In Genetic and Evolutionary Computation Conference (GECCO 2012),
Philadelphia, Pennsylvania, July 2012.

HyperNEAT-GGP: A HyperNEAT-based Atari General Game Player

Matthew Hausknecht, Piyush Khandelwal, Risto Miikkulainen, Peter Stone
Department of Computer Science
University of Texas at Austin
{mhauskn,piyushk,risto,pstone}@cs.utexas.edu

ABSTRACT

This paper considers the challenge of enabling agents to learn with as little domain-specific knowledge as possible. The main contribution is HyperNEAT-GGP, a HyperNEAT-based General Game Playing approach to Atari games. By leveraging the geometric regularities present in the Atari game screen, HyperNEAT effectively evolves policies for playing two different Atari games, Asterix and Freeway. Results show that HyperNEAT-GGP outperforms existing benchmarks on these games. HyperNEAT-GGP represents a step towards the ambitious goal of creating an agent capable of learning and seamlessly transitioning between many different tasks.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning, Intelligent Agents

General Terms

Algorithms, Experimentation, Performance

Keywords

HyperNEAT, General Game Playing, Atari, Learning Agents, Neuroevolution

1. INTRODUCTION

A major challenge for AI is to develop agents that can learn and perform many different tasks. To this end, this paper aims at developing a learning agent capable of playing a large number of games with as little domain specific knowledge as possible. Famous game playing AI systems such as Deep Blue for chess [1], Watson for Jeopardy [6], and TD-Gammon for backgammon [17] all demonstrate that with enough manpower and ingenuity it is possible to tackle AI challenges that may have previously seemed insurmountable. Unlike these game intelligences which were created and tuned specifically for a single task, the game playing agent described herein must be general enough to tackle many different Atari 2600 games. This requires general intelligence to be built into the agent itself rather than just imparted by the programmer of the agent in the form of clever single-purpose algorithms.

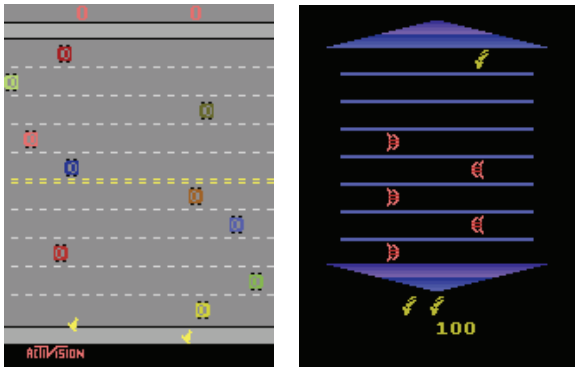


Figure 1: Freeway and Asterix, two of the many games available for the Atari 2600.

This work focuses on learning to play Atari 2600 games, a middle ground between classic board games and newer, graphically intensive video games. The Atari 2600 includes many different games, including complex ones such as chess and checkers, yet lacks the complex 3-D graphics of newer video games. Like traditional board games, the Atari provides opportunities for agents to benefit from a solid understanding of the game's dynamics and allows for careful planning while at the same time incorporating simple visual representations that can be processed and interpreted. Dynamics of Atari games vary wildly from Checkers to Space Invaders, necessitating the use of general learning algorithms.

Despite the variability of game dynamics, all Atari games share a standard interface designed for humans to interact with and enjoy. Game state is conveyed to the player through a 2D game screen, and in response, the player controls game elements by manipulating a joystick and pressing a single button. This standard interface, combined with the large number of available games, makes Atari a convenient platform for AI researchers.

This paper presents HyperNEAT-GGP, an agent which uses an evolutionary algorithm called Hypercube-based NeuroEvolution of Augmenting Topologies (HyperNEAT) [7]. Unlike most other approaches, HyperNEAT is capable of exploiting geometric regularities present in the 2D game screen in order to evolve highly effective game playing poli-

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'12, July 7–11, 2012, Philadelphia, Pennsylvania, USA.
Copyright 2012 ACM 978-1-4503-1177-9/12/07 ...$10.00.

cies. This paper applies HyperNEAT-GGP to the two Atari games shown in Figure 1, Freeway and Asterix.

The next section contains related work. Section 3 discusses the merits of the Atari 2600 console as a research platform and describes the two games used in this paper. Section 4 covers the basics of HyperNEAT and how it is able to take advantage of game geometry. Next, Section 5 presents our visual processing architecture and parallel computing framework. Experiments are presented in Section 6, followed by future work and conclusions.

2. RELATED WORK

The study of general intelligence through the development of general game playing agents is not unique to this paper. Organizers in the field of General Game Playing (GGP) hold annual competitions for general game playing agents [8]. These agents are typically given a declarative description of an arbitrary game including a complete description of the game dynamics. They have no prior knowledge of this description and must formulate strategies to play this game on the fly. Unlike specialized game players, general game playing agents cannot rely on algorithms designed in advance for specific games. Successful agents typically incorporate artificial intelligence technologies such as knowledge representation, reasoning, learning, rational decision making and automatic theorem proving.

While motivations are similar, the work reported in this paper differs from GGP in that Atari games are not formalized in such an abstract representation. Atari games convey state through a visual representation, and the dynamics of the game must first be learned before strategies can be formulated. Additionally, HyperNEAT-GGP currently only considers single player games as opposed to GGP players who compete against each other.

Another class of game playing agents is found at the annual Ms. Pac-Man competition. Like HyperNEAT-GGP, Ms. Pac-Man agents utilize actual screen representations [9] and must deal with non-determinism introduced by delays in processing the game screen. Successful entries in this competition have now far exceeded novice human players [10]. HyperNEAT-GGP uses an approach similar to the Ms. Pac-Man agents in that both agents extract objects from the game screen. However, the object detection and game playing machinery used by HyperNEAT-GGP must be generally applicable enough to handle multiple games rather than specialized toward a single game such as Ms. Pac-Man.

In 3D environments, backpropagation and Lamarckian neuroevolution were used to train a neural network visual controller for agents in the Quake II environment [12]. In order to process the 3D game screen, the agents used an artificial retina consisting of 28 grayscale blocks arranged into a grid spanning the width of the game screen. Like the human retina, more blocks were clustered near the center of the visual field to allow the agent to better recognize if the target was directly ahead. The resulting agent was able to navigate a room with a large central pillar, seek out an enemy, and shoot it. However, how these techniques could be used to implement general game playing was not addressed.

To the best of the authors' knowledge, the first work on Atari game playing was an R-Max learning agent which employed an Object-oriented MDP representation [4]. Objects were detected in the game screen of the popular Atari game, *Pitfall*. The agent was able to make it past the first screen. Subsequent work on learning in Atari games includes that of Naddaf [11]. Naddaf modified the popular Atari 2600 emulator, Stella, in order to allow it to be easily controlled by computer programs such as learning agents. Naddaf quantified the performance of several reinforcement learning and search agents over 50 different Atari games. Reinforcement learning agents included a gradient descent Sarsa(λ) agent with linear function approximation, capable of learning from feature vectors generated from either the game screen or the console RAM. Search tree agents include full tree search and UCT-based agents. Also motivated by the desire to create general game playing agents, Naddaf compiled experimental results of RL and search tree agents in over 50 Atari games. More than just a experimental benchmark, this work served as the inspiration for much of the visual processing described in Section 5.

Learning to play games based on overhead representations has been previously attempted by Verbancsics and Stanley [18] who focused on the RoboCup Keepaway Soccer domain [14]. In this domain a number of *keeper* agents must maneuver and pass a soccer ball so that it is not captured by one of the *taker* agents. Verbancsics and Stanley encode the state of the game using an overhead representation of the objects on the playing field – namely the keepers, takers, and the ball. HyperNEAT is used to exploit the geometric regularities present in this overhead representation of the field. The learned policy is competitive with top learning algorithms for this task [15]. Additionally, the learned policy can be effectively transferred with no further learning to the same task at a higher resolution or a different number of players on the field. This successful transfer is a result of the indirect encoding of HyperNEAT. HyperNEAT has also been successfully applied to other domains such as checkers [7], multi-agent predator prey [3], and quadruped locomotion [2].

While HyperNEAT-GGP is quite similar to Verbancsics and Stanley's approach, in many ways Atari games represent a more challenging learning target than RoboCup Keepaway. In Keepaway there are a fixed number of object classes such as takers, keepers, and the ball. On the other hand, Atari games may contain an arbitrary number of object classes that interact with each other in unexpected ways, requiring HyperNEAT-GGP to be more general. Additionally, the dynamics in any given Atari game are highly variable, ranging from simple games in which the agent must reach the goal while avoiding cars to highly complex games in which the agent must shoot fish while attempting to rescue five swimmers, all before the oxygen in the player's submarine is depleted. Though this paper only demonstrates results on two of the many Atari games, the successful results represent an important step towards the long-term goal of testing the approach on, and refining it to generalize to, the full range of Atari games.

3. ATARI FOR RESEARCH

The Atari 2600 video game console was released in October 1977. It was the first console to create game cartridges that decoupled game code from console hardware (previous devices all contained dedicated hardware with games already built in). Selling over 30 million consoles [5], Atari was considered wildly successful as an entertainment device. Today, while Atari is no longer at the forefront of entertainment, the console has good research potential for three reasons:

First, the Atari console has a large collection of games. These games vary greatly from board games such as chess to action-exploration games like Pitfall to shooting games such as Asteroids and Space Invaders. Many games have support for a second player, opening the possibility of multi-agent learning. Having such a large number of games allows AI researchers to develop a single learning agent and then quickly and easily apply it to a large set of domains, facilitating the development of general game playing agents.

Second, a number of open-source Atari emulators exist, including projects such as Atari Learning Environment (ALE)[1] that are designed specifically to accommodate learning agents. Furthermore, since the Atari 2600 CPU ran at 1.19 megahertz, modern emulators can run at high speeds of up to 2000 frames per second, expediting the evaluation of agents and algorithms.

Third, the Atari state and action interface is simple enough for learning agents, but complex enough to control many different games. The state of an Atari game can be described relatively simply by its 2D graphics (containing between 8 and 256 colors depending on the color mode and a native resolution of 160×210), elementary sound effects, and 128 bytes of console RAM. The discrete action space for Atari consists of eight directions of movement for the joystick (up, down, left, right, up&left, up&right, etc) as well as a single button. This button can be pressed alone or simultaneously with any of the joystick movements. Including *NO-OP* (no action), this yields a total of 18 possible actions.

This work evaluates two Atari 2600 games – Freeway and Asterix. The objectives and controls of these games are as follows: In Freeway the player controls a chicken as it crosses a ten lane highway filled with traffic in an effort to "get to the other side." The chicken is allowed to move only up, down, or remain in place. Colliding with a car results in the chicken being thrown a distance towards the bottom of the screen. Each time a chicken reaches the top of the screen, the player is rewarded a point and the chicken respawns at the bottom of the screen. Gameplay continues for two minutes and sixteen seconds.

In Asterix the player controls a unit called Asterix in his quest to collect as many objects as possible while avoiding deadly lyres. Asterix can move in any of the four cardinal directions and receives 50 score whenever he collects a magic potion. Asterix loses a life when he touches a lyre and when Asterix has exhausted his supply of lives the game ends. Lyres and collectible objects spawn along the edges of the screen and move horizontally. Objects gain speed as the game progresses, necessitating quick reflexes and decisions.

Freeway and Asterix were selected as representative Atari 2600 games because their game dynamics were sufficiently different from each other, and both have been studied in the past [11].

Having motivated the Atari as a suitable research platform for the development of general game playing agents, the next challenge is developing a capable HyperNEAT-based learning agent.

4. HYPERNEAT

This section reviews the fundamentals of the HyperNEAT learning algorithm. HyperNEAT is an extension of the Neuro Evolution of Augmenting Topologies (NEAT) algorithm [13].

[1]http://yavar.naddaf.name/ale/

NEAT evolves the topology and weights of an Artificial Neural Network (ANN) that is applied directly to the problem of interest. In contrast, HyperNEAT, introduced by Gauci and Stanley [7], evolves an *indirect encoding* called a Compositional Pattern Producing Network (CPPN). The CPPN is then used to define the weights of an ANN that produces a solution for the problem. Furthermore, because the CPPN is aware of domain geometry, the ANN it encodes implicitly contains knowledge about geometric relationships present in a given domain. In comparison to standard NEAT, HyperNEAT's encoding allows it to take advantage of geometric regularities present in many board and 2D games, such as those in Atari 2600.

Specifically, HyperNEAT works in four stages:

1. The weights and topology of the CPPN are evolved. Internally a CPPN consists of functions such as Gaussians and sinusoids connected in a weighted topology (see Figure 2).
2. The CPPN is used to determine the weights for every pair of *(input,output)* nodes in a fully connected ANN.
3. With fully specified weights, the ANN is applied to the problem of interest. The performance of the ANN determines the fitness of the CPPN that generates it.
4. Based on fitness scores, the population of CPPNs is maintained, evaluated, and evolved via NEAT.

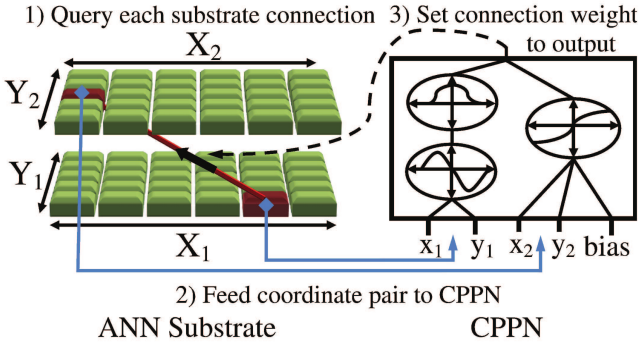


Figure 2: HyperNEAT evolves the weights and topology of a CPPN (right). This CPPN is subsequently used to determine all of the weights between substrate nodes in the ANN (left). Finally, the ANN is used to compute the solution to the desired problem. CPPNs are said to be geometrically aware because when they compute the weights of the associated ANN, they are given as input the x,y location of both the input and output node in the ANN. Figure duplicated from [18].

The ANN's input representation will be discussed further in Section 5.3. For more information on HyperNEAT, refer to [7].

5. APPROACH

This section describes the components of HyperNEAT-GGP. The main points are the manner in which the raw Atari game screen is processed, the self-agent is identified, and HyperNEAT is interfaced with the detected objects.

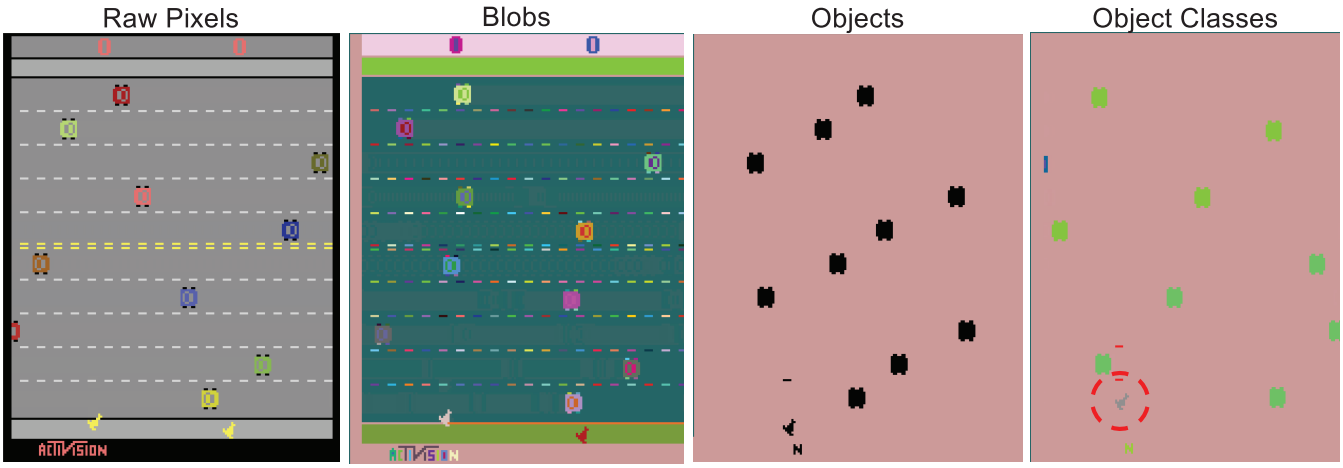


Figure 3: Visual Processing Architecture applied to the game Freeway. Raw pixels from the game screen are displayed on the left. Next, contiguous pixels of the same color are merged into blobs. Objects are then extracted by merging adjacent blobs which exhibit constant velocity over the last two frames. Next, objects are clustered into object classes based on a pixel similarity score. Three main object classes are found – cars facing left, cars facing right, and the chicken. This approach is similar to that of Naddaf [11]. Finally, self detection successfully identifies the chicken blob as the agent and colors it gray (circled in a red dashed line in rightmost screen).

5.1 Visual Processing

For nearly any machine learning problem, the question of how to encode the state space is of great importance. Similar to Verbancsics and Stanley's example, HyperNEAT-GGP uses an overhead object representation of the current game screen. Since the Atari provides only the raw pixels of the screen as input, a visual processing stack identifies objects and game entities without a priori knowledge of a specific game [11]. A graphical depiction of this stack is shown in Figure 3. While it is likely possible to learn from the raw screen's pixels, object detection requires little work and reduces the complexity of the learning task by eliminating pixels not directly relevant to playing the game.

Visual processing begins at the raw pixels of the game screen. Image segmentation groups adjacent raw pixels with similar colors into blobs. Next, blob merging occurs, outputting a set of current objects on screen. This process examines all of the recently discovered blobs and compares them with equivalent blobs in the last frame in order to compute a velocity for each blob. Blobs are matched between screens using pixel similarity. Velocity is computed by measuring the displacement of blob centroids. Once each blob is assigned a velocity, adjacent blobs with the same non-zero velocity are merged into objects. Objects that are too small or become stationary are thrown out. This check helps reduce the number of false positives in the object-detection process.

Finally, objects are clustered into object classes or prototypes. Specifically, the shape of each pair of objects is compared and if found to exceed a similarity threshold (97% pixel match in these experiments), the objects are grouped into the same class. Pixel similarity between two objects is computed by comparing the presence of pixels relative to each object's bounding box. As Figure 3 indicates, different object classes are discovered for the cars at the bottom half of the screen, cars at the top half of the screen, and the chicken.

To reduce the number of spurious prototypes, prototypes are passed to the next stage of the approach only when a number of instances of that prototype are seen within successive frames. In the example in Figure 3, this check helps remove prototypes for objects created when the cars are at the edges of the screen. Prototypes failing this check are removed while those which pass are assigned a unique real number (see Section 5.3 for more details).

Objects are assigned to the same class if their shape is relatively similar without taking color into consideration. This assumption has a potential drawback in certain games if different objects have similar shapes but different colors.

5.2 Self-Identification

The self identification step is meant to identify the location of an on-screen entity that is being controlled by the agent. In the vast majority of Atari games, the player's actions affect the movement of some on-screen entity, here termed the *self*. Knowledge of the location of the *self* is crucial to selecting an action, as described below. HyperNEAT-GGP uses an approach based on information gain to identify a blob most likely to correspond to the *self*. Pseudocode is given in Algorithm 1.

At the high level, this approach makes certain assumptions about the self entity. First, it is assumed that the self blob will move similarly whenever the same action is performed. That is, whenever an action, say Joystick Up, is taken, the resulting velocity of the self blob should have a similar value (e.g. blob.y_velocity = -1).

As input, in lines 1-3, the algorithm has access to the set of possible joystick and button actions applicable to the current game (this is typically a subset of the 18 possible actions present on the Atari console), a list of blobs detected in the current frame, and a history of the actions taken by the agent. Additionally, line 5 assumes access to the (x, y) velocity history $vHist$ of every blob. Next, Algorithm 1 computes the entropy of blob b's velocity his-

Algorithm 1 Identify Self

1: $actions \leftarrow$ set of actions applicable to this game
2: $current_blobs \leftarrow$ set of blobs in the current game frame
3: $ActionHist \leftarrow$ Set of actions at time 0...n
4: **for** blob $b \in current_blobs$ **do**
5: $\quad vHist_b \leftarrow$ Set of velocities of blob b at time 0...n
6: $\quad H_b \leftarrow H(vHist_b)$
7: $\quad$ **for** action $a \in actions$ **do**
8: $\quad\quad vHist_{(b|a)} \leftarrow [vHist_b[t] \; \forall_t : ActionHist[t-1] == a]$
9: $\quad\quad H_{(b|a)} \leftarrow H(vHist_{(b|a)})$
10: $\quad$ **end for**
11: $\quad InfoGain_b \leftarrow H_b - sum_{a \in actions}(p_a * H_{(b|a)})$
12: **end for**
13: **return** $arg_max_{b \in current_blobs}(InfoGain_b)$

tory. Entropy is calculated using the standard formulation: $H(X) = -\sum_{i=1}^{n} p(x_i) * ln(p(x_i))$. Taking entropy over a velocity history involves computing the distribution over blob b's velocity values. This computation is done using the empirically observed frequencies of each observed (x, y) velocity value in b's velocity history. A blob with highly random movement will exhibit a high information entropy over its full velocity history while a blob with highly regular movement will yield low entropy.

Having computed the information entropy over b's full velocity history, Algorithm 1 next examines each action individually and, in line 8, create b's selective velocity history $vHist_{(b|a)}$. The selective velocity history simply filters the full velocity history by including only velocities that were observed in frames after which action a was taken. For example, if $a =$ joystick left, then $vHist_{(b|a)}$ would only contain resultant velocities for frames in which a was the action selected. In line 9, entropy is computed over b's selective velocity history. This selective entropy $H_{(b|a)}$ should be low if a given action reliably causes the blob to move in a certain direction.

Finally, information gain is computed in line 11 by subtracting a frequency weighted sum of a blob's selective velocity entropies from the blob's full velocity entropy. If each of the selective entropies is low, as should be the case for the self blob, a high information gain results. In line 13, the algorithm concludes by returning the blob with maximum information gain.

While Algorithm 1 is generally successful in identifying the self blob, sometimes game dynamics break the assumption that actions result in similar velocities. For example, in the Freeway game, after colliding with a car, control is taken from the player and the chicken inadvertently is moved down for several frames regardless of which actions the agent is executing. This temporary lack of control results in irregular selective velocity histories and temporarily poorer identification of the self. However, in some sense the algorithm is correct in losing confidence in an object over which it no longer has control. In practice, since the chicken is still the most controlled blob, it remains the self.

5.3 Atari-HyperNEAT Interface

After extracting object classes as well as the location of the self from the raw game screen, this information needs to be sent to HyperNEAT. As discussed in Section 4, HyperNEAT evolves a CPPN that encodes an ANN. This section assumes access to a fully connected 2-layer ANN whose weights have been specified by the CPPN. At a high level, information from the game screen needs to be translated to activations of nodes in the input layer of the ANN. Then, after the network has been run in the standard feed-forward fashion, the activation of nodes on the output layer must be interpreted in order to select an action.

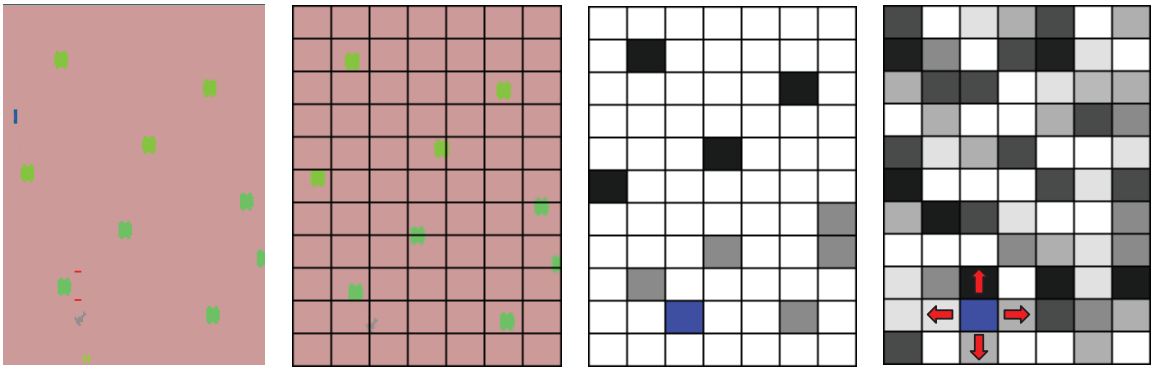
Figure 4 shows an example of how object classes are given as input to the substrate layer of the ANN. Since the ANN input nodes can only take real-valued activations, each class of objects must be mapped to a real number. Thus a mapping from object classes to real values is maintained. Upon discovery of new object classes, real values are incrementally added to the map in a non-decreasing fashion. The raw game screen of the Atari console displays a native resolution of 160×210, which is discretized by a factor of 10 in each dimension to produce a 16×21 grid of ANN input nodes. Figure 4b shows an example discretization. Following this step, each cell that contains an object is fed as input to the ANN with a real valued activation corresponding to the mapping of that object class. Cells devoid of objects are given input activations of zero. The ANN is run in a standard feed-forward manner, producing activations of the nodes in the 16×21 output layer (Figure 4d). Action selection involves locating the cell corresponding to the detected self object (colored blue in Figure 4d). The activation of this cell as well as the activations of the four adjacent cells (shown with red arrows) are compared and the action corresponding to the arrow in the highest of these five cells is returned (or no-op if the self square has the highest value). This method of action selection supports up to five possible actions. Future work involves extending this framework to support more complex actions such a button presses and combinations of joystick and button-press actions.

6. EXPERIMENTAL SETUP

The experimental setup was the same for both Freeway and Asterix: HyperNEAT-GGP was run for 250 generations, with 100 individuals in each generation and a substrate resolution of 16×21. All individuals learn from scratch and were evaluated on the Atari simulator using the same random seed. Evaluations of the 100 individuals in each generation were performed in parallel on a Condor cluster, resulting in each generation taking several minutes of wall-clock time to evaluate. Results are averaged across five runs of HyperNEAT-GGP evolution and compared with previous results obtained using gradient descent Sarsa(λ) with linear function approximation[11].

Sarsa(λ) was implemented as follows. Due to the large size of the state space, the Sarsa(λ) agent uses linear function approximation in which each state is represented by a vector of n values, $\Phi(s)$, known as the feature vector, where $n \ll |S|$. An additional vector of parameters θ is used to estimate the value of a state as follows: $V_t(s) = \sum_{i=1}^{n} \theta_t(i)\Phi(s)(i)$. Thus the objective is to find values for θ such that, when combined with the feature vector for any state, the estimated state value is close to the actual state value. This is accomplished by updating θ at each timestep as follows: $\theta_{t+1} = \theta_t + \alpha\delta e_t$ where α is the learning rate, δ is the temporal difference error, and e_t is the eligibility trace. More information about Sarsa(λ) can be found in Sutton and Barto [16].

Three different variations of the Sarsa(λ) agent were included, each using different feature vectors to represent the state of the Atari game. First, Sarsa(λ)-BASS (Basic Ab-



(a) Object Classes **(b)** Grid Overlay **(c)** ANN Input **(d)** ANN Output

Figure 4: The interface between visual processing framework and the HyperNEAT ANN. Classes of objects are discretized into a grid whose cells are fed to the input nodes of the ANN via a map from object classes to real numbers. After running the ANN, activations of the output layer in cells adjacent to the detected self are used to select which action the agent should take. In this case, the agent would move up since that adjacent cell has the highest activation.

straction of Screen Shots) discretizes the screen into a binary vector v_l of length 14x18x8 where 14 and 18 are the width and height of the discretized screen and 8 is the number of possible colors in SECAM mode. Next it creates another binary vector v_q which contains the pairwise ANDs of all items in v_l. Thus the full feature vector Φ_a is a $(|v_l|+|v_q|)\times|A|$ bit binary vector representing an abstraction of the raw game screen.

Second, Sarsa(λ)-DISCO (Detecting Instances of Class Objects) uses an object detection framework similar to the one outlined in Section 5 to detect classes of objects present in the current screen. To generate Φ it encodes the absolute location of each object class on the screen as well as the tile-coded relative position and velocity for each pair of object classes. Since there may be multiple instances of each object class present on a given screen and Φ is required to have a fixed length, tile-coded positions and velocities are found for each instance and summed.

Third, Sarsa(λ)-RAM uses a state representation based on the 1024 bits of random access memory (RAM) available to the Atari console. A binary vector v_l is created which contains each of the 1024 bits of memory. Next another binary vector v_q encodes the ANDs of all bits in v_l. Φ is a direct concatenation of v_l and v_q. For detailed descriptions of these learning agents, see Chapter 2 of [11].

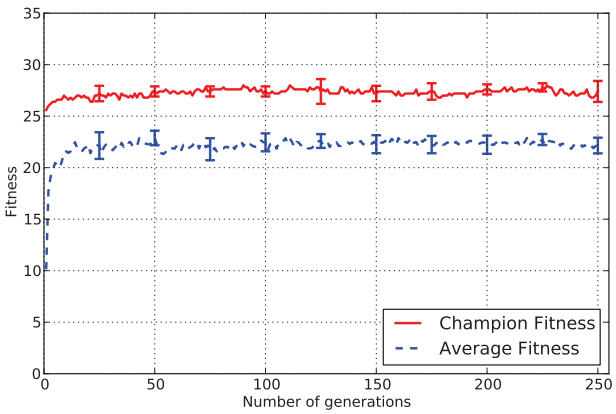


Figure 5: HyperNEAT-GGP learning performance on the Freeway game. Average fitness of the population along with the champion fitness for each generation are displayed. Error bars represent standard deviation. The fitness of an individual corresponds exactly to its game score. As the figure shows, effective Freeway policies are found in early generations.

7. RESULTS

This section provides experimental results obtained from applying HyperNEAT-GGP on two Atari games, *Freeway* and *Asterix*. Table 1 presents the fitness values of the best individual averaged across all the runs, as well as the overall best individual found. Previously published performances of BASS, DISCO, and RAM gradient descent Sarsa(λ) agents are also presented as well as an agent taking random actions each step [11].

7.1 Freeway

In Freeway the player controls a chicken as it attempts to cross the road. Average fitness of the population and the champion fitness throughout the learning process can be seen in Fig. 5. The most striking observation from these results is that the Sarsa(λ) agents are unable to find a solution to the problem (evident in the zero scores in Table 1), whereas even the first generation in HyperNEAT-GGP produces a champion with a very high fitness (starting average champion fitness = 26.4). Sarsa(λ) agents exhibit zero score due to the sparse reward on this domain. An agent would have had to perform a large number of exploratory actions before stumbling on the goal state at the top of the screen and receiving a reward. On the other hand, all individuals in HyperNEAT are produced by randomly initialized CPPNs. Due to the nature of CPPNs, at least a few individuals have the propensity to always take the *Up* action and obtain a large fitness at the outset. Evolution helps individuals to learn quickly to avoid cars over the next 50 generations, which can be seen by the slight increase in champion and average fitness.

	Freeway	Asterix
Sarsa(λ)-BASS	0	402
Sarsa(λ)-DISCO	0	301
Sarsa(λ)-RAM	0	545
Random	0	156
HyperNEAT-GGP (Average)	27.4	870
HyperNEAT-GGP (Best)	29	1000

Table 1: Game scores obtained in the Freeway and Asterix games. HyperNEAT-GGP substantially outperforms Sarsa(λ) on both Freeway and Asterix. The last two lines report the average and best champion's score at generation 250.

7.2 Asterix

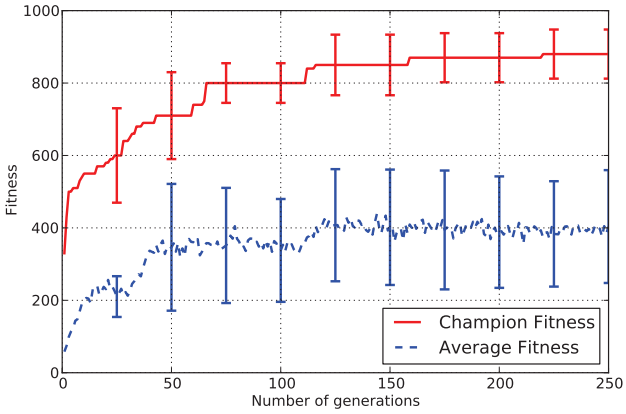


Figure 6: HyperNEAT-GGP learning performance on the Asterix game. The average fitness of the population along with the champion at each generation in the Asterix game. Error bars represent standard deviation. Fitness of an individual corresponds exactly to their game score. As the figure shows, policies are continually improved throughout the course of the 250 generations.

In Asterix, the player controls a unit called Asterix with the objective of collecting magic potions and avoiding *lyres*. Average fitness of the population and the champion fitness throughout the learning process can be seen in Fig. 6. Results for Asterix were averaged across five runs of HyperNEAT-GGP evolution. Table 1 contrasts our approach with previous results. The results for Asterix are qualitatively different from Freeway in a number of ways: First, random exploration obtains non-zero reward in Asterix as Asterix inadvertently collects magic potions. The Sarsa(λ) agents can bootstrap from this information and learn to become statistically better than random. Additionally, champions from HyperNEAT evolution start at close to random performance (starting average champion fitness = 80) and improve their performance to the same level as Sarsa(λ) within 50 generations. Finally, the learning process steadily improves the fitness through the entirety of 250 generations. This steady improvement demonstrates the power of using CPPNs at representing good policies for this game.

Results show excellent performance by HyperNEAT-GGP on the Asterix and Freeway games. Informal comparisons with agents controlled by the authors of this paper indicate that HyperNEAT-GGP achieves scores on par with human play. However, to extend HyperNEAT-GGP to play arbitrary games in the Atari simulator, some future work is required.

8. FUTURE WORK

The most pressing direction for future work is to extend HyperNEAT-GGP to a larger set of games. There are three main challenges: large numbers of possible actions, many different object classes, and robust visual processing.

In Freeway and Asterix, like in Robocup Keepaway, there are relatively few classes of objects that matter: cars and the chicken for Freeway, potions and lyres for Asterix, and takers and keepers for Keepaway. With a limited number of object classes it is easy to map from objects to substrate activations. For example, in Keepaway, keepers can be assigned values of 1 and takers values of -1. Having such few values allows HyperNEAT to easily differentiate between classes of objects and exhibit appropriate behaviors for each, such as avoiding lyres and collecting potions.

It is more difficult to differentiate between object classes as the number of classes increases and crowds the map of object class to substrate values. Accordingly, HyperNEAT becomes increasingly unable to distinguish between and formulate appropriate strategies for dealing with each class.

The second area of future work involves developing a better way to handle a large number of actions. In games like Freeway in which there are only a few actions (up, down, no-op), it is possible to choose which action to take by examining values of the output nodes adjacent to the self node (as described in Section 5.3). This issue becomes more complicated when other actions such as button presses are involved. For example, which node's value should be examined to decide if the button press action should be taken?

Finally, the visual processing stack could be made more robust. Specifically, objects are sometimes lost when they change shape or rotate. Additionally unmoving objects such as walls are not detected. This introduces difficulties in games such as Pac-man where static objects are essential to the game dynamics. Finally self-identification could also be improved by incorporating additional information about how long each object has been on-screen, with the assumption that the self object remains on screen while other objects are transitory. Further changes are encountered with self objects which retain velocity – such as the spaceship in Asteroids.

Addressing these areas of future work will go a long way towards making HyperNEAT-GGP more generally applicable to Atari games.

9. CONCLUSION

This paper introduces HyperNEAT-GGP, a HyperNEAT-based general Atari game playing agent. Many Atari games contain geometric regularities in the two-dimensional space of the game screen. This structure allows HyperNEAT to quickly learn effective policies. To reduce the complexity of learning from the raw game screen, HyperNEAT-GGP employs a game-independent visual processing hierarchy designed to identify classes of objects as well as the entity that the player controls on the game screen. Identified objects are

provided as input to HyperNEAT. Due to the computational overhead of visual processing applied to each game screen, a parallel architecture is used to evaluate multiple individuals simultaneously. Results were presented for two Atari games, *Freeway* and *Asterix*. In both cases, HyperNEAT-GGP was shown to outperform previous reinforcement learning benchmarks [11]. While no single Atari game, if studied in isolation and given extensive feature engineering, likely poses too great a challenge for modern AI techniques, the full collection of over 900 Atari games presents a daunting task for a single learning agent. HyperNEAT-GGP represents a first step towards the ambitious goal of creating an agent capable of learning and seamlessly transitioning between many different tasks.

10. ACKNOWLEDGMENTS

This research is supported in part by NSF (IIS-0917122,DBI-0939454,IIS-0915038), ONR (N00014-09-1-0658), and FHWA (DTFH61-07-H-00030).

11. REFERENCES

[1] M. Campbell, A. J. H. Jr., and F. hsiung Hsu. Deep blue. *Artif. Intell.*, 134(1-2):57–83, 2002.

[2] J. Clune, B. E. Beckmann, C. Ofria, and R. T. Pennock. Evolving coordinated quadruped gaits with the hyperneat generative encoding. In *Proceedings of the Eleventh conference on Congress on Evolutionary Computation*, CEC'09, pages 2764–2771, Piscataway, NJ, USA, 2009. IEEE Press.

[3] D. B. D'Ambrosio and K. O. Stanley. Generative encoding for multiagent learning. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 819–826, New York, NY, USA, 2008. ACM.

[4] C. Diuk, A. Cohen, and M. L. Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of 25th International Conference on Machine Learning (ICML)*, pages 240–247, 2008.

[5] G. Edgers. Atari and the deep history of video games. `http://www.boston.com/bostonglobe/ideas/articles/2009/03/08/a_talk_with_nick_montfort/`.

[6] D. Ferrucci, E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. Prager, N. Schlaefer, and C. Welty. Building Watson: An Overview of the DeepQA Project. *AI Magazine*, 31(3), 2010.

[7] J. Gauci and K. O. Stanley. A case study on the critical role of geometric regularity in machine learning. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI)*, 2008.

[8] M. Genesereth and N. Love. General game playing: Overview of the aaai competition. *AI Magazine*, 26:62–72, 2005.

[9] S. M. Lucas. Ms pac-man competition (screen capture mode). `http://dces.essex.ac.uk/staff/sml/pacman/CIG2011Results.html`.

[10] S. M. Lucas. Ms pac-man competition. *SIGEVOlution*, 2(4):37–38, 2007.

[11] Y. Naddaf. Game-independent ai agents for playing atari 2600 console games. Master's thesis, University of Alberta, 2010.

[12] M. Parker and B. Bryant. Backpropagation without human supervision for visual control in quake ii. *Proceedings of the 2009 IEEE Symposium on Computational Intelligence and Games (CIG'09)*, pages 287–293, 2009.

[13] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[14] P. Stone and R. S. Sutton. Scaling reinforcement learning toward RoboCup soccer. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 537–544. Morgan Kaufmann, San Francisco, CA, 2001.

[15] P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

[16] R. S. Sutton and A. G. Barto. Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks*, 9(5):1054–1054, 1998.

[17] G. Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Comput.*, 6:215–219, March 1994.

[18] P. Verbancsics and K. O. Stanley. Evolving static representations for task transfer. *J. Mach. Learn. Res.*, 11:1737–1769, August 2010.