NEAT networks only had a median score of around 1,000 by generation 200. However, by this point HyperNEAT performance had flattened out while NEAT performance kept climbing. In fact,

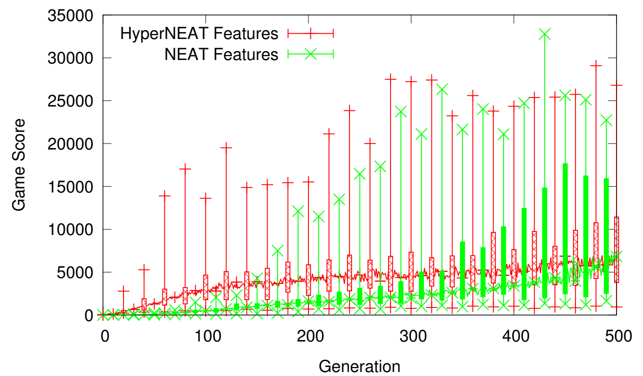

Figure 4: Boxplots of HyperNEAT and NEAT Performance with Hand-Designed Features. Boxplots depicting minimum, lower quartile, median, upper quartile and maximum performance of the champions from each trial are shown for NEAT and HyperNEAT across 30 runs with hand-designed features. The median is plotted for all generations, but boxplots are only shown at staggered intervals for the sake of readability. HyperNEAT performance is superior to NEAT early in evolution, but levels out while, in contrast, NEAT performance gradually grows. Median NEAT performance eventually reaches median HyperNEAT performance, and NEAT's upper quartile surpasses that of HyperNEAT. So although NEAT performance climbs more slowly, it does produce more strong results in the end.

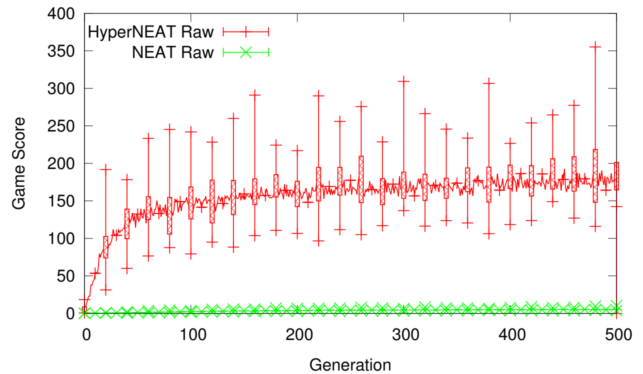

Figure 5: Boxplots of HyperNEAT and NEAT Performance with Raw Screen Inputs. Across 30 runs for each method using raw screen inputs, performance is depicted in the same manner as in Figure 4. HyperNEAT performance shoots up before reaching a plateau, but NEAT is incapable of learning to work with the raw inputs at all.

the distribution of NEAT scores begins to shift upward, with upper quartile and maximum scores reaching higher and higher. Near generation 500, both approaches had tied median scores and tied maximum scores, but the upper quartile of NEAT was much higher than that of HyperNEAT runs. However, this slight difference in the final generation was not significant according to a two-tailed Mann-Whitney U test ($U = 500, N = 30, p \approx 0.4671$).

In contrast, HyperNEAT vastly outperformed NEAT when using raw screen inputs (Fig. 5), and this difference *is* significant ($U = 0, N = 30, p \approx 2.82 \times 10^{-11}$) because all HyperNEAT scores are

higher than all NEAT scores. Across runs, HyperNEAT reaches a median score of 200 and its maximum scoring run reached 350. In contrast, NEAT networks were unable to reach a score of more than 15 even by generation 500. These scores are of course much lower than what is achieved using the intelligent hand-designed features, but HyperNEAT at least demonstrates that it is capable of learning to play the game, whereas NEAT was completely incapable.

Videos of representative behaviors provide insight into how different methods play the game. These videos can be seen at southwestern.edu/~schrum2/SCOPE/tetris.html. The best performing agents for HyperNEAT and NEAT with raw inputs each developed distinct behaviors, which is not surprising given the vast difference in scores.

NEAT champions using raw features exhibit the same behavior of stacking pieces on the sides of the board until they were full, thus forcing the agent to finally fill in the center of the board. However, this filling was done so inefficiently and created so many holes that lines were rarely cleared. In fact, by the time that these agents were forced to place pieces in the middle area of the screen, the few lines that they cleared seemed to be mostly coincidental. Once the center began to fill up, it was too late for the network to overcome the maze of holes it had created and it quickly lost soon after.

