# Evolving Indirectly Encoded Convolutional Neural Networks to Play Tetris With Low-Level Features

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

## ABSTRACT

Tetris is a challenging puzzle game that has received much attention from the AI community, but much of this work relies on intelligent high-level features. Recently, agents played the game using low-level features ($10 \times 20$ board) as input to fully connected neural networks evolved with the indirect encoding HyperNEAT. However, research in deep learning indicates that convolutional neural networks (CNNs) are superior to fully connected networks in processing visuospatial inputs. Therefore, this paper uses HyperNEAT to evolve CNNs. The results indicate that CNNs are indeed superior to fully connected neural networks in Tetris, and identify several factors that influence the successful evolution of indirectly encoded CNNs.

## CCS CONCEPTS

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

## KEYWORDS

Games, Tetris, Indirect encoding, Neural networks

**ACM Reference Format:**
Jacob Schrum. 2018. Evolving Indirectly Encoded Convolutional Neural Networks to Play Tetris With Low-Level Features. In *GECCO '18: Genetic and Evolutionary Computation Conference, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205455.3205459

## 1 INTRODUCTION

Tetris is a challenging game that has received much attention from the AI community [15, 17, 27, 36, 37], but nearly all of this research relies on the use of a small set of high-level features. Deep Reinforcement Learning (deep RL [23]) would seem well-suited to succeeding in this domain using low-level features, but despite current interest in deep RL, there are almost no published papers in which the full game of Tetris is played using low-level features.

A notable exception is a paper applying the evolutionary indirect encoding HyperNEAT [30] to Tetris [16] rather than stochastic gradient descent (SGD). Although the HyperNEAT afterstate evaluators evolved in this paper far surpassed those evolved with the direct encoding NEAT [31], there is still room for improvement. In particular, the HyperNEAT networks used a fully connected architecture with a single hidden layer. In contrast, this current paper uses HyperNEAT to indirectly encode convolutional neural networks (CNNs) [21], which have risen to prominence with the success of Deep Learning.

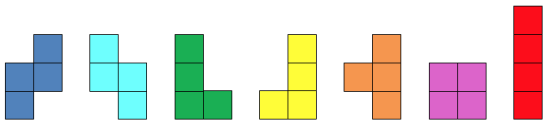


**Figure 1: Tetrominoes. The seven tetrominoes in Tetris, named using the letters they most closely resemble: Z, S, L, J, T, O, I.**

HyperNEAT has been used to purposefully [38] and inadvertently [13] encode CNNs before, for supervised and unsupervised tasks respectively, but these applications relied on SGD in addition to evolution. This paper uses HyperNEAT to generate CNNs for a Reinforcement Learning task without any use of SGD.

Experiments are conducted comparing evolved CNNs to fully connected networks in Tetris, and the results indicate that shallow CNN scores surpass those of the best fully connected networks by an order of magnitude. However, despite the high performance of the evolved CNNs, HyperNEAT struggles to evolve effective deep architectures with either type of connectivity pattern. After discussing these results, the paper concludes by proposing several approaches for enhancing evolution's performance in the future.

## 2 TETRIS

Tetris was created in 1989 by Russian game designer Alexey Pajitnov. It is an extremely popular puzzle game that has been repeatedly re-released on many platforms. Its relatively simple game mechanics combined with its large state space make it both 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


*GECCO '18, July 15–19, 2018, Kyoto, Japan*

ACM ISBN 978-1-4503-5618-3/18/07…$15.00
https://doi.org/10.1145/3205455.3205459

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 (from RL-Glue [35]), one, two, four or eight points are earned for simultaneously clearing one, two, three or four lines, respectively. 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.

Tetris is hard because it is impossible to place all pieces in a way that will not eventually lead to a loss, since the Z and S shaped tetrominoes assure the eventual termination of every game [5]. Furthermore, Tetris is an NP-complete problem, even when the player knows the identity and order of all the pieces [4]. Some Tetris implementations allow the player to know the next piece to fall, but agents in this paper are only aware of the current piece. Work developing intelligent controllers to play Tetris is described next.

## 2.2 Previous Research

Beyond being a widely popular game, Tetris is also a popular AI benchmark. An early approach to Tetris was that of Bertsekas and Ioffe [1], known as $\lambda$-Policy Iteration. This work is most influential for the high-level features it introduced: the individual column heights, the differences between adjacent columns, the maximum height across columns, and the number of holes on the board.

Many RL algorithms have been applied to Tetris using these or similar features. All of these approaches define state-value functions that consider what the board would look like after each possible placement of the current piece. For example, the noisy cross-entropy method [33] performed one hundred times better than the original results of Bertsekas and Ioffe. Approximate Dynamic Programming later matched this performance with fewer samples [15]. Tetris was also a domain in the 2008 RL Competition [40]. The winning approach was an extension of the cross entropy method [37].

Recently, record-breaking results were achieved in $10 \times 10$ Tetris (half board size) and SZ-Tetris (only S and Z pieces are available [34]) with an approach in which multiple learning runs execute in parallel to adapt hyperparameters for TD($\lambda$) [11]. This approach is like evolution, in that the different parallel runs are competing to promote their set of hyperparameters for use during learning.

More straight-forward variants of evolutionary computation have also been applied to Tetris. Examples include the evolution of weights for a linear function approximator [2] and an application of CMA-ES [3]. A recent approach explores both evolved heuristics, neural networks trained with SGD, and combination of both approaches [14], but once again uses a set of high-level features.

The most relevant work to this paper is the previously mentioned application of HyperNEAT using low-level features [16]. Unlike much work in deep RL [23], this work did not use raw pixel inputs, but did use simple binary features based on the presence/absence of blocks in the $10 \times 20$ game board. The same features are used in the current paper (Section 4.2). In the previous paper, NEAT and HyperNEAT evolved afterstate evaluators for Tetris using low-level or high-level features. NEAT was slightly, though not significantly, better than HyperNEAT with high-level features, but HyperNEAT was vastly superior to NEAT when using low-level features. One goal of the current paper is to surpass this previous HyperNEAT result by encoding CNNs instead of fully connected networks.

There are also applications of deep RL with TD($\lambda$) to SZ-Tetris using similar low-level features as input to an afterstate evaluator [10, 12]. These results are impressive, but cannot be directly compared to results in the full game. Also, these results relied on a special shaping reward not based on the game score. The current paper depends on using evolution to maximize game score and trial duration.

# 3 EVOLUTIONARY ALGORITHMS

This section explains the two versions of HyperNEAT [30] used in this paper: the original fully connected version (Full-HyperNEAT) and a new version that creates convolutional neural networks (CNN-HyperNEAT). Both versions extend the original NEAT [31], but use the multiobjective evolutionary algorithm NSGA-II [9] for selection. 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 [9]) is a Pareto-based multiobjective evolutionary optimization algorithm. NSGA-II makes the use of both game score and trial duration as separate objectives 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 networks whose scores are not Pareto dominated by scores of other networks in the same layer, or lower layers. One score only dominates another if it is at least as good in all objectives, and strictly better in at least one objective. Thus, the layer whose scores are not dominated by any scores in the population (the Pareto front) consists of the best individuals, which are 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 lower layers. In the $(\mu + \lambda)$ paradigm, a parent population of size $\mu$ is evaluated, and then produces a child population of size $\lambda$. Selection on the combined parent and child population creates a new parent population of size $\mu$. NSGA-II uses $\mu = \lambda$. To break ties among individuals in the same layer, *crowding distance* [9] favors solutions in sparsely occupied regions of that layer.

NSGA-II provides a way to select the best solutions based on multiple objectives. NSGA-II is used to evolve artificial neural networks using HyperNEAT, which is based on the original NEAT.

## 3.2 NEAT

NeuroEvolution of Augmenting Topologies (NEAT [31]) is a direct encoding neuroevolution algorithm that evolves both the topology and weights of its neural networks. NEAT starts with an initial population of simple networks with no hidden nodes, and gradually complexifies networks over generations by augmenting topology

[1] Download at http://people.southwestern.edu/~schrum2/re/mm-neat.php

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 allow for efficient alignment of network components with shared origin.

Other important components of NEAT are the use of speciation and fitness sharing to influence selection. However, no speciation is used in this paper because the selection mechanisms of NSGA-II completely override those of NEAT. Despite the discarding of speciation, previous work has shown that such a combination of NEAT and NSGA-II can be successful [25, 26].

NEAT is a direct encoding because every component in the genome directly corresponds to a component of the network phenotype. Therefore, 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 [6, 26, 29]. However, NEAT struggles when scaling up to domains with larger numbers of features, because a single structural mutation seldom significantly modifies the behavior of an agent, yet many such mutations are needed to optimize the behavior of a large network. Furthermore, NEAT networks have no way of leveraging information about the geometric organization of inputs if such information is available. These shortcomings of NEAT are addressed by HyperNEAT, described next.

### 3.3 Full-HyperNEAT

Hypercube-based NEAT [30] is an indirect encoding that extends NEAT by evolving networks that encode the connectivity patterns of typically larger *substrate* networks, which are evaluated in a given domain. Specifically, HyperNEAT genotypes are Compositional Pattern-Producing Networks (CPPNs [28]), which differ from standard NEAT networks in that their activation functions are not limited to a single type. Instead, CPPN activation functions come from a set of user-specified functions that produce useful patterns, such as symmetry and repetition. The specific activation functions used in this paper are sigmoid, Gaussian, sine, sawtooth wave, triangle wave, square wave, absolute value, and identity clamped to the range [0,1]. Because there are multiple activation functions, there is also a mutation operation that randomly replaces the activation function of a random node with another one from the set of possibilities.

However, the primary distinction that makes CPPNs and 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. Essentially, the CPPN describes how to encode the link weights as a function of geometry. These substrates are collections of layered neurons assigned explicit geometric locations with pre-determined potential connections between neurons in different layers. Each substrate defines its own coordinate frame, and multiple substrates can exist in a single network layer. The layout of substrates is defined by the experimenter and is domain-specific. It specifies how many substrate layers are needed, how many neurons are in each layer, what their activation functions are, 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 will exist, and if so, what its weight will be. Standard Full-HyperNEAT has the potential to create all links in a fully connected network. The specific inputs and outputs of a CPPN depend on how the experimenter defines the geometry of the substrate network. Several options are explored in the experiments described in Section 4, but for now, consider a straight-forward example in which two 2D substrates are being connected.

This CPPN has five inputs: x and y coordinates of both source and target neurons in different substrate layers, and a constant bias of 1.0. Input coordinates are scaled to the range $[-1, 1]$ with respect to the size of the current substrate. The CPPN has two outputs: one for the weights of links, and one for bias values within hidden neurons.

For each source and target neuron queried, the weight output defines the link weight. The bias output defines a constant bias associated with each neuron in a non-input substrate. Use of a separate bias output allows the magnitudes of bias values to be decoupled from link weight values. Bias values are derived by querying the CPPN with the x and y coordinates of the target neuron, while leaving the coordinates of the source neuron as $(0, 0)$. A more complicated example specific to Tetris is shown in Figure 2.

