An Incremental Approach to Creating an Automated Tetris Player Using Expectiminimax Informed By Heuristic Functions Learned by Evolutionary Algorithms

Hayden Johnson, Daniel Chang, Ibrahim Nour

December 2021

1 Abstract

Tetris is a popular video game that is useful for studying sequential decision making under uncertainty. There exist many documented approaches to creating an automated Tetris player using artificial intelligence algorithms. However, unlike with popular turn based games such as chess and go, ensembling AI algorithms is not an approach that is commonly taken with Tetris despite the possibility of greater performance. In this study, we detail and test a possible approach to this ensembling paradigm. This paper explains an incremental formulation of a Tetris agent that uses heuristic functions learned by evolutionary algorithms for use in an expectiminimax search.

2 Introduction

The game of Tetris is one of the most iconic video games of all time. In addition to its general popularity, it has also provided useful benchmarks to study sequential decision making under uncertainty. Despite its simple rules and discrete state space, Tetris has proven to be quite a difficult game to solve. For example, even when the entire sequence of pieces that will be dropped are given ahead of time, determining the set of actions that maximizes the number of rows cleared is NP-complete, proven via polynomial time reduction from the 3-partition problem, a strongly NP-complete problem [EDDLN04]. For this reason, under the current $P \neq NP$ hypothesis, there is no polynomial-time algorithm for creating an optimal automated Tetris player. Because of this, AI methods must be used to approximate such a solution in a computationally feasible way.

There are a multitude of artificial intelligence algorithms that can be applied to Tetris, which we will explore below. Constructing the best algorithm is a decision that requires balancing computational complexity with optimizing performance. As we explore later, there are many artificial intelligence algorithms that are commonly used in Tetris research.

One approach to creating an intelligent agent for playing games is to ensemble different artificial intelligence algorithms.

3 Background

Like for most simple games, Tetris can be automated with two different classes of artificial intelligence algorithms: search algorithms and machine learning algorithms. Within search algorithms, there are two main sub-classes: search algorithms that guarantee optimality, and those that do not. Not necessarily in a hierarchical fashion, search algorithms can also be divided into those that treat the problem as an adversarial game, and those that simply search the environment for a goal state. Within machine learning, there are also two main sub-classes relevant to games: reinforcement learning, and relational reinforcement learning.

3.1 Search Algorithms in Tetris

The goal of search algorithms is to systematically explore the state space of a problem, where a state space is the set of possible states that the environment can be in. [RN10]. To evaluate different search algorithms for Tetris, one must first have a well-defined state-space.

3.1.1 Tetris state space

In the context of games, the purpose of search algorithms is to return a move or a decision that allows a player to follow a path that leads to a terminal state of the highest possible value. In games, as Russel *et al.* have defined, "terminal-states" are states that indicate the end of the game. An objective function is used to assign a value to each terminal state.

In the case of Tetris, an obvious definition of a state would be a description of the position of all the tetrominoes on the game board. From this definition, a terminal state would thus be a valid set of tetromino positions that contains at least one tetromino that exceeds the maximum height of the board. A possible simple objective function could be defined as the number of rows cleared that have been cleared in the entire game, at any state.

3.1.2 Game Representation

One question that arises when attempting to analyze search algorithms for Tetris, is whether or not to use an algorithm for adversarial games.

Although Tetris is a single player game, Tetris can be modeled as an adversarial game, and automated with the minimax algorithm, as described by Rovatsou *et al.* [RL10]. Briefly, this would entail creating a pseudo-adversary, treating the randomly generated piece sequence as the opponent. Using the minimax like procedure described by Rovatsou *et al.*, the randomly generated piece at each turn would be selected to be the piece that would lead to the worst performance by the automated player.

Alternatively, Tetris can be treated as a simple search problem, i.e., one that does not involve an adversary. For example, if the piece sequence is determined ahead of time, the problem can be solved with an uninformed search algorithm, or with a heuristic search.

The most realistic approach is probably one that treats Tetris as a stochastic game, as Tetris neither generates the piece that is selected to be the piece that hurts the player most, nor does Tetris give the piece sequence ahead of time. Instead, Tetris generates the next tetromino type during runtime from a uniform distribution. As Lamanosa *et al.* describes, this type of search can be made to be extremely efficient via parallel search [LLM+13].

3.1.3 Search Algorithms with Optimality Guarantee

Search algorithms that are guaranteed to return the optimal solution are exhaustive, meaning that they explore the entire search space before returning a decision.

In complex games, this is not feasible, because exhaustive search typically is an exponential time algorithm, since the search space is often exponential in its input size [RN10]. Specific to Tetris, the branching factor is extremely high. Even though there are only five available pieces to use, there are hundreds of possible positions and rotations available for each piece.