In contrast, the HyperNEAT networks using raw screen inputs play the game competently. These agents stack pieces efficiently and clear lines reasonably well. Sometimes they leave one column unfilled which creates an opportunity to get a Tetris (four lines cleared) whenever an I-shaped tetromino appears. Clearing multiple lines simultaneously is worth more points, so it makes sense that such a behavior would arise. However, if no I-shaped tetromino appears, the agent ends up in a dangerous situation. Often, HyperNEAT networks would settle for using L-shaped and J-shaped tetrominoes to fill these gaps, clearing only two or three lines depending on the arrangement of blocks. This behavior ultimately led to the downfall of the HyperNEAT agents, because it sometimes created holes that were difficult to fill, often leading to a loss.

The behaviors exhibited by NEAT and HyperNEAT networks using hand-designed features were not radically different from each other, but the best results produced by NEAT were more efficient than their HyperNEAT counterparts. Both approaches favored clearing rows whenever the opportunity arose. Because of the randomness in which pieces fall, there are some situations in which a piling up of blocks is unavoidable, and it can be hard to tease out distinctions between approaches based purely on observation. However, both NEAT and HyperNEAT exhibit the occasional ability to clear out an arrangement of highly stacked blocks back to the bottom of the board. Ultimately, the main difference between NEAT and HyperNEAT when using hand-designed features seems to be that the best NEAT champions are less tolerant of holes, which allows them to maintain the level of the blocks at a lower height for longer.

6 DISCUSSION AND FUTURE WORK

The scores achieved by even the best NEAT networks using hand-designed features are not record-breaking, but the lessons learned from contrasting the performance of NEAT and HyperNEAT with both raw and hand-designed features are interesting.

Overall, it is clear to see the advantage HyperNEAT has over NEAT in the Tetris domain when using raw features. No NEAT agent using raw features was able to play competently at all, and all lines cleared seemed to be coincidence rather than strategy. NEAT agents likely only survived as long as they did due to the fact that loss states are never considered (Section 4.1).

However, this dismal performance by NEAT did give us pause, and led to the consideration of other experimental setups to make the comparison more fair. Specifically, we wondered if the simplicity of NEAT's initial population, lacking in all hidden neurons, might be putting NEAT at an unfair disadvantage. Therefore, preliminary runs were also conducted in which directly encoded networks evolved by NEAT were initialized with a fully connected version of the topology defined by HyperNEAT substrates using raw features. It was hoped that seeding the initial population in this fashion might provide it with useful structure. However, these seeded populations performed even worse than regular NEAT with raw screen inputs, solidifying the conclusion that HyperNEAT is far superior to NEAT when using these inputs. The reason that both variants of NEAT fail seems to simply be that the size of the genome is too large, and it is improbable that any sort of useful regular structure can evolve across such a large space using only isolated mutations. Seeding the initial structure simply added more structural components to the genome, making it even harder for NEAT to learn to play the game.

Another benefit that HyperNEAT has over NEAT is its ability to harness the geometry of the game space due to its indirect encoding. This geometric awareness HyperNEAT networks have makes it easier for said networks to learn the relative risks and benefits of having blocks and holes located in different regions of the board. For example, it only takes a few small mutations of a CPPN to encode a preference against having blocks exist in higher y-coordinates.

However, the usefulness of indirect encoding decreases when using the hand-designed features. Initially, the geometric awareness HyperNEAT encodes gave its networks an advantage. However, after a few hundred generations, HyperNEAT networks hit a consistent fitness ceiling while NEAT networks began to excel. This stagnation seems to be due in part to the limited usefulness of geometric information within these features: knowing the geometric layout of column heights might help a little, but simply favoring states in which all of these values are low is probably useful enough in most cases. Additionally, the column difference features actually already provide a bit of highly localized geometric information, making HyperNEAT's geometric awareness somewhat redundant. The fixed topology imposed by the HyperNEAT substrate may also contribute to the stagnation, since it prevents the growth of interesting hidden structures, and forces the CPPN to find a way to use a particular organization of hidden neurons (or to learn to ignore them).

In contrast, NEAT can directly interpret these hand-designed features fairly easily because there are so few of them. Lacking a means of interpreting these features in a regular way across the horizontal dimension seems to have made learning slow early on, but as mutations in disparate parts of the genome happened to coincidentally complement each other, and gradual complexification of the genome allowed it to refine its feature processing in more nuanced ways, performance continued to rise. In fact, the learning curve for the NEAT runs still seems to be moving upward at the end of 500 generations.

The results of experiments with hand-designed features showed that HyperNEAT was more successful at first, with NEAT overtaking its performance in later generations. This phenomenon indicates

that Tetris is a prime candidate in which to apply the Hybridized Indirect and Direct encoding (HybrID [9]) algorithm, which is a hybrid between HyperNEAT and NEAT. How HybrID works is that substrate networks are evolved with HyperNEAT for a set number of generations, and once this cutoff is reached, the CPPNs generating these networks are discarded, and further evolution occurs using only direct encodings of the generated substrate networks. HybrID can capitalize on the strengths of both HyperNEAT and NEAT.

Another direction for future work in Tetris is to apply Deep Reinforcement Learning using raw screen inputs. There has been much research done with other reinforcement learning methods in Tetris, but no published work using Deep Reinforcement Learning yet exists. Recent successes in Atari [17] and Go [22] indicate that Tetris is also a prime candidate for this approach.

7 CONCLUSION

Tetris is a well-known and popular domain that has been thoroughly studied by the AI community. However, this previous research had not focused on neuroevolution approaches or on the use of raw screen inputs. Two neuroevolution algorithms, NEAT and HyperNEAT, were compared to see how directly and indirectly encoded networks using both raw and hand-designed inputs would perform in this complicated domain. Overall, HyperNEAT was superior to NEAT when using raw features, thanks in part to the geometric awareness indirect encodings afford. However, NEAT ultimately produced more high scoring champions when using hand-designed features. Overall, these results indicate that despite past successes with HyperNEAT, more improvements are needed in order for it to perform better in an absolute sense when using raw visual information, and better with respect to standard NEAT when using hand-designed features. Further exploration of approaches such as HybrID, that combine the strengths of both NEAT and HyperNEAT, is interesting future work.

ACKNOWLEDGMENTS

This work is supported in part by the Howard Hughes Medical Institute through the Undergraduate Science Education Initiative Program under Grant No.: 52007558.

REFERENCES

[1] D. Bertsekas and S. Ioffe. 1996. *Temporal Differences-Based Policy Iteration and Applications in Neuro-Dynamic Programming*. Technical Report LIDS-P-2349. MIT.
[2] Dimitri P. Bertsekas and John N. Tsitsiklis. 1996. *Neuro-Dynamic Programming* (1st ed.). Athena Scientific.
[3] Niko Böhm, Gabriella Kókai, and Stefan Mandl. 2004. Evolving a Heuristic Function for the Game of Tetris. In *Lernen - Wissensentdeckung - Adaptivität*.
[4] A. Boumaza. 2009. On the Evolution of Artificial Tetris Players. In *Computational Intelligence and Games*. 387–393.
[5] Ron Breukelaar, Erik D. Demaine, Susan Hohenberger, Hendrik Jan Hoogeboom, Walter A. Kosters, and David Liben-Nowell. 2004. Tetris is hard, even to approximate. *International Journal of Computational Geometry and Applications* 14, 1–2 (2004), 41–68.
[6] Heidi Burgiel. 1997. How to Lose at Tetris. *Mathematical Gazette* 81, 491 (1997).
[7] Murray Campbell, A. Joseph Hoane, Jr., and Feng-hsiung Hsu. 2002. Deep Blue. *Artificial Intelligence* 134, 1–2 (2002), 57–83.
[8] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. 2009. Evolving Competitive Car Controllers for Racing Games with Neuroevolution.. In *Genetic and Evolutionary Computation Conference*. ACM, 1179–1186.
[9] Jeff Clune, Benjamin E. Beckmann, Robert T. Pennock, and Charles Ofria. 2009. HybrID: A Hybridization of Indirect and Direct Encodings for Evolutionary Computation. In *European Conference on Artificial Life*. 134–141.
[10] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6 (2002), 182–197.
[11] Victor Gabillon, Mohammad Ghavamzadeh, and Bruno Scherrer. 2013. Approximate Dynamic Programming Finally Performs Well in the Game of Tetris. In *Advances in Neural Information Processing Systems 26*. 1754–1762.
[12] Jason Gauci and Kenneth O. Stanley. 2008. A Case Study on the Critical Role of Geometric Regularity in Machine Learning. In *National Conference on Artificial Intelligence*. 628–633.
[13] Michael Genesereth and Yngvi Björnsson. 2013. The International General Game Playing Competition. *AI Magazine* (2013), 111.
[14] Alexander Groß, Jan Friedland, and Friedhelm Schwenker. 2008. Learning to Play Tetris Applying Reinforcement Learning Methods. In *European Symposium on Artificial Neural Networks*. 131–136.
[15] Matthew Hausknecht, Piyush Khandelwal, Risto Miikkulainen, and Peter Stone. 2012. HyperNEAT-GGP: A HyperNEAT-based Atari General Game Player. In *Genetic and Evolutionary Computation Conference*. 217–224.
[16] Patrick MacAlpine, Josiah Hanna, Jason Liang, and Peter Stone. 2016. UT Austin Villa: RoboCup 2015 3D Simulation League Competition and Technical Challenges Champions. In *RoboCup-2015: Robot Soccer World Cup XIX*, Luis Almeida, Jianmin Ji, Gerald Steinbauer, and Sean Luke (Eds.). Springer Verlag, Berlin, Germany. http://www.cs.utexas.edu/users/ai-lab/?macalpine:lnai15
[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. In *NIPS Deep Learning Workshop*.
[18] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, A. Coutoux, J. Lee, C. U. Lim, and T. Thompson. 2016. The 2014 General Video Game Playing Competition. *IEEE Transactions on Computational Intelligence and AI in Games* 8, 3 (2016), 229–243.
[19] Justin K. Pugh and Kenneth O. Stanley. 2013. Evolving Multimodal Controllers with HyperNEAT. In *Genetic and Evolutionary Computation Conference*. 8. http://doi.acm.org/10.1145/2463372.2463459
[20] Jacob Schrum, Igor V. Karpov, and Risto Miikkulainen. 2011. UT^2: Human-like Behavior via Neuroevolution of Combat Behavior and Replay of Human Traces. In *Computational Intelligence and Games*. 329–336.
[21] Jacob Schrum and Risto Miikkulainen. 2016. Discovering Multimodal Behavior in Ms. Pac-Man through Evolution of Modular Neural Networks. *IEEE Transactions on Computational Intelligence and AI in Games* 8, 1 (2016), 67–81.
[22] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the Game of Go With Deep Neural Networks and Tree Search. *Nature* 529 (2016), 484–503.
[23] Kenneth O. Stanley. 2007. Compositional Pattern Producing Networks: A Novel Abstraction of Development. *Genetic Programming and Evolvable Machines* 8, 2 (2007), 131–162.
[24] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. 2005. Evolving Neural Network Agents in the NERO Video Game. In *Computational Intelligence and Games*.
[25] Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. 2009. A Hypercube-based Encoding for Evolving Large-scale Neural Networks. *Artificial Life* 15, 2 (2009), 185–212.
[26] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation* 10 (2002), 99–127.
[27] Istvan Szita and András Lörincz. 2006. Learning Tetris Using the Noisy Cross-Entropy Method. *Neural Computation* 18, 12 (2006), 2936–2941.
[28] Brian Tanner and Adam White. 2009. RL-Glue : Language-Independent Software for Reinforcement-Learning Experiments. *Journal of Machine Learning Research* 10 (2009), 2133–2136.
[29] G. J. Tesauro. 1994. TD-Gammon, a Self-teaching Backgammon Program, Achieves Master-level Play. *Neural Computation* 6, 2 (1994), 215–219.
[30] Christophe Thiery and Bruno Scherrer. 2009. Building Controllers for Tetris. *International Computer Games Association Journal* 32 (2009), 3–11.
[31] Christophe Thiery and Bruno Scherrer. 2009. Improvements on Learning Tetris with Cross Entropy. *International Computer Games Association Journal* 32 (2009).
[32] Niels van Hoorn, Julian Togelius, and Jürgen Schmidhuber. 2009. Hierarchical Controller Learning in a First-Person Shooter. In *Conference on Computational Intelligence and Games*. IEEE, 294–301.
[33] Phillip Verbancsics and Kenneth O. Stanley. 2010. Transfer Learning Through Indirect Encoding. In *Genetic and Evolutionary Computation Conference*. 8.
[34] Shimon Whiteson, Brian Tanner, and Adam White. 2010. The Reinforcement Learning Competitions. *AI Magazine* 31, 2 (2010), 81–94.