Though the approach described above allows Full-HyperNEAT to encode large neural networks with geometric domain awareness, it does not take advantage of modern architectures used in image processing tasks, such as convolutional neural networks (CNNs).

### 3.4 CNN-HyperNEAT

The new variant of HyperNEAT in this paper encodes CNNs in various ways, one of which is similar to previous work by Verbancsics and Harguess [38]. CNNs have achieved impressive performance in image processing tasks [21], including RL tasks [23].

CNN connectivity is sparser than in fully connected networks, but organized so that each hidden neuron focuses on a localized region of the preceding layer: its receptive field. Typical image processing applications split 2D color inputs into separate color channels to create a 3D input volume, though different input channels are used in this paper (Section 4.2). Regardless of how the input is shaped, higher layers are typically spread out across several feature maps.

CNN-HyperNEAT encodes CNNs by treating each feature map as a separate neural substrate. Every neuron in a convolutional layer connects to a $3 \times 3$ receptive field in each substrate of the preceding layer. The specific region that a neuron connects to corresponds to its location: the upper-left neuron in a hidden substrate connects to the upper-left $3 \times 3$ region in each substrate beneath it. As the target neuron shifts to an adjacent neuron, the receptive field slides across the inputs by a number of steps known as the *stride*, which in this paper is 1. Therefore, nearby hidden neurons connect to overlapping input regions. Additionally, because the receptive fields are confined to the inside of the source substrate (no zero padding is used), each subsequent layer of the network has substrates that are smaller than the preceding layer along each edge by two neurons.

Typically, neurons in the same feature map share the same weights, hence the name *feature map*: a certain configuration of link weights corresponds to a particular feature being searched for in the preceding layer. CNN-HyperNEAT does not enforce weight sharing. For a simple example connecting two substrates, link weights are instead

based on five CPPN inputs: x and y coordinates of the target neuron in a feature map substrate, a constant bias of 1, and x and y offsets from the center of the receptive field within the source substrate. The coordinates of the target neuron are scaled with respect to the size of the target substrate as before, but the receptive field offsets are scaled with respect to the size of the receptive field. Instead of treating the whole input substrate as a coordinate plane, this encoding treats each receptive field as a separate coordinate plane. A more complicated example specific to Tetris is shown in Figure 3.

CNN-HyperNEAT encodes a fewer potential links than Full-HyperNEAT using a different substrate geometry. The effectiveness of this approach is explored in the experiments below.

# 4 EXPERIMENTAL SETUP

This section describes how Tetris agents use neural networks to determine their actions, the low-level inputs used by these networks, the specific configurations 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 an evolved network as an afterstate evaluator. Before a piece begins to fall, a search algorithm considers every possible location the piece can be placed and remembers the sequence of moves leading to each placement. Placements that lead to immediate loss are not considered. Network inputs are taken from states that result from each valid placement. If a placement fills a row, that row will be cleared before the state's feature values are calculated. For each afterstate, the network produces a single output: a utility score between -1 and 1. The piece placement with the highest utility score across all possible placements is selected, and the agent carries out the remembered sequence of actions that lead to the desired placement. After placement, a new piece appears and the process repeats.

## 4.2 Low-Level Screen Inputs

All networks are evolved with screen inputs split into two sets. The first set identifies the locations of blocks, using 1 to represent the presence of a block and 0 the absence of a block. The second set identifies the locations of holes, which have a value of $-1$, while all other locations have a value of 0. Including the locations of holes made it easier to distinguish holes from empty non-hole locations.

Since there are two input channels from a $10 \times 20$ board, the number of inputs is 400. Each channel of 200 inputs occupied a separate input substrate within the networks encoded by HyperNEAT.

## 4.3 HyperNEAT Substrates

Several approaches to configuring the HyperNEAT substrates were used. For both Full-HyperNEAT and CNN-HyperNEAT, three architectures were evaluated. Links were encoded using either a threshold (THRESH) approach or LEO [39]. Two distinct coordinate geometries were also used: multi-spatial substrates (MSS [24]), and global coordinates. Each configuration choice is discussed in detail below.

*4.3.1 Architectures.* The simplest architecture has a depth and width of one (D1W1), meaning that there is a single hidden layer with of a single substrate. There are also direct links from the input substrates to the output neuron, bypassing the hidden layer. All neurons use the hyperbolic tangent (tanh) activation function.

For Full-HyperNEAT architectures, the size of each hidden substrate is $10 \times 20$, corresponding to the input size. Therefore, this first architecture is identical to the one previously used to apply HyperNEAT to Tetris [16]. For CNN-HyperNEAT, the hidden substrate is $8 \times 18$ because of how convolutional layers shrink with increasing layers. Although the hidden substrate is a convolutional layer, all links to the output neuron, from both the hidden and input substrates, were fully connected in all CNN-HyperNEAT architectures.

Note that tanh is also used for CNNs, despite the widespread popularity of rectified linear units (ReLU) for deep CNNs. One of the primary benefits of ReLU is that it combats the vanishing gradient problem [20] that arises when using SGD, but this problem is not an issue for evolutionary search. Nevertheless, preliminary experiments were conducted using ReLU, but no difference in performance was observed, so large batches of experiments were conducted with tanh to fairly compare with previous results using Full-HyperNEAT.

The next architecture has a depth of one, but a width of four (D1W4), i.e. four adjacent hidden substrates in one layer. In a CNN, these substrates correspond to feature maps. There are also bypass connections from the inputs to the output, as in D1W1.

The final architecture has a depth and width of four (D4W4). Unlike the previous architectures, no direct connections from the inputs to the output are allowed, because preliminary experiments indicated that there is a strong local optimum that relies only on bypass connections. Thus, the bypass option was removed to see if evolution could take advantage of a deep architecture.

For CNNs, the substrate sizes at each depth are $8 \times 18$, $6 \times 16$, $4 \times 14$, and $2 \times 12$. Since another subtraction by two would shrink the width to 0, four is the maximum CNN depth possible without using zero padding to prevent layers from shrinking. Depths between one and four were also used in preliminary experiments, but results indicated that the only meaningful distinction was between one layer and more than one. Therefore, experiments focus on the distinction between the deepest and shallowest possible architectures.

*4.3.2 Link Encoding.* In the original HyperNEAT, a single CPPN output determines both whether a link exists, and its weight. Specifically, for each source and target neuron queried, a link is only expressed if the magnitude of the weight output exceeds 0.2. If expressed, the link weight equals the output value scaled toward 0 to eliminate the region between $-0.2$ and 0.2. This approach is called the threshold (THRESH) approach in the results below.

An alternative approach is to use an additional link expression output (LEO [39]). In this paradigm, a link is present if the additional link expression output is greater than zero, and the weight of said link is simply the output of the corresponding weight output. The LEO approach allows network link structure to be decoupled from link weight magnitude, and thus allows for more modular networks.

*4.3.3 Coordinate Geometry.* For a given architecture, a means of specifying all link weights using a CPPN is necessary. The original HyperNEAT used global coordinates, which is also how Verbancsics and Harguess used HyperNEAT to encode CNNs [38]. In this approach, not only is each neuron situated within the coordinate frame of a substrate, but each substrate is situated within a coordinate frame containing the entire network.

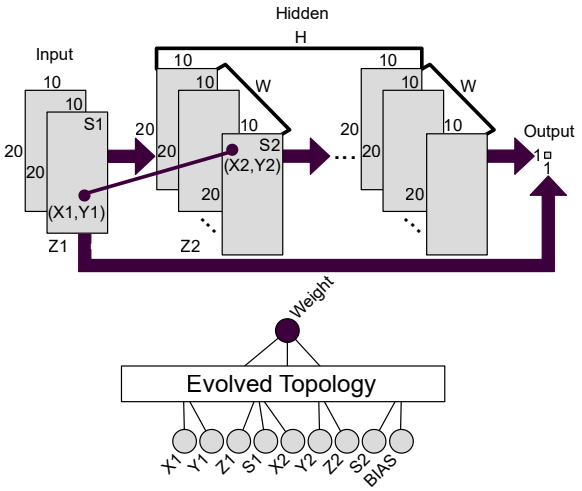



**Figure 2: Network Encoding with Full-HyperNEAT Using Global Coordinates. Tetris uses two input substrates (blocks and holes) and a single output neuron (utility). The number of hidden layers and substrates per layer vary by architecture, but each substrate in one layer has the potential to be fully connected to each substrate of the following layer. Using global coordinates, the locations of two neurons are provided as input to the CPPN (bottom) to produce a weight output (bias output and optional LEO output not shown). This one output neuron encodes all link weights in the substrate network, which allows for a more compact output layer, but imposes a geometric relationship between links across different substrates and layers. Full-HyperNEAT can also use multi-spatial substrates to define its link weights (not shown).**

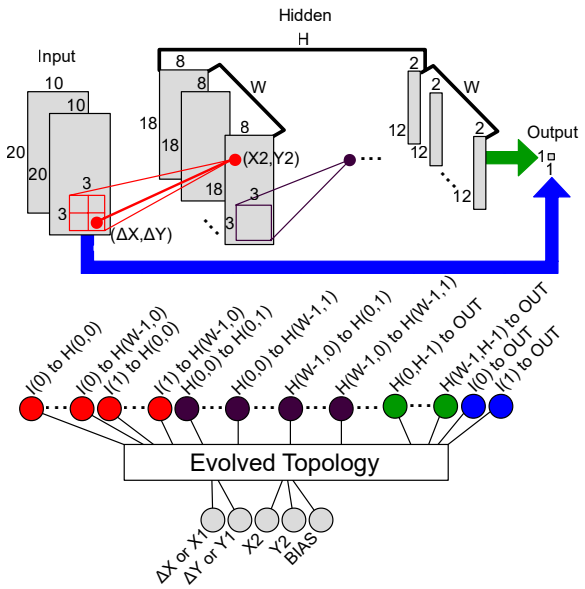



**Figure 3: Network Encoding with CNN-HyperNEAT Using Multi-Spatial Substrates. The same input and output substrates are present as in Figure 2, and different numbers of layers and substrates per layer are once again supported. However, substrates in each layer are smaller than in the layer preceding it because of how convolutional layers shrink. Connections into convolutional layers are depicted by the receptive field of a single neuron, but thick arrows indicate that layers are potentially fully connected. With multi-spatial substrates, every pair of connected substrates has its own CPPN output for defining link weights (additional bias outputs and optional LEO outputs not shown). Groups of output neurons are color coded by the pairs of layers they connect, though there are multiple output neurons per layer pair because layers can contain multiple substrates. Output labels indicate the source and target substrates, where $I(0)$ and $I(1)$ are the two input substrates, $H(S, Z)$ is a hidden substrate at depth $Z$ and layer position $S$, and $OUT$ is the output substrate containing one neuron. Because there are separate outputs for substrates at different global coordinates, the only necessary CPPN inputs are coordinates within the connected substrates for fully connected layers, and receptive field offsets and target neuron coordinates for convolutional layers. The output layers of the CPPNs are now crowded, but each output can encode distinct weight patterns unrelated to the geometry between layers. CNN-HyperNEAT can also use global coordinates to define its link weights (not shown).**

As a result, every neuron is uniquely identified by four coordinates: $(z, s, x, y)$. The z coordinate specifies the layer within the network, starting at 0 with the input layer and increasing upward to 1 at the output layer, with hidden layers evenly spaced between. The s coordinate specifies the substrate within a particular layer, and is scaled to $[-1, 1]$ from left to right. Finally, x and y are the coordinates within a particular substrate. The x coordinate is scaled to $[-1, 1]$, but the y coordinate is scaled to $[0, 1]$. The special scaling for y coordinates was favored because there is no geometric significance to the vertical center of the Tetris game board. Along the vertical axis, the most significant location is the floor, so coordinate values increase upward from there. A similar argument with respect to network structure applies to how the z coordinates are scaled.

Thus, when using global coordinates, CPPNs require nine inputs: $(z_1, s_1, x_1, y_1)$ and $(z_2, s_2, x_2, y_2)$ for source and target neuron locations, and a constant bias value of 1. When using THRESH encoding, the CPPN has two outputs: one for the link weight, and another for the neuron bias. When LEO is used, there is an additional link expression output. Figure 2 shows how global coordinates are used to encode a network in Full-HyperNEAT. CNN-HyperNEAT can also use global coordinates, though some slight adjustments must be made, which are described at the end of this section below.

An alternative to global coordinates is the multi-spatial substrate (MSS [24]) approach. This approach removes unnecessary geometric constraints from the structure of the network. Specifically, global coordinates impose a geometric relation between substrates based on location within the network, which is completely independent from any geometric information in the domain. There is no a priori reason to expect useful information to exist within the structure of a human-designed network. Therefore, MSS only uses geometric information from the domain, and not from the network structure.

Specifically, MSS treats the weights connecting every pair of substrates as if they exist within unrelated coordinate frames by evolving CPPNs with separate outputs corresponding to each pair of connected layers. For every two substrates that are connected, there exists a weight output and, when using LEO, a link expression output. Additionally, every non-input substrate also has its own output responsible for defining the bias values of its neurons. The number of outputs in a CPPN can thus be very large, but the use of these separate outputs assures that the weight patterns connecting different layers need not be bound by any geometric relationship.

However, MSS still leverages geometric information from the domain, because the neuron coordinates within each substrate are still used as CPPN inputs. Specifically, five inputs are used: $(x_1, y_1)$

for the source neuron in some substrate, $(x_2, y_2)$ for the target neuron within a target substrate, and a constant bias of 1. As with global coordinates, x and y coordinates are scaled to $[-1, 1]$ and $[0, 1]$ respectively. When deriving link weights between different pairs of layers, the same inputs may occur, but different CPPN outputs will be used, thus allowing different weights to be defined. MSS was used in the original work applying Full-HyperNEAT to Tetris [16]. Figure 3 shows how MSS coordinates are applied in CNN-HyperNEAT.

Whether global or MSS coordinates are used, coordinate geometry is slightly different with CNN-HyperNEAT. Although CNNs use convolutional layers, substrates that connect to the output neuron are fully connected, and use the same substrate geometry described above. However, when the target substrate is within a convolutional layer, $x_1$ and $y_1$ inputs are replaced with $\Delta x$ and $\Delta y$ inputs, which are offsets from the center of the current receptive field within the source substrate. Both $\Delta x$ and $\Delta y$ are scaled to the range $[-1, 1]$.

These various configurations allow for a variety of different experimental conditions to be compared, but several parameter settings remained constant throughout all experiments.

### 4.4 Evaluation Setup

Most experiments consisted of 30 runs per approach lasting 500 generations with a population size of $\mu = \lambda = 50$, but there were only 10 runs per approach when using the D4W4 architecture due to the excessive time required to evaluate larger networks.

Tetris is a noisy domain, meaning that repeated evaluations can yield wildly different scores due to randomness in the sequence of tetrominoes in each game. To mitigate the noisiness, fitness scores were averaged across three trials. The specific objectives used by NSGA-II for selection were the game score and the number of time steps the agent survived. The time steps objective is particularly useful early in evolution when agents cannot clear any rows at all.

When creating the next generation of CPPNs, the chance of creating new offspring using crossover was 50%. Each link in an offspring network had a 5% chance of Gaussian perturbation. There was a 40% chance of adding a new link, a 20% chance of adding a new node, and a 30% chance of a randomly chosen node having its activation swapped with another random function from the available set (Section 3.3). These settings led to the results discussed next.

## 5 RESULTS

Collectively, the results show that CNN-HyperNEAT is superior to Full-HyperNEAT, MSS is superior to global coordinates, there is no clear winner between THRESH and LEO, and shallow architectures are easier to evolve than deep architectures. Figure 4 shows game scores during evolution with methods grouped by architecture, and also shows the distribution of scores in the final generation.

Unsurprisingly, a Kruskal-Wallis test indicates that there are significant differences between game scores in the final generation across the 24 methods ($H = 302.69, df = 23, p \approx 2.2 \times 10^{-16}$), but of greater interest is how specific pairs of methods compare. Pairwise Mann-Whitney-Wilcoxon tests are used to compare all methods using Bonferroni error correction to control the familywise error rate. The $p$ values reported are adjusted to account for the correction.

CNN-HyperNEAT with MSS coordinates and D1W1 or D1W4 produces the best results. There is no significant difference between these two architectures with THRESH or LEO ($p \approx 1.0$), though median scores with LEO are slightly lower with both architectures.

These four conditions are better than their counterparts using global coordinates ($p < 0.05$). Statistically, the shallow CNN approaches using global coordinates are lumped in with all of the poor performing fully connected shallow networks ($p > 0.05$), though these CNN approaches have outliers that surpass the median scores of the CNN approaches with MSS coordinates (Figure 4d).

The best scores achieved by fully connected networks with each architecture are between 200 and 300. Specifically, D1W1 Full MSS THRESH reproduces previous HyperNEAT results in Tetris [16] with a median score of 264.33. However, all shallow CNNs surpass these scores by an order of magnitude.

Deep CNNs do not perform as well. In fact, all D4W4 methods are statistically tied ($p > 0.05$). However, the best D4W4 scores come from CNNs, although Full MSS THRESH does nearly as well as CNNs with MSS. Other interesting observations with D4W4 are an outlier from CNN Global THRESH that scores just below 1,400, and surprisingly bad performance by CNN Global LEO.

The best evolved champions exhibit impressive behavior, being able to recover multiple times from screens nearly filled with blocks: https://people.southwestern.edu/~schrum2/re/tetris-cnn.php. A shallow CNN playing for nearly an hour eventually loses with a score of 2,901. The best score in the final generation of evolution was 6,640.

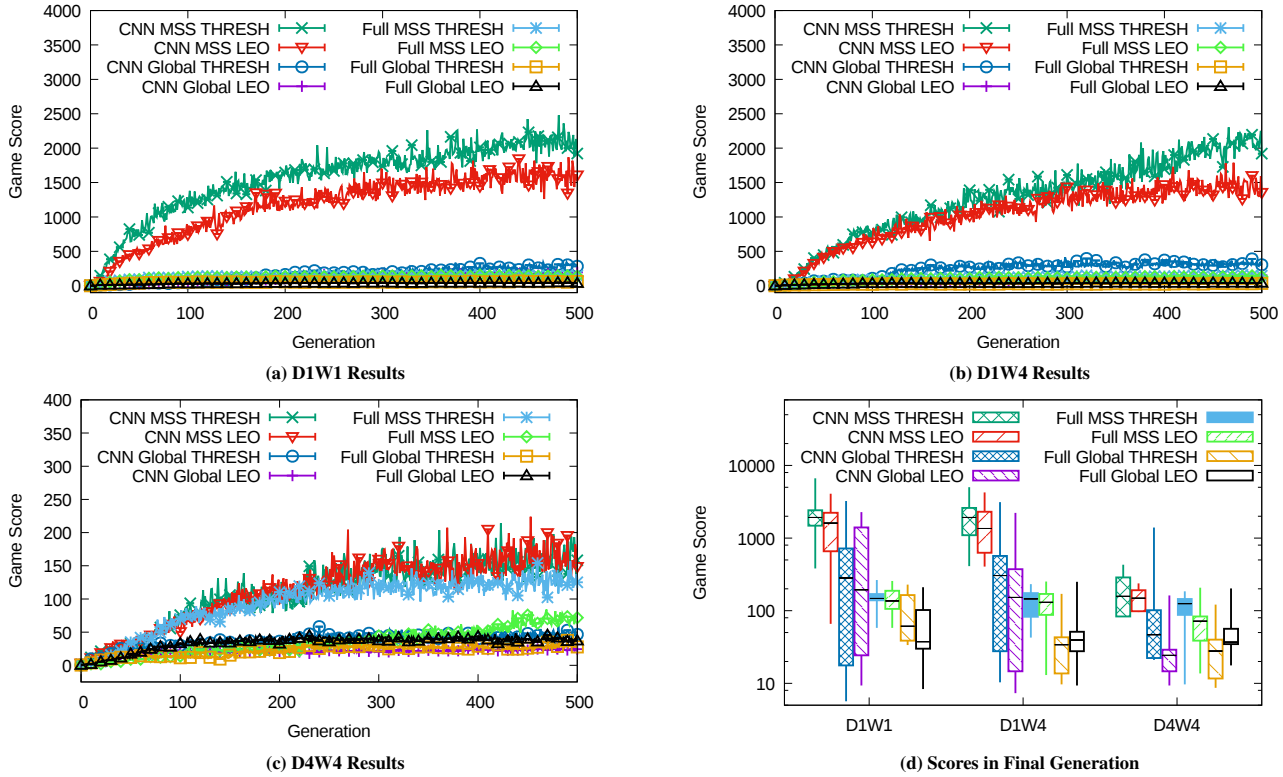
As good as these results are, higher scores have been achieved in the past with high-level features, and can hopefully be achieved in the future with low-level features. Further analysis of these results and ideas for future improvement are discussed next.

## 6 DISCUSSION AND FUTURE WORK

The poor performance of deep architectures is disappointing. HyperNEAT can compactly encode large, deep networks, but has trouble evolving effective deep networks for Tetris. Part of the challenge for HyperNEAT could be its reliance on geometric coordinates to define link weights. In the case of global substrate coordinates, there is no reason to expect there to be a geometric relationship between different convolutional filters. This lack of a relationship is why MSS is superior to global coordinates, but when defining deep architectures, the number of additional CPPN outputs required to apply MSS is so high that it is difficult for evolution to fine tune them all.

An indirect encoding approach similar to HyperNEAT is HyperNetworks [18]. Like a CPPN, a HyperNetwork is queried to generate the weights of another network, but its inputs are not geometric coordinates. Rather, the input is an embedding vector in a latent space, that is itself learned during HyperNetwork training via backpropagation. Importantly, this approach can generate effective convolutional networks for visual tasks. The authors point out that while using CPPNs to generate convolutional filters (as done in [13]) results in nice, regular patterns in the weight visualization, effective convolutional filters are often quite messy and irregular.

However, CNN-HyperNEAT was successful at evolving shallow convolutional networks to play Tetris. Though these networks were not deep, they were still fairly large, consisting of many parameters: 3,281 for D1W1, and 11,921 for D1W4. When using Full-HyperNEAT, networks had even more parameters, though these networks did not fare as well. Any fully connected architecture can




(a) D1W1 Results

(b) D1W4 Results

(c) D4W4 Results

(d) Scores in Final Generation

**Figure 4: Median Game Scores. Performance of different runs of the same method does not conform to a normal distribution, so median game scores across 30 runs are shown by generation for the (a) D1W1 and (b) D1W4 architectures. Performance of each method using these architectures is similar: CNNs using MSS coordinates are much better than other methods. For the (c) D4W4 architecture, median scores across 10 runs are shown. The best median scores with D4W4 are an order of magnitude lower than those with the shallow architectures. Within this group, CNNs with MSS coordinates are tied with Full MSS THRESH for best performance. (d) Because different methods attain scores at different orders of magnitude, the distributions of scores in the final generation are shown on a logarithmic scale using boxplots, which depict minimum, first quartile, median, third quartile, and maximum scores. Although shallow CNNs using global coordinates have low median scores, the best runs of these approaches exceed the median scores of CNNs with MSS coordinates. Additionally, among the D4W4 results, the best outlier run of CNN Global THRESH surpasses all other D4W4 scores. Ultimately, shallow CNNs with MSS coordinates perform the best, but other CNN approaches also produce high scores.**

be pruned back to a convolutional architecture, but evolution had trouble doing so, hence the superiority of CNNs.

The best CNN results used multi-spatial substrates. Previous work evolving CNNs with HyperNEAT [38] only used global coordinates. These architectures were deep, but not particularly successful with evolution alone. In fact, the authors indicate that evolving CNNs rather than fully connected networks did not result in increased performance. Their evolved CNNs were only better than fully connected networks when used as feature extractors, which had networks trained using SGD on top of them. In contrast, the successful CNN results in Tetris relied on evolution alone.

The use of LEO vs. threshold link encoding had the smallest effect. Threshold encoding tends to produce scores slightly higher than LEO's, but this difference is not statistically significant. LEO is meant to encourage modularity via intelligent pruning of links, but CNNs are already comparatively sparse and modular, which is probably why LEO offers no benefit.

Although CNN-HyperNEAT can encode deep architectures, the shape of the search space apparently makes it difficult to find effective weight configurations in this space. However, recent results from Uber AI [32] demonstrate that a fairly simple direct encoding can evolve effective weight values for large fixed-architecture CNNs with millions of parameters. Uber AI's results made use of massive parallelization at a scale not available for the experiments in this paper. In fact, the outliers in Tetris could mean that the small population size of 50 may have stranded certain populations within effectively inescapable regions of low fitness. In contrast, Uber AI used population sizes in the thousands, which was only practical due to the availability of massive parallelization. Some of Uber AI's experiments [8, 32] also made use of Novelty Search [22], a technique especially tailored to escaping local optima by using a novelty score for selection in place of a standard fitness function.

However, population sizes and selection mechanism are likely not the only reasons that deep networks performed poorly. As mentioned above, HyperNEAT has a bias toward regularity, which has pros and cons. However, an extension to HyperNEAT that allows more

irregularity is the Hybridized Indirect and Direct encoding (HybrID [7]), which begins evolution using HyperNEAT, and then evolves only the directly-encoded substrate networks further after a fixed number of generations. There are also improvements to HybrID that automatically determine the switch point, and combine indirectly-encoding CPPNs with directly-encoded weight offsets to combine regularity and irregularity in the encoded substrate networks [19].

Evolution of deep networks can also be combined with backpropagation to fine tune evolved networks, but the additional computational burden is restrictive. However, the use of backpropagation in Tetris is not straightforward. Deep RL methods have not been applied to the full game of Tetris yet, but work in SZ-Tetris using low-level features required special tricks to succeed [10, 12]. Specifically, a special shaping reward function based on holes rather than the game score was used, new activation functions with special properties were introduced, and softmax action selection was used to make TD($\lambda$) effective. In short, previous success in SZ-Tetris was not a simple matter of applying some well known deep RL method with minimal modification. Future success in the full game of Tetris with deep RL could require additional tricks.

Regardless, the results in this paper show that evolution can successfully evolve shallow convolutional networks, and also point toward some potential areas of exploration that could lead to successful deep CNNs in the future. CNN-HyperNEAT could succeed in other video games and visual RL tasks as well.

## 7 CONCLUSION

CNN-HyperNEAT can evolve convolutional neural networks using an indirect encoding. This approach successfully evolved afterstate evaluators for the game of Tetris with shallow architectures using multi-spatial substrates, rather than global substrate coordinates. The CNN-HyperNEAT approach can be applied to other domains that use visuospatial inputs, such as video games. Further development, via combination with Novelty Search, HybrID, and/or backpropagation could scale CNN-HyperNEAT up to deeper networks.

## 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] Niko Böhm, Gabriella Kókai, and Stefan Mandl. 2004. Evolving a Heuristic Function for the Game of Tetris. In *Lernen - Wissensentdeckung - Adaptivität*.

[3] A. Boumaza. 2009. On the Evolution of Artificial Tetris Players. In *Computational Intelligence and Games*. 387–393.

[4] R. Breukelaar, E. D. Demaine, S. Hohenberger, H. J. Hoogeboom, W. A. Kosters, and D. Liben-Nowell. 2004. Tetris is hard, even to approximate. *International Journal of Computational Geometry and Applications* 14, 1–2 (2004), 41–68.

[5] Heidi Burgiel. 1997. How to Lose at Tetris. *Mathematical Gazette* 81, 491 (1997).

[6] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. 2009. Evolving Competitive Car Controllers for Racing Games with Neuroevolution. In *Genetic and Evolutionary Computation Conference*. 1179–1186.

[7] 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.

[8] Edoardo Conti, Vashisht Madhavan, Felipe Petroski Such, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. 2017. Improving Exploration in Evolution Strategies for Deep Reinforcement Learning via a Population of Novelty-Seeking Agents. *ArXiv e-prints* (2017). arXiv:1712.06560

[9] 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.

[10] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. 2016. From free energy to expected energy: Improving energy-based value function approximation in reinforcement learning. *Neural Networks* 84 (2016), 17–27.

[11] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. 2017. Online Meta-learning by Parallel Algorithm Competition. *ArXiv e-prints* (2017). arXiv:1702.07490

[12] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. 2017. Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning. *ArXiv e-prints* (2017). arXiv:1702.03118

[13] C. Fernando, D. Banarse, M. Reynolds, F. Besse, D. Pfau, M. Jaderberg, M. Lanctot, and D. Wierstra. 2016. Convolution by Evolution: Differentiable Pattern Producing Networks. In *Genetic and Evolutionary Computation Conference*.

[14] Jose M. Font, Daniel Manrique, Sergio Larrodera, and Pablo Ramos Criado. 2017. Towards a Hybrid Neural and Evolutionary Heuristic Approach for Playing Tile-matching Puzzle Games. In *Computational Intelligence and Games*.

[15] Victor Gabillon, Mohammad Ghavamzadeh, and Bruno Scherrer. 2013. Approximate Dynamic Programming Finally Performs Well in the Game of Tetris. In *Neural Information Processing Systems*. 1754–1762.

[16] 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 *Genetic and Evolutionary Computation Conference*.

[17] 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.

[18] David Ha, Andrew Dai, and Quoc V. Le. 2017. HyperNetworks. In *International Conference on Learning Representations*.

[19] Lucas Helms and Jeff Clune. 2017. Improving HybrID: How to best combine indirect and direct encoding in evolutionary algorithms. *PLOS ONE* 12, 3 (2017).

[20] Sepp Hochreiter. 1991. *Untersuchungen zu dynamischen neuronalen Netzen*. Ph.D. Dissertation. TU Munich.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Neural Information Processing Systems*.

[22] Joel Lehman and Kenneth O. Stanley. 2008. Exploiting Open-Endedness to Solve Problems Through the Search for Novelty. In *Artificial Life*. MIT Press.

[23] 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*.

[24] Justin K. Pugh and Kenneth O. Stanley. 2013. Evolving Multimodal Controllers with HyperNEAT. In *Genetic and Evolutionary Computation Conference*.

[25] Jacob Schrum and Risto Miikkulainen. 2012. Evolving Multimodal Networks for Multitask Games. *TCIAIG* 4, 2 (2012), 94–111.

[26] 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.

[27] Özgür Simsek, Simon Algorta, and Amit Kothiyal. 2016. Why Most Decisions Are Easy in Tetris - And Perhaps in Other Sequential Decision Problems, As Well. In *International Conference on Machine Learning*. 1757–1765.

[28] Kenneth O. Stanley. 2007. Compositional Pattern Producing Networks: A Novel Abstraction of Development. *Genetic Programming and Evolvable Machines* 8, 2 (2007), 131–162.

[29] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. 2005. Evolving Neural Network Agents in the NERO Video Game. In *Computational Intelligence and Games*.

[30] Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. 2009. A Hypercube-based Encoding for Evolving Large-scale Neural Networks. *Artificial Life* (2009).

[31] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation* 10 (2002), 99–127.

[32] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. 2017. Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. *ArXiv e-prints* (2017). arXiv:1712.06567

[33] Istvan Szita and András Lörincz. 2006. Learning Tetris Using the Noisy Cross-Entropy Method. *Neural Computation* 18, 12 (2006), 2936–2941.

[34] István Szita and Csaba Szepesvári. 2010. SZ-Tetris as a benchmark for studying key problems of reinforcement learning. In *ICML workshop on ML and games*.

[35] Brian Tanner and Adam White. 2009. RL-Glue : Language-Independent Software for Reinforcement-Learning Experiments. *Journal of Machine Learning Research* 10 (2009), 2133–2136.

[36] Christophe Thiery and Bruno Scherrer. 2009. Building Controllers for Tetris. *International Computer Games Association Journal* 32 (2009), 3–11.

[37] Christophe Thiery and Bruno Scherrer. 2009. Improvements on Learning Tetris with Cross Entropy. *International Computer Games Association Journal* (2009).

[38] P. Verbancsics and J. Harguess. 2015. Image Classification Using Generative Neuro Evolution for Deep Learning. In *Winter Conference on Applications of Computer Vision*. 488–493. https://doi.org/10.1109/WACV.2015.71

[39] Phillip Verbancsics and Kenneth O. Stanley. 2011. Constraining Connectivity to Encourage Modularity in HyperNEAT. In *Genetic and Evolutionary Computation Conference*.

[40] Shimon Whiteson, Brian Tanner, and Adam White. 2010. The Reinforcement Learning Competitions. *AI Magazine* 31, 2 (2010), 81–94.