For example, algorithms such as regular minimax must explore the entire state-space in order to return a solution. Even with alpha-beta pruning and move ordering implemented, the algorithm still has an exponential runtime. In the case of a known sequence of tetromino pieces, uniformed searches like depth/breadth first search run into the same issue. Analogously, in the case of a stochastic piece generator, one can introduce chance nodes to use expectiminimax [RN10], a minimax like algorithm that returns the move that results in the highest expected value. However, no matter how you traverse the state-space, it is exponential in nature.

Due to the aforementioned reasons, it would be computationally infeasible to attempt to reach a terminal state with an exhaustive search. However, as we will discuss below, modifications to exhaustive search can and have been made in order to allow them to be useful.

3.1.4 Search Algorithms without Optimality Guarantee

Using slight modifications, we can still utilize algorithms such as mini-max and breadth or depth first search. This can be done with heavy use of heuristics.

For example, with minimax, we can provide a maximum depth to the search and perform heuristic search [RN10]. States or positions at a maximum, prechosen, depth will be treated as terminal states. Their values, instead of being the measure of performance that the user desires to optimize such as the number of rows cleared, will instead be an estimate of the desired value that will occur later on in the subtree of the state-space, if the automated player decides to chose that path.

In an almost identical fashion, this can be applied to the case of Tetris with a predetermined sequence of tetrominoes. Using an uninformed search algorithm, the state-space

search can be cut off at a certain depth from the root, and evaluating the state at the cut-off with the heuristic. In the same way, this can be applied to the last case, Tetris with a stochastic piece generator. Expectimax can be modified to also use a cutoff depth and heuristic search.

Heuristic search in Tetris has many different possible implementations. For example, as described by Bohm *et al.*, a possible heuristic for a state could be defined as a linear function of the highest row occupied, number of cleared rows, number of holes, and number of occupied cells. [BKM04]

Another approach is to approximate the optimal solution by taking advantage the convergent nature of stochasticity.

Monte Carlo tree search (MCTS) is similar in purpose as heuristic minimax search. However, instead of defining an explicit heuristic function for max-depth states, they are evaluated via simulations, dictated by a playout policy [BPW+12]. With a playout policy that sufficiently approximates a "good" player, as the number of simulations (playouts) increases, the evaluation becomes more accurate.

Genetic algorithms have the same goal as MCTS. They typically explore the state-space by representing a possible solution with a (bit)string, evaluate how well each possible solution performs with a heuristic, or fitness, function, and generate successor states via a selection process that mimics biological reproduction and natural selection [For93]. One important feature of genetic algorithms is that they have a lot of hyperparameters that need to be tuned in order to find the best algorithm. To name a few, genetic algorithms require a mixing number, or the number of parents that join to create a new solution, the recombination procedure, or the way that parents' "DNA" will combine to create a child, and a mutation rate, the rate at which random bits in each child solution gets changed [RN10].

3.2 Machine learning in Tetris

Machine learning is a subset of Artificial Intelligence in which a computer uses a dataset to build a model and uses the model as a hypothesis to make future predictions on unseen examples [RN10]. In other words, machine learning algorithms give computers the ability to learn and make future predictions without being explicitly programmed to do so.

3.2.1 Reinforcement Learning

Reinforcement learning is a subset of machine learning where an agent is situated in an environment and learns to take actions that maximize a reward signal. Generally, this situation is formalized as a Markov Decision Process. In order for an agent to improve its performance in an MDP, the agent must learn a policy (i.e. a rule for determining its own actions) based on the expected reward. In an MDP, finding an optimal policy equates to finding a solution to the Bellman equation. Two primary approaches for accomplishing this are value iteration and policy iteration.

In value iteration, the agent tries to estimate the value at every state and derives a policy based on the set of estimated values. In policy iteration, the agent directly tries to find the

policy that maximizes the reward. RL algorithms can further be categorized based on their treatment of the environment. In some cases, algorithms will attempt to learn an model of the the environment in order to improve its policy – these are referred to as "model-based" algorithms. In other cases, the algorithm will learn actions with out learning a model for the environment – these are referred to as "model-free" algorithms.

Reinforcement learning has some problems. One example of the problems of reinforcement learning is the fact that RL works best in small domains [Car05]. For example, Carr *et al.* reports that it is easy to create agents that plays games like Tic-Tac-Toe or Blackjack. Early on, Boyan *et al.* considered the application of Q-learning to packet routing called "Q-routing" algorithm. The algorithm had no information about geometry and traffic of the network, and the network did not have routing center and yet was able to find an excellent routing routing strategy in a changing network [BL94]. Another example of problems of RL is that the dimension of the problem increases exponentially as the input increases which is known as the "curse of dimensionality" as demonstrated by Boyan *et al.*.

Due to the large state space of Tetris, reinforcement learning works well because we can learn a policy or value function rather than representing the entire space of potential states. Early attempts along these lines often implemented forms of dynamic programming with either policy or value iteration.

