Reinforcement Learning and Neural Networks for Tetris

Nicholas Lundgaard SHADOWFAX@OU.EDU
Brian McKee B1MCK@OU.EDU
University of Oklahoma

Abstract

This paper presents the results of experiments carried out with the goal of applying the machine learning techniques of reinforcement learning and neural networks with reinforcement learning to the game of Tetris. Tetris is a well-known computer game that can be played either by a single player or competitively with slight variations, toward the end of accumulating a high score or defeating the opponent. The fundamental hypothesis of this paper is that if the points earned in Tetris are used as the reward function for a machine learning agent, then that agent should be able to learn to play Tetris without other supervision. Toward this end, a state-space that summarizes the essential feature of the Tetris board is designed, high-level actions are developed to interact with the game, and agents are trained using Q-Learning and neural networks. As a result of these efforts, agents learn to play Tetris and to compete with other players. While the learning agents fail to accumulate as many points as the most advanced AI agents, they do learn to play more efficiently.

1. Introduction

The goal of this project is to explore the effectiveness of unsupervised machine learning techniques for learning to play the game of Tetris, and the effectiveness of the strategies learned by agents using these techniques when compared to the strategies of other players. Choices have been made to increase the similarity of the experience of the learning agent to the experiences of a human player.

1.1 Tetris

Tetris is a well known computer game originally programmed by Russian mathematician Alexey Pajitnov, and subsequently released for nearly every computer and gaming platform ever made. It is played on a rectangular grid partitioned into smaller square areas, typically ten units wide by twenty units tall. The player controls the orientation and horizontal location of pieces that fall from the top of the board to the bottom and earns points by forming complete horizontal lines, which are then removed from play, causing pieces placed higher to move downward.

Each piece is made of four square units of size to match the square units of the grid, joined on complete edges. All possible combinations of four squares result in seven possible pieces. These pieces, called tetrominoes, can be labeled using the Roman letters they most resemble; the piece consisting of four blocks in a single row is labeled an ‘I’, the piece consisting of four blocks in a 2-by-2 square an ‘O’, and so on. The other pieces are labeled ‘J’, ‘L’, ‘Z’, ‘S’, and ‘T’. Pieces are often given distinct colors or patterns to differentiate them from each other.

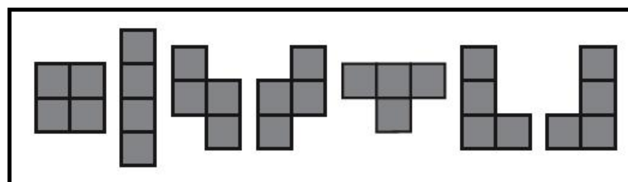

Figure 1. Tetris Pieces. From left to right, O, I, S, Z, T, L, and J. Source: (Siegel, Chaffee, 1996).

Score is awarded for clearing lines, with the amount of points earned for clearing multiple lines simultaneously greatly exceeding the amount earned for clearing the same number of lines individually. As cleared lines are accumulated, the speed at which pieces fall increase, generally after groups of ten lines have been cleared. Small amounts of points are also awarded simply for placing pieces.

Perhaps the most influential version of Tetris ever was released in 1989 for the Nintendo Entertainment System. A closely related version was also packaged with new Nintendo Gameboy units, and would later be released on the Super Nintendo Entertainment System featuring competitive play. Although there are many variants of Tetris, this version holds a central place in the history of the game, and the version of Tetris implemented here is based on this version.

1.2 Earlier Tetris Players

Tetris has been the subject of much inquiry as to its complexity. It is known to be NP-Complete (Demaine, Hohenberger, Liben-Nowell, 2003), and that a long sequence of Z or S tetrominoes will cause any Tetris game to be lost (Burgiel, 1997). No winning strategy has

been described, although every player surely has his or her own method.

Many programs exist to play Tetris, but of particular note is one developed by Colin Fahey, who attained notoriety in 2003 for creating a very effective Tetris playing agent. Fahey implemented a version of Tetris based on the initial PC release, and created a framework by which it could be operated by either a human player, or by an agent designed to simply be plugged in to the modular framework[1]. A modified version of Fahey's Tetris is used for this experiment.

Fahey's agent, given a board configuration, considers possible board configurations created by placing the currently falling piece in all possible locations (known as translations) and orientations (rotations). Each created board is assigned a merit based on characteristics including the overall height of the pile of pieces, and the number of buried, empty holes in the pile. These characteristics are weighted using factors determined by Fahey through a process of systematic and randomized searching.

Also making use of Fahey's framework is Pierre Dellacherie, whose agent similarly explores possible moves and evaluates the resulting boards based on a weighted set of characteristics. As compared to Fahey, Dellacherie created several new metrics of evaluating a board configuration, and chose his weights by hand after manually varying them. While Dellacherie's agent's evaluation metrics are hand-chosen, Colin Fahey noted that Roger Llima utilized the machine learning method of genetic algorithms to optimize the weights of the various evaluation metrics available to the agents in his framework (Fahey, 2003). This method of machine learning has previously been applied to Tetris (Siegel, Chaffee, 1996), but had not previously been used for computing metric weights.

1.3 Reinforcement Learning and Neural Networks for Games

Reinforcement learning is a machine learning technique in which agents make actions based on their environment, affecting the environment and receiving a reward. A policy is created mapping states of the environment to actions taken, and this policy is modified based on rewards received toward the end of maximizing the long term reward.

Tetris appears to lend itself to reinforcement learning because in particular because it is ill-suited for supervised learning (as it is nearly impossible to definitively say what was a right or wrong move), and because there is an obvious and convenient reward function built into the game: the scoring function. Reinforcement learning has previously been applied to Tetris, with particularly mediocre results (Carr, 2005).

However, other literature has suggested that a method of reinforcement learning called "Relational Reinforcement Learning" (RRL) might be useful for the game of Tetris (Driessens, Ǯzeroski, 2004). It is certainly true that RRL is one method of generalizing the same basic action across numerous raw states, and that this solution appears to lend itself to Tetris. However, this method of generalization was not pursued here, and cannot accurately be compared to RRL methods due to a lack of available results in any framework using such a method. Here, a meshing of high-level heuristic actions and high-level state representation is used to allow agents to generalize from the intractable state-space.

Neural networks are machine learning tools built to simulate biological neurons. Given input, output is generated as the input flows through a set of interconnected neurons, each connection being multiplied by a certain weight to generate the values that reach the outermost output neurons. The output of the net can then be considered the prediction of the model. Neural networks are a supervised learning technique, and the weights are adjusted using the difference between the model prediction and the expected result.

Neural networks can be combined with reinforcement learning for use in unsupervised learning. The output of the net becomes the expected reward, which can be used to determine which action to take. Weights are then adjusted using the difference between the expected reward and the actual reward received. In this manner the network summarizes the state-action policy and the expected rewards.

Neural networks with reinforcement learning have been applied with great success to the game of backgammon (Tesauro, 1995). Here experiments were performed using neural networks and reinforcement learning using both the raw configuration of the Tetris board and a higher level state representation as input.

2. Tetris Implementation Details and Hypotheses

Fahey and Dellacherie have created agents that survive millions of pieces and clear hundreds of thousands of lines. However, several questions arise for the seasoned human Tetris player.

2.1 Line Limits

First, although computer agents are capable of playing thousands of pieces per second, because the speed at which pieces fall increases, human players are limited by their reflexes to a much smaller number of cleared lines. Some implementations of the game may even increase the downward velocity of the pieces so far beyond the horizontal velocity that meaningful play is literally impossible.

[1] For details on the Tetris variant and on Fahey's agent, see http://www.colinfahey.com/tetris

To simulate a human player's experience, in this project, an option has been added to Fahey's Tetris implementation to limit the number of lines cleared before the game ends.

2.2 Score Versus Lines

Second, Fahey's implementation, in contrast to most other (including the aforementioned Nintendo version) awards points only for dropping pieces quickly, not for clearing lines. The effect is that the same number of lines cleared results in roughly the same number of points earned, regardless of if these lines were cleared simultaneously or individually. Indeed, Fahey presents his results in terms of lines cleared rather than score earned.

If there is no score incentive for clearing multiple lines simultaneously, there is no reason for these agents to engage in risky behavior such as building large piles of pieces to be cleared (simultaneously) later. Since high score, not the exact manner in which it was earned, can be considered the motivating factor for any game, for this project, the scoring function from the Nintendo release, which heavily weights lines cleared simultaneously, was added to Fahey's Tetris implementation.

2.3 Competitive Play

Related to the increased score for multiple simultaneous lines is the competitive version of Tetris. Two players compete to place the same sequence of pieces on their own respective boards. When a player simultaneously clears multiple lines, 'garbage' lines are added to the bottom of the opponent's board, shifting their pieces up.

Lines are sent to the opponent in the amount of those lines cleared in excess of one, except in the case of four lines cleared, in which case all four lines are sent to the opponent. Interestingly, lines sent to the opponent appear with only one column not filled, ready to be cleared simultaneously again.

It is clear that agents who simply clear lines individually may not defeat an opponent who clears lines simultaneously. A goal of this project was to implement agents that clear multiple lines simultaneously both to earn a high score, and to defeat opponents in competitive play. As such, these competitive features have been added to Fahey's implementation.

2.4 Hypotheses

The most fundamental goal of this project was to create Tetris-playing agents that play the variant of Tetris described above. It is believed that if agents are trained with nothing more than the scoring function of lines cleared as their reward, then these agents should learn strategies enabling the effective play of this Tetris variant.

As compared to agents that work only from heuristic board evaluation functions, agents that are trained using the scoring function (that heavily weights multi-line clears) as their reward function should be able to employ strategies involving accumulating piles of pieces to be cleared by one long piece later, and by doing so score more points per line than existing agents.

It was further believed that these agents trained in this way would be able to achieve higher overall performance than existing agents.

3. States and Actions

Although some experiments were performed using the raw configuration of the Tetris board as input to the model (the results of which will be shown below), most experiments were performed using a higher-level representation of the Tetris board, and of the possible actions, in an effort to speed learning by summarizing the relevant features of the state space. Both the raw representation and the high-level representation are discussed below.

3.1 Raw State Representation

3.1.1 The Board

As mentioned above, the Tetris board is a rectangle gridded area measuring twenty units tall by ten units wide. Each unit may be either occupied or unoccupied by a section of a previously fallen piece. Additionally, at any time, there is one piece falling toward the pile from the top of the board, which may take on one of seven shapes. Additionally, there is an optional look ahead of one piece, the next to fall. The calculation for the number of distinct Tetris states is shown below.

$$2^{200} \times 7^2 \approx 7.87 \times 10^{61}$$

3.1.2 Rotations and Translations

Each falling piece is placed on top of the topmost line of accumulated pieces, possibly having been rotated and/or translated. Depending on the piece shape, there is a minimum of one (for the 'O' shape) and a maximum of four (for the 'T', 'J' and 'L' shapes) orientations. Depending on its orientation, a piece's width allows it to be placed in eight to ten horizontal locations among the ten columns. It can be stated naively that there are approximately 40 different final placements for any piece.

3.1.3 Problems with the Raw State

In addition to its foreboding size, there are problems with the raw configuration of the board as a meaningful representation of the state of the game. Most obvious is the fact that, given that the falling piece shapes are horizontally symmetrical to either rotated instances of themselves or to another piece (as is the case with 'J' and 'L' pieces). Combined with the fact that the pile of already fallen pieces can be horizontally symmetrical, this means that some game states are exactly flipped versions of other states, and that the actions taken in one, if reversed, should produce the same results in the other.

Clearly a high-level state representation should account for this.

Similarly, because pieces, regardless of orientation, are of width no greater than four, and some orientations result in pieces only one or two units wide, game boards with the same columns (or groups of columns) rearranged can also be considered equivalent. For instance, consider a board completely filled to a depth of 5, save for one column, which is not filled at all. It makes little difference whether this open column is shifted to the left or right.

Finally, pieces are also of extremely limited height, and the depth of the openings into the pile of accumulated pieces may be limited as well. The exact configuration of the board below the top layer may well be irrelevant when placing a piece that cannot penetrate so far.

3.2 High-Level State and Actions

3.2.1 High-Level State

In constructing a high-level state for learning, it is important to reduce the size of the state-space to a tractable size that will allow agents to experience all or almost all states multiple times during training. Yet this size reduction needs to be carefully constructed such that it exposes important features of the game to the machine learning agent.

When a human looks at a Tetris board during game-play, it's clear that person is not observing the board's raw state. Humans effectively note features like the overall height and evenness of the pieces on the board, the number of holes in the board, and the presence of "valleys" into which pieces may be dropped to clear lines.

With such generalizations in mind, the state-space constructed contained the following features:

1. Current Piece
   This is the shape of the piece that the agent has to place in the current round. Shape is ∈ {I, O, S, Z, T L, J}.
2. Next Piece
   This is the shape of the piece that the agent will receive as the current piece in the next round. Shape is ∈ {I, O, S, Z, T L, J}.
3. Board Height
   This is a measure of the average height of the board. Board Height is ∈ {Z | 0 <= height < 20}.
4. Board Level
   This measure is a Boolean value that states whether or not the column heights of the board are even (within a modest threshold).
5. Has Single Width Valley
   This measure is a Boolean value that states whether or not there is a slot that an I piece might fit into.
6. Has Multiple Valleys
   This measure is a Boolean value that states whether or not there are multiple slots. Multiple slots counterbalance the single slot measure because multiple slots mean that you're potentially building a "tower," which is the bane of many a game.
7. Number of Buried Holes
   This aspect of the board measures the number of holes that are covered by other pieces on the board. The value is reduced to the following ranges: none, 1-5, 6-10, 10 or more.

Using these features, the state-space is constrained to:

$$7^2 \times 20 \times 2^3 \times 5 \approx 40{,}000$$

This state-space is easily discoverable in a few games, and it should give agents a way to identify important features such as valleys and relative board levelness across raw states that would otherwise look totally different.

3.2.2 High-Level Actions

Now that the state-space has been defined in such a way that agents can generalize states, they need a set of high-level actions that they can take that will work generically—e.g., an action that, given a slot of depth 4 and an I piece, will drop that piece into that slot, whether that slot is on the left, right, or somewhere in the middle. In other words, the actions need to be configuration-independent.

To do this, the actions are simply classes that have a function to perform a greedy search on a board configuration across every possible translation and rotation of a given piece, based on a given evaluation of a move. Each action below uses the same greedy search method; in fact, the only difference between each action is the merit evaluation function it uses on each trial move.

Note that the performance of each individual action described in the following sections is evaluated concretely in Section 5.2 on Heuristic Agents results.

3.2.2.1 Minimize Holes

The minimize holes action penalizes moves that increase the number of covered holes in the board configuration. This action is actually fairly effective, except that it inserts large towers in the board configuration over time, because it will stack pieces very high rather than inserting a hole by stacking them more evenly.

3.2.2.2 Minimize Column Height

The minimize column height action uses a simple measure of penalizing the score based on the sum of the column heights on the board.

This action might be considered a somewhat worthless action (see the results in Section 5.2). It was included in

the action set in order to test agents' ability to learn not to use it.

3.2.2.3 MAXIMIZE LINES

This action simply measures a move's utility based on how many lines it scores. This is quintessentially a great measure of performance when there is a board configuration that has moves available that can score lines. If there are no such moves available, however, the measure gives no hints as to what to do, and so the action performs randomly (poorly).

3.2.2.4 CREATE 'I' VALLEY

This action measures actions based on the existence of a single-width valley in the board configuration of depth 4, for the placement of an I piece. In addition to attempting to leave such a valley, the action values hole-minimization in the placement

This is our "riskiest" action in that it avoids clearing lines in order to maintain the opportunity to clear 4 lines in a single move when an I piece is received. Of course, an agent would have to learn to take a different action after the valley is created by this action.

3.2.2.5 CREATE 'L' VALLEY

This action is nearly identical to the previous action. However, it is considerably less risky inasmuch as it tries to create a single-width valley of depth 2 rather than 4, in order to place an L or J piece.

Once again, any agent hoping to profit from this action will have to learn to take a different action to fill the valley after it is created, because this action will preserve it.

3.2.2.6 CLEAR BOARD ACTION

The clear board action is intended to be the most effective action: it acts to minimize the holes on the board, maximize the evenness of the column heights, and minimize the average height of the board. In essence, when this action is invoked, it tries to decrease the board height by getting lines. While it would clear multiple lines at a time if that action were available, it makes no attempt to create such situations—it simply greedily tries to get lines. It is anticipated that the action primarily gets single lines, occasionally getting double-line clearings.

4. Implemented Agents

4.1 Random Agent

A random agent operating in the Tetris game space need only generate random rotation and translation values for each falling piece. Although not all piece shapes have four distinct orientations, pieces rotated extra times simply return to earlier orientations. If a translation is chosen that would carry a piece beyond the edges of the board, it simply stops at the edge.

For these reasons, implementing a random agent was as easy as generating one random integer between 0 and 3, and one random integer between -5 and 5. The results for the random agent are presented below.

4.2 Heuristic Agents

Agents were implemented that always chose to perform a single high-level action from the list above. These agents proved to be useful tools for evaluating the efficacy of the high-level actions themselves, and served as interesting and useful baselines to compare with the learning agents described below. The results for some of these heuristic agents are presented in Section 5.2.

4.3 Reinforcement Learning Agent

In designing a reinforcement learning agent, it was decided that Q-Learning would be used. The reasoning behind this choice is driven by 2 primary factors. First, Q-Learning appeared to be a good fit for the neural networking agent described in the next section. The main reason, however, that Q-Learning was chosen, however, is that it is generally much riskier and ultimately more optimal than SARSA learning.

The goal of the reinforcement learning agent is to progressively evolve a simple move-evaluation function, $Q(s,a)$, which is a function that measures the value of action a for state s based on the reward function, which in this case is the number of points scored. Thus, $Q(s,a)$ is a measure of the expected number of points scored for the given move. The purpose of this evolution is to gradually adjust the value of an action given a state based on performance feedback of actually performing the action on the environment the agent acts in. Eventually, the $Q(s,a)$ function converges to the optimal function $Q^*(s,a)$ for the given state-space and action set.

Now, while the agent should generally pick what it thinks is the optimal action at each step, such a policy will severely limit the amount of exploration that the agent will perform. To counteract such behavior, the agent uses an ε-greedy policy (where $\varepsilon = 0.05$) to determine whether or not to take the optimal action at any given step. Thus, 5% of the time, the agent will perform a random action rather than the optimal action for the given state, meaning that over time it will tend to explore many other possibilities.

The $Q(s,a)$ function is implemented as a hash-map that maps a state and an action to a reward value. The agent can query and set any value of $Q(s,a)$.

4.3.1 Q-LEARNING FOR THE REINFORCEMENT LEARNING AGENT

Each time the agent receives a new piece and a board configuration, It considers all values for $Q(s,a_i)$, where s is the current state, and a_i is a member of the set of all available actions in this state. It selects the action a_i with the highest value of Q.

After selecting the action it will take, it simulates the action on the board and computes the reward for the action, r, and the state of the board after performing action a_i, which is s_{t+1}. It then modifies $Q(s,a_i)$ based on the following formula:

$$Q(s, a_i) = Q(s, a_i) + \alpha[r + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s, a_i)]$$

Where a' is the set of available actions for state s_{t+1}, and $\max_{a'} Q(s_{t+1},a')$ is the largest value of Q from among those actions, α is the weighting factor for the mutation of Q, and γ is the weighting factor for the predicted future return of the subsequent state (Sutton, Barto, 1998). Thus, the agent slowly modifies the weights until (hopefully) they converge.

In practice, the agent seems to perform rather poorly. While it appears to learn under some circumstances, it appears that the constantly inconsistent performance of each of the high-level actions prevents the agent's Q-function from converging. The results of the experiments performed with the agent in varying environments are discussed in Section 5.3.

4.4 Neural Network Agent

Experiments were performed using neural networks paired with the Tetris scoring function as a reward function for unsupervised learning. In both cases described below, the network was constructed as input nodes corresponding to the features of the state, each connected to every node of a hidden layer of nodes with sigmoid activation functions. These hidden layer nodes were then each connected to every node of the output layer, in which nodes had linear activation functions, and corresponded to a particular action.

The goal of the network is to predict the value for the actions based on the input state. The use of the linear activation function in the output layer allows the output of these nodes to be unbounded, unlike the output of the hidden nodes with the sigmoid activation function.

Actions are selected, after sending input to the network, using an ε-greedy policy (ε = 0.1), based on the predicted values of the actions. Weights are updated throughout the network using back-propagation of the error between the predicted value and the observed value.

The neural network is implemented through the use of an abstract class 'Neuron', lacking definitions for only its activation function and its differentiated activation function, which are then implemented by subclasses to represent to sigmoid hidden layer nodes and the linear output layer nodes. (Bishop, 2006) was used as a reference for neural network backpropagation equations.

4.4.1 Q-Learning and Back-Propagation for the Neural Network

The actual value of the action chosen by the state is calculated using the observed reward, and the predicted value of the best action for the state arrived in after that action is taken, as per the off-policy reinforcement method Q-Learning.

$$Q(s_t,a_t)_{\text{observed}} = r_{t+1} + \gamma \max_a Q(s_{t+1},a_{t+1})$$

Here, the γ value used was 0.1. The value of the next state, s_{t+1}, is calculated using the network itself, taking the maximum valued output, corresponding to the maximum valued action, rather than using the ε-greedy policy as above.

Of particular interest for the Tetris problem is that the configuration of the next state is created using the actual board configuration created after placing the piece according to the chosen high-level action, and, importantly, using the piece shape previously available as the look-ahead piece shape as the falling piece. In fact, it was the availability of this piece that pointed so clearly to a one-step TD algorithm.

For nodes on the output layer, the error (δ_k) is computed as the difference between the predicted and observed values, multiplied by the value of the differentiated activation function h on the summed, weighted input to that node.

$$\delta_k = h'(i_k)(Q_{\text{predicted}} - Q_{\text{observed}})$$

For nodes on the hidden layer, the error (δ_h) is computed as the sum of the weighted errors of all connected output units k times the value of the differentiated activation function on the summed, weighted input of that node. Since these nodes use the sigmoid activation function, the value of the differentiated activation function can be computed as a function of the output of the node, o_h.

$$\delta_h = o_h(1 - o_h) \Sigma_k w_{hk} \delta_k$$

All weights in the network are updated using the computed δ values as factors of the output value corresponding to that weight. A constant factor, η = 0.1, is used to control the learning rate of the network.

$$w_{ij} = w_{ij} + \eta \delta_j x_{ji}$$

4.4.2 Low-Level Neural Net

Inspired by the success of Tesauro in using neural networks to play the extremely computationally complex game of Backgammon, a net was created which used as its input the features of the low level state described above, and as its actions the raw combinations of the possible rotations and translations of the pieces.

The net featured 202 inputs, 200 corresponding to the grid structure of the board and taking on values of 0 or 1 to indicate if that location was filled, and two corresponding to the shapes of the currently falling and predicted pieces, taking on seven different values. There were 40 output nodes, each assigned a translation value between +5 and -5, and a rotation value between 0 and 3. Upon choosing an output, these translation and rotation values were

applied to the currently falling piece. The number of hidden nodes was varied between 40 and 50, with the results presented below from a net with 40 hidden nodes.

It is believed that this network, given sufficient time, would learn to play Tetris. However, learning results for this network, presented below, seemed to show that this process may be too slow to be useful.

4.4.3 High-Level Neural Net

A.L. Samuel in his 1959 "Some Studies in Machine Learning Using the Game of Checkers" is unimpressed by the possibility of learning a such complex system using a "randomly connected switching net" and instead opts to use a "highly organized network…designed to learn only certain specific things" (March 2000 reprint, p. 207). He goes on to detail features considered important about the Checkers board and strategies for evaluating them, some of which influenced the development of the high-level state above.

After the experiments with the low-level neural net, a high-level neural net was created using the features of the high-level state described above as its input, and as its actions the high-level actions detailed above.

The net features six inputs corresponding to the board height, the current and predicted piece, whether the board can be considered level, whether it has one valley, and whether it has multiple valleys. Boolean values are converter to 0 indicating false and 1 indicating true, and piece shape is represented as one of seven integer values corresponding to the different shapes. There are also six outputs, corresponding to the six actions described above. The results presented below are from a network trained with ten hidden nodes.

Results from the high-level neural net are more encouraging than from the low-level net due in part to the underlying efficacy of the high-level actions. It is clear, nonetheless, that this net learns strategies for Tetris over time.

5. Results

The sections below detail experiments performed with the various agents that were constructed in designing the learning agents, as well as the experiments on the agents themselves. Sections 5.3 and 5.4 are the sections that describe results on the learning agents.

5.1 Random Agent

The random agent earns a small amount of points simply for placing pieces on the board, but very rarely does it randomly clear lines. After playing 500 games and placing over 10,000 pieces, the random agent cleared fewer than 10 lines. There may exist some tasks where a random agent could perform with limited but measureable success; Tetris is clearly to complex a task for this to be true.

Presented below is the performance (that is, the score earned) of the random agent playing 500 games. A moving average with a period of 50 games is drawn on top of the raw data.

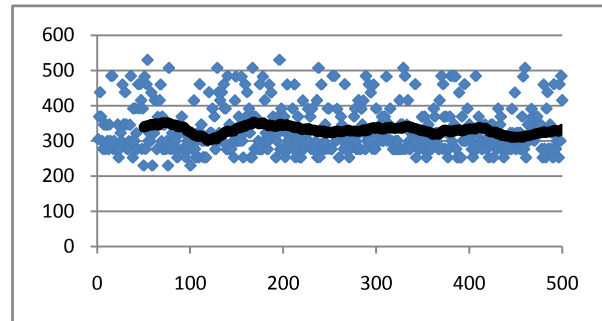

Figure 2. Random agent score with moving average.

5.2 Heuristic Agents

Some of the heuristic agents showed interesting results when used to demonstrate the effectiveness of the high level actions.

5.2.1 Maximize Lines Agent

The results of the agent that always performs maximize lines action are interesting. Somewhat surprisingly, this agent cleared only one like when run for 500 games, placing nearly 8000 pieces. The maximize lines action, as described above, simply places the falling piece wherever it will clear the most lines. If the piece cannot clear any lines, then the last evaluated placement is used, as it shares the same merit as all other placements.

This agent actually performs worse than the random agent. Presented below is the performance of the maximize lines agent playing 500 games. A moving average with a period of 50 is shown on top of the raw data.

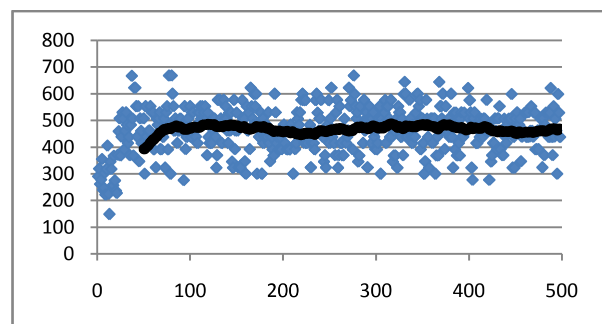

Figure 3. *Maximize lines agent score with moving average.*

5.2.2 Minimize Holes and Minimize Column Height

In contrast to the random agent and the agent that always performed the maximize lines action, the agents that always performed the minimize holes action and the minimize column height action earned points for actually clearing lines.

Both these strategies encourage the agent to clear lines and to support its own longevity, as a board with no internal holes is a board with lines ready to be cleared, and a board with low column height is a board that can still have pieces placed on it. Performance for these two agents is markedly similar across 500 games, with the minimize holes agent averaging 9.522 lines cleared and 1218.84 points earned, and the minimize column height agent averaging 9.542 lines cleared and 1231.56 points earned.

Presented below is the performance of these two agents over 500 games with a moving average with period 50 shown above the raw data.

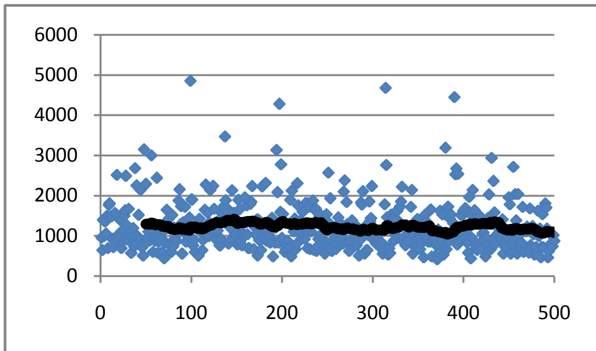

Figure 4. Minimize holes agent score with moving average.

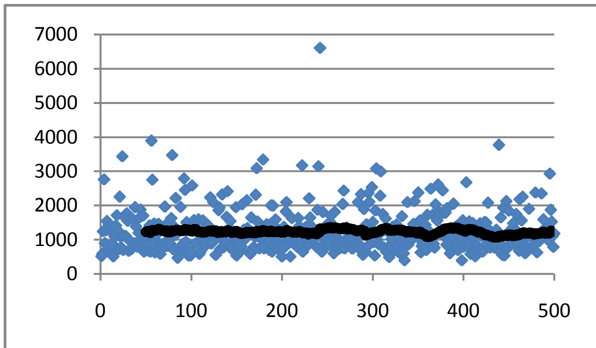

Figure 5. *Minimize column height agent performance with moving average.*

5.2.3 Clear Board Agent

The clear board action, as described above, is a combination of the minimize holes action and the minimize column height action. It was hoped during development that an agent using only this action would have all the advantages of both the agents using these two actions individually, and outperform both of them.

Testing the agent that always performs the clear board action shows that this hope was not unfounded. Although the agent is incapable of forming complicated plans, as it simply explores possible moves with the currently falling piece and evaluates resulting board configurations, across 500 games this agent nevertheless clears an average of 466.494 lines and earns over an average of over 50,000 points. It can be noted that this agent was not tested with a maximum line limit, and on one occasion cleared nearly 3,500 lines.

Presented below is the performance of the clear board agent over 500 games with a moving average of period 50 drawn above the raw data.

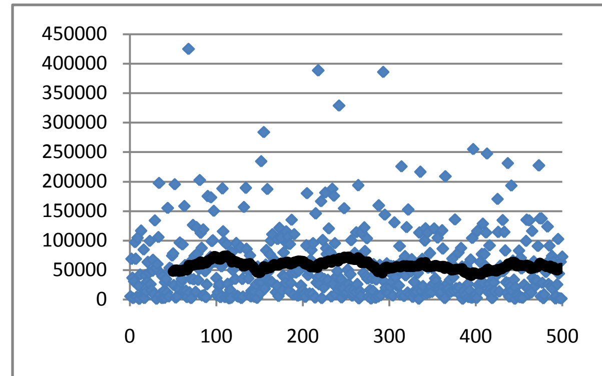

Figure 6. Clear board agent performance with moving average.

5.3 Reinforcement Learning Agent

The reinforcement learning agent, as mentioned previously, did not learn to play very well. Peculiarly, it actually improved its performance and learned noticeably in a competitive environment playing a reasonably expert agent, Pierre Dellacherie, whereas it did not appear to learn at all when playing alone.

The following sections detail 3 experiments performed on the agent in various environments.

5.3.1 Varied α and ε

One of the first things that seemed important to establish through experimentation was the ideal values for α and ε. For reference, α is a measure of how much the value of Q can change at each turn, and ε is a measure of how frequently the agent performs a random action rather than the greedy one. That is, higher α values increase the responsiveness of the agent to new feedback, and higher values of ε increase how much and how quickly the agent explores the state-space of the game. It was hypothesized that modest values for α and ε (0.1 and 0.01, respectively) would perform the best.

The figure below shows the performance of the reinforcement learning agent averaged over 5 runs of 500 games for each selected variation of α and ε (displayed exclusively as moving averages of width 25 to resolve noise). The experiment shows that high values of α and ε (highly responsive and random) perform very poorly. Additionally, it demonstrates that low values of α and ε yield better performance, but generally no indication of learning. However, setting a high degree of responsiveness ($\alpha = 0.5$) and a fairly low degree of randomness ($\varepsilon = 0.01$) actually made the agent learn to do much better than it was initially.

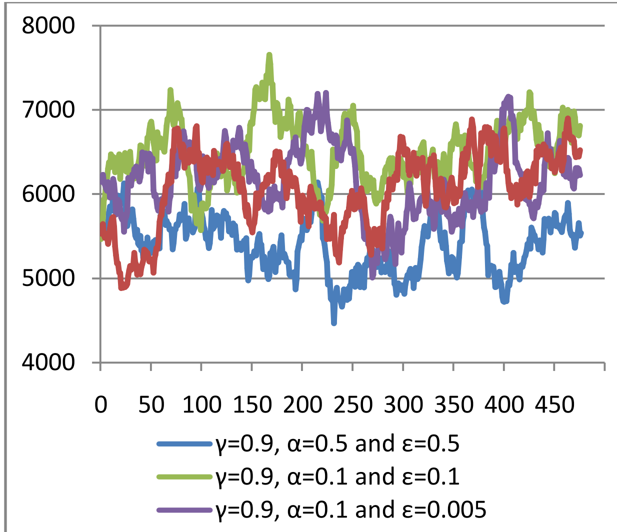

Figure 7. Performance of RL agent over 5 runs of 500 games with varying values of α and ε. Each line represents a moving average of width 50.

5.3.2 Single Player

Considering the above results, the values for α and ε were set for all subsequent RL experiments to be 0.25 and 0.01, respectively. While the hypothesis was that the agent would learn, it did not appear to in a single-player environment. As the figure below demonstrates, the agent did not perform any better than the board clearing agent in the previous section. The figure displays an average over 5 runs of 500 games, overlaid with a moving average plot of width 50.

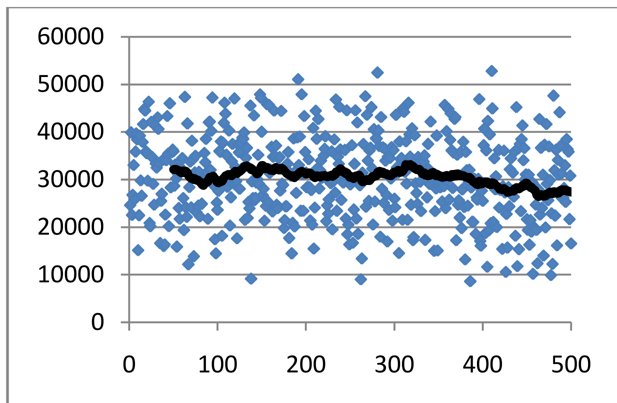

Figure 8. Single player RL agent performance over 5 runs of 500 games, with 50-game moving average.

Again, this behavior appears to be attributable to inconsistent performance of the actions available to the player. Certainly, the very high level of noise suggests this—it matches the heuristic players in appearance. However, note that the player performs a few thousand points more poorly than the board-clearing agent discussed in section 5.2.3. This is clearly attributable to inconsistent actions. In the series of runs above, the agent had all of the high-level actions available to it, even though several of them, such as the line-maximizing agent, perform extremely poorly overall. The fact remains that, however poorly such actions perform, they get extremely large rewards under certain circumstances, and the RL agent appears to totally lack the facilities to observe and avoid this inconsistency.

5.3.3 Competitive

Given the agent's poor performance with all actions available to it, it was decided that the agent would be restricted to the more consistent, higher-performing high-level actions. In addition, a new action was created based on Pierre Dellacherie's agent—essentially a high-level action that performs the exact same action that Dellacherie's agent would take, given the same raw state. It was hypothesized that such a modification would allow the agent to learn to beat Dellacherie in competitive play by setting itself up for multiline clears using the valley-creating action, and then using an action such as the Dellacherie high-level action to get the large clear.

The first figure below (Fig. 9) demonstrates fairly similar behavior to the single player experiment in terms of noise.

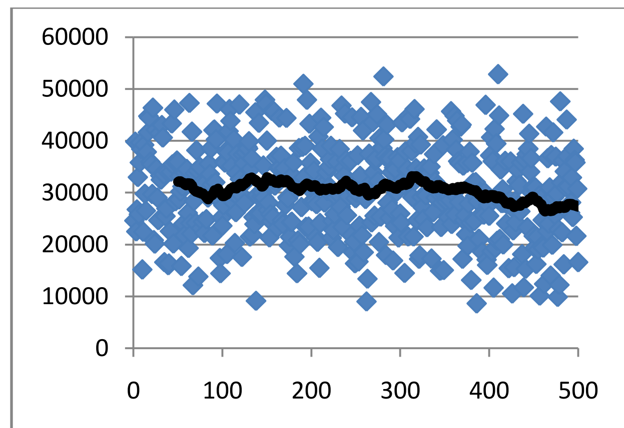

Figure 9. Competitive RL performance over 5 runs of 500 games with width of 50 on moving average .

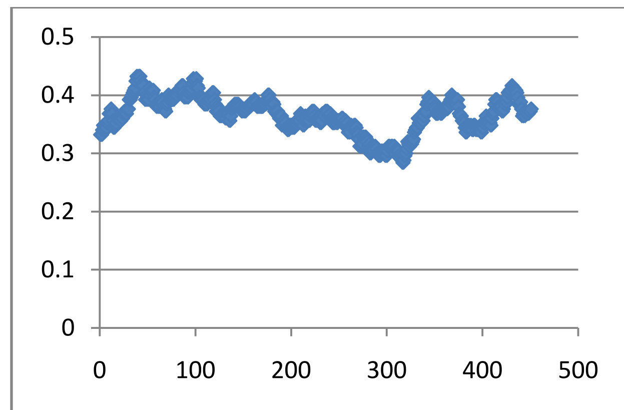

Figure 10. Average percentage of wins for RL vs. Pierre Dellacherie's agent over 5 runs of 500 games.

Now, in terms of learning, its score is clearly not improving, and it degrades from what it initially learns for

the most part. However, the Figure 10 demonstrates that the agent does initially improve its win-rate over Dellacherie's agent. However, its improvement is then consistently struck out by the opponent.

It is significant that the agent does consistently better than Dellacherie's agent in terms of lines efficiency (that is, average score per line). The figure below shows that the RL agent is more efficient than Dellacherie even at its worst. It takes risks, and as a result tends to get more multi-line clears, which means that its score is consistently higher.

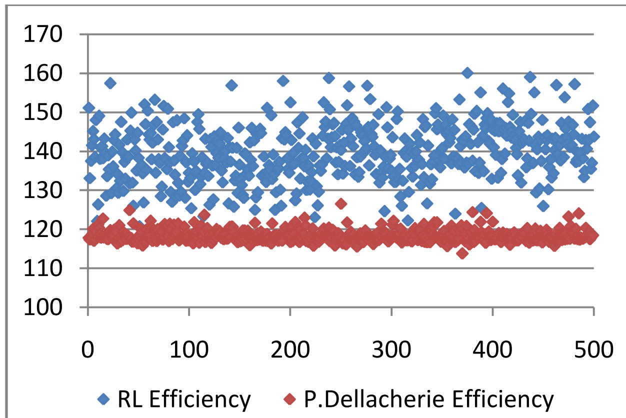

Figure 11. Average score per line for RL agent vs. Pierre Dellacherie's agent during competitive play.

To conclude the results on reinforcement learning, it must be observed that there is one reason that the agent loses so badly and struggles to improve its perceived performance, and this reason can only be observed by watching gameplay. As noted previously, players who clear multiple lines send "junk" to the bottom of their opponent's screen. This junk is not arranged randomly, however. The junk is simply solid blocks with a single-width slot down it. This means that if the opponent can clear its own stack of blocks and get an 'I' piece, it is easy for that opponent to send the lines of junk right back to the original player. This consistently happens to the reinforcement learning agent, because its risky behavior means that it frequently sends junk to Pierre's agent. However, that agent's un-risky behavior means that it is able to clear its own stack of blocks and use the junk it just received against the RL agent, which often cannot respond quickly enough because of its risky handling of its piece sequence.

The trouble, then, is that the agent doesn't know from its performance measure to act less riskily. Its only goal is to get as many points as possible, regardless of how often it dies.

5.4 Neural Network Agent

5.4.1 Low-Level Neural Network

The low-level neural network agent operated on the low-level state space described above, and suffers from all the same problems described for that state space: primarily, that it is incapable of recognizing similar states as being similar. Although presumably the hidden units of the network help to generalize from known input to unknown input, the state space is so large that meaningful training may take a prohibitively large amount of time.

Weights were initialized equally on the network, and as such that the first placement considered was weighted no worse than any other placement, resulting in extremely bad performance, as with the maximize lines agent. Due to the ε-greedy policy for choosing actions, the agent would soon explore random actions and encounter slightly increased rewards.

After training for only a few hundred games, performance of this agent became comparable to that of the random agent. However, even when training was continued for over 6,500 games and 110,000 pieces placed, performance did not improve. Training was repeated several times, with extremely similar results—especially because the agent really only scored one of a few different totals, as can be seen on the graph below.

This agent was approximately as unlikely to clear lines through random actions as was the random agent, and even when it cleared a line randomly, the knowledge it gained was unlikely to prove useful because it would rarely enter this state again.

Presented below is the performance of the low-level neural network over 1,500 games with a moving average of period 100 superimposed. Of particular interest is the learning at the beginning of the training process.

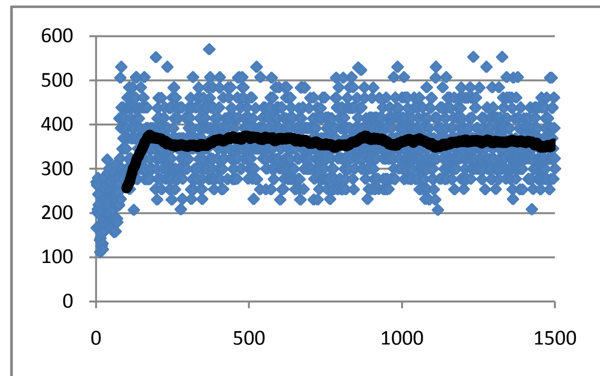

Figure 12. Low-level neural net performance with moving average.

5.4.2 High-Level Neural Network

As described above, the high-level neural network agent operates on the high-level state representation, and chooses between the high-level actions rather than simply rotating and translating pieces. Although this agent does not compare to the agents of Fahey and Dellacherie, it displays learning results clearly linked to the reward strategy.

5.4.2.1 No Line Limit

The high-level neural net agent, like previously discussed agents, was trained in games with no opponent and no line limit. Of particular interest during this training were certain cases where, early in training, the agent received positive rewards for choosing the maximize lines action, and began to choose this action in similar situations. However, when no lines are available to be cleared by the currently failing piece, this action performs extremely badly, and the agent therefore performed badly until exploration caused it to explore other actions.

At this point performance of the high-level neural net began comparable to the performance of the agent that always performs the clear board action. Averaged over five training runs of 500 games, the neural net agent's average score is 51520.4248, as compared to the clear board agent's 56,449,878. However, the neural net agent achieves this score with an average of only 305.312 lines per game, as opposed to the clear board action agent's 466.494.

This suggests that the weighted scoring for multiple lines cleared simultaneously is causing the neural network agent to make use of the build valley actions to score more points per line cleared, rather than simply use clear board action exclusively. The downside of this is that this risky behavior results in a shorter average game (as measured in lines cleared).

Presented below is the performance of the neural network agent averaged over five runs of 500 games with a moving average of period 50.

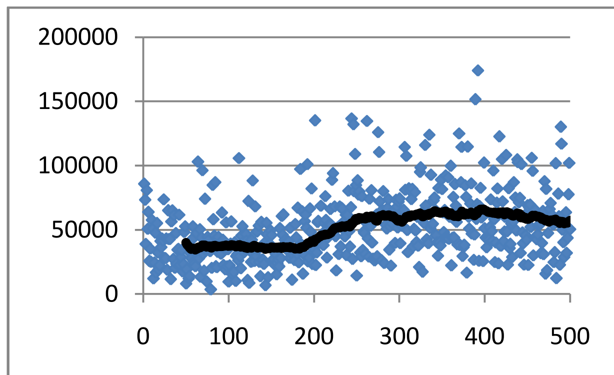

Figure 13. Performance of the high-level neural net agent with moving average

Additionally, presented below is the performance of the high level neural net in a single run of 500 games. Note how the agent first learns an ineffective strategy, due to a large reward gained from an action that is not, on the whole, very effective. The agent then, due to random exploration, learns a new, more effective strategy.

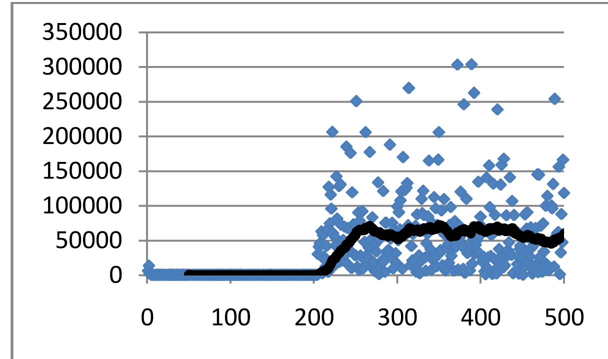

Figure 14. Neural net performance on single run displaying strategy shifts.

5.4.3 Line Limit

The higher average points to lines ratio presented above is a positive sign that the high-level neural net agent may accomplish one of the goals set out in this experiment; that the human experience of being required to maximize points over a limited number of lines (as required by human reflexes) could be replicated through a machine learning agent.

To explore this possibility, the high-level neural net's performance was compared to that of Pierre Dellacherie's manually weighted agent over a maximum number of lines cleared. Here, the maximum number of lines was 300, although if several lines were cleared simultaneously as many as 303 lines could be scored. Dellacherie's agent almost always clears all 300 lines, and as can be seen on the graph below, scores a fairly constant amount of points over these 300 lines.

The high-level neural net, on the other hand, sometimes clears all 300 lines and sometimes fails to clear them. Interestingly, when it succeeds in surviving the entire game, earns more points than Dellacherie's agent. Over five runs of 500 games, the average performance of the neural net is still lower than Dellacherie, earning an average of 34696.5628 points over an average of 216.4608 lines cleared, as compared to Dellacherie's 35,530.504 over 300.134 lines. Note that where Dellacherie's agent earns an average of 118.382 points per line, the high-level neural net earns an average of 160.29 points per line.

Presented below is the performance of the two agents averaged over five runs of 500 games, with moving averages of period 50.

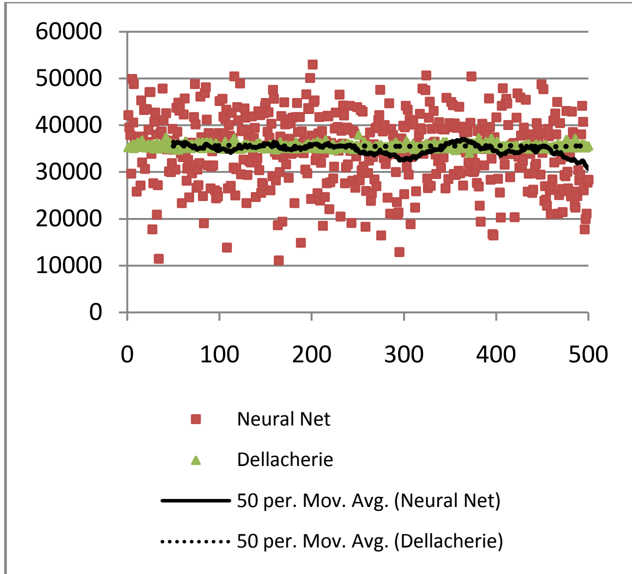

Figure 15. Neural net performance vs. Dellacherie over 300 lines.

5.4.4 Competitive

The remaining question is if the neural nets predisposition toward clearing multiple lines simultaneously means it will be a more effective competitive player. To test this, the agent was played against Dellacherie's agent with the multiplayer rules described above, including garbage lines being sent to the opponent in the event of clearing multiple lines simultaneously.

Although Dellacherie's agent is not specifically designed to compete with others, it nevertheless proved effective at clearing garbage lines sent by the neural net agent. The neural net agent's risky behavior, on the other hand, translated into higher pile heights and many games lost. Over 500 games, the neural net won 29.75% of the games, with the opponent winning the remaining 71.25%. Although it is impossible to definitively say what caused an agent to lose, it seemed to the observer that the neural net lost on account of its own risky behavior more often than because of garbage sent by the opponent.

Over the course of 500 games, the neural net agent averaged 50.422 lines cleared before the game ended and 5980.37 points, whereas the opponent averaged 53.03 lines and 5368.159 points. It is important to recall that the object of competitive play is not to accumulate points but instead to cause the opponent to fail. Presented below is the winning percentage of the neural net agent over the previous ten games, for the 500-game period.

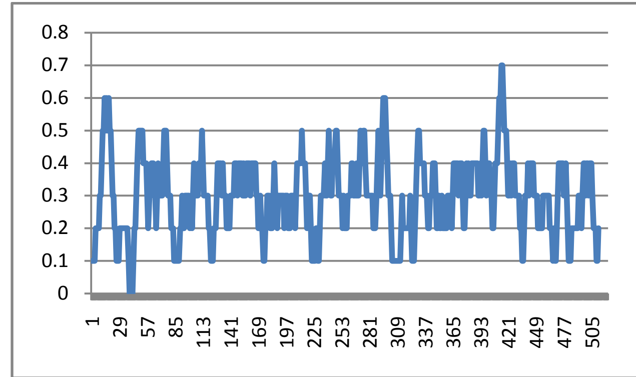

Figure 16. Winning percentage of neural net vs. Dellacherie over previous 10 games.

6. Conclusions

It is clear from examining the results of the agents presented above that Tetris is a very complicated and unstable environment. Results are inconsistent throughout this presentation, and some are particularly discouraging. It is nevertheless clear that the hypothesis that these learning agents, on average, would earn more points per line cleared than the existing heuristic agents is true. It is also clear that the learning agents outperform the agents that have only one high-level action available to them.

Under certain circumstances, such as the neural net agent playing alone, performance seems to improve as a result of additional training. Under other circumstances, such as the same agent under competitive play, this is not true. It may be that in competitive play any strategy change as a result of learning nevertheless results in approximately the same result, due to the efficacy of the opponent's play.

Both of these results seem to confirm, or at least support, the first hypothesis presented in section 2.4. However, these results do not support the second hypothesis that the learning agents would outperform the existing heuristic agents. It is clear that although the agents presented above are capable of automatically learning strategies for the game of Tetris, these strategies cannot compare to the existing carefully and manually tuned strategies in terms of length of survival. Now, as mentioned before, this is not necessarily of interest. The fact that current strategies can survive millions of lines is of no interest when constructing an agent to approximate human experience, which could not conceivably play more than 300 lines in a traditional Tetris game. Compared to human performance, the learning agents appear to perform well: while they can't survive forever, they perform better during game-play.

7. Future Work

There are a number of possible improvements in the area of the state representation and high level actions. The

state could be modified to include information about the potential valleys that are blocked by the top layer of the pile, which is notably absent from the current representation, and as a result the agents operating on this state have no reason not to build on top of holes they should be attempting to uncover. It may also be that the state should be re-hashed in its entirety: a key to the success of a learning agent is a state-space that represents the raw states of the game accurately enough that high-level actions have similar results for each raw state that maps to the same high-level state.

The high-level actions should be improved as well; it is apparent by observing the action of the valley-creating actions that these actions could achieve this goal more efficiently, and without producing so many interior holes and towers in the pile. The high-level actions are so significantly inconsistent that their performance may explain most of the noise in the learning graphs of the learning agents. It would be most interesting to create high-level actions through the use of genetic algorithms that attempt to minimize the inconsistency of actions by fine-tuning the weights of the various components of the evaluation heuristic for each action.

In terms of learning, it might prove productive to penalize agents' rewards for actions that result in loss of the game. Additionally, it may be interesting to explore changing the high-level action less frequently, to allow for more complicated strategies to be carried out across several pieces, potentially modifying high-level actions so that they control play until their objective is achieved.

This project has demonstrated that, in the end, the agents may learn to play Tetris with some limited proficiency given a set of actions, and only reinforcement from the scoring function of the game itself.

References

Bishop, C. M. (2006). Pattern Recognition and Machine Learning. *Information Science and Statistics Series.* Springer, 2006.

Burgiel, H. (1997). How to Lose at Tetris. *The Mathematical Gazette*, Vol. 81, No. 491. (Jul., 1997), pp. 194-200.

Carr, D. (2005). Applying Reinforcement Learning to Tetris. Rhodes University, Grahamstown, South Africa.

Demaine, E.D., S. Hohenberger and D. Liben-Nowel (2002). Tetris is Hard, Even to Approximate. *Technical Report MIT-LCS-TR-865* Laboratory of Computer Science, Massachusetts Institute of Technology.

(Driessens, K., and S. Ďzeroski (2004). Relational Reinforcement Learning. *Machine Learning,* Vol. 57, No. 3 (Dec., 2004), pp. 271-304.

Fahey, C. Tetris AI: Computer Plays Tetris. http://www.colinfahey.com/tetris/tetris.html. Accessed 12-5-2007.

Samuel, A. L. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development, Vol. 3, No. 3.* Reprinted in IBM Journal of Research and Development, Vol. 44 No. 1/2 (January/March 2000).

Siegel, E., and A. D. Chaffee (1996). Genetically Optimizing the Speed of Programs Evolved to Play Tetris. *Advances in Genetic Programming,* Volume 2.

Sutton, R. and A. G. Barto (1998). Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA.

Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Communications of the ACM,* March 1995 / Vol. 38, No. 3.