In 1996, Lambda policy iteration was used to clear 2800 lines [BT95]. In 2002, an implementation of least-squares policy iteration was able to clear an average of 3000 lines[LPL02]. In the same year, a policy gradient method was able to clear 6800 lines[Kak02], and in 2006 a linear programming solver applied directly to the Bellman equation cleared 4500 lines[FVR06].

However, it wasn't until 2013 that a reinforcement learning method was able to contend with evolutionary methods. That year, Gabillon *et al* developed an agent based on classification based policy iteration that was able to clear 51,000,000 lines[GGS13].

3.2.2 Deep Reinforcement Learning

Deep reinforcement learning combines classic reinforcement learning with modern deep neural networks. Neural networks are powerful function approximaters and can be used with reinforcement learning to approximate the state/value function when the state-space is too large. Notably, this method allows reinforcement learning agents to learn from raw input rather than a set of hand crafted features. This was first demonstrated in 2013 when a team at Google Deepmind used deep Q-learning to play Atari games on raw inputs[MKS$^+$13]. Since then, deep RL has been used extensively in space of game-playing producing state of the art results in games such as Chess[SHS$^+$18], Go[SHM$^+$16], and Starcraft[VBC$^+$19].

The first attempt to apply deep neural networks to the game of Tetris was in 2015. At this time, Stevens and Pradhan were able to build a Tetris agent based on Q-learning, but found it was unable to learn from raw input and did not perform as well as state of the art methods[SP16]. They argue that the reason for this was that there is a substantial delay between actions and reward signals. This is consistent with prior work that found deep Q-learning works far better for reflex based games (where the effect of actions are immediately

realized) rather than games with long term dependencies. The difficultly of modeling Tetris with neural networks was described in another work that same year [LB15]. Since that time, deep reinforcement learning has become vastly more power in its ability to deal with large and difficult environments [SHS+18][SHM+16][VBC+19]. However, most have these more sophisticated algorithms have yet to see published results for the game of Tetris.

3.2.3 Relational Reinforcement Learning

Relational reinforcement learning (RRL) is a machine learning technique that unites the standard Q-learning algorithm and relational regression algorithm, and RRL agents need to learn a policy to decide what actions to perform because agents does not know the impact of their actions [RD04]. RRL are useful for planning tasks, bioinformatics, and natural language processing. One of the examples of planning tasks is stacking blocks where each block has a variable name such as x or y. As stated by Džeroski *et al.*, using variable names allows us to achieve general learning plans that do not depend on details of the current data, and therefore, learning a plan that stacks block x onto block y would transfer to one that stacks block z onto block w. [DDRD01].

Relational reinforcement learning has a few advantages over reinforcement learning (RL). For example, consider the case where we have a problem to solve and we train an agent to achieve the goal that solves the problem. If we later change the goal in a way that does not affect the logical backbone of the solution, a RRL agent does not need to be retrained, whereas a RL agent needs to be retrained. Furthermore, in reinforcement learning, it is possible to reuse the outcome of learning in a simple domain when learning in a complex domain, i.e., learning to stack 5 blocks given that we previously stacked 2, 3, or 4 blocks, [DDRD01].

Relational reinforcement learning can be used to train agents that can play Tetris, but some restrictions have to be imposed on the game of the Tetris because the stochasticity and chaos of Tetris make the learning process using RRL difficult. For example, as stated by Ramon and Driessens, it is extremely difficult to map a Q-value to a state or action [RD04]. Another way we can use RRL for Tetris is considering specific tasks of Tetris such as determining where to put the next piece of tetrominoes, given that we know the shape of the piece. In this case, all we need to find is the best orientation and location of next falling piece of the tetrominoes. Knowing the best orientation and location of the next falling piece make the learning process easier because it addresses the stochastic nature of tetromino piece generation [RD04].

4 Problem Representation and Algorithms

4.1 Objectives

As we have discussed, the stochastic nature and long-term dependencies of Tetris make it a difficult benchmark for computational agents. Agents that have preformed well are often constructed from many components including hand-crafted features, aggregation functions,

and search strategies. The function of these components are often left unexplained in the academic literature and can leave the reader puzzled over the selection of the author.

In order to better understand how these models are constructed, we look to decompose them into simple components and build a complex agent incrementally. Along the way, we will provide explanations on what benefits each method provides.

Specifically, we look to develop an agent in three stages. First, we build a greedy agent based on the common feature sets for Tetris, using a naive heuristic function. Secondly, we will explore how genetic algorithms can be used to learn optimal weights for an improved heuristic function – as a linear function of these simple features. Lastly, we will explore the benefits of using our improved heuristic function in an expectiminimax search.

4.2 Tetris Representation and Environment

As mentioned previously, creating an appropriate state space representation is paramount to the creation of any intelligent agent.

Tetris consists of placing tetrominoes, which are seven differently shaped objects, on a game board. During each move, a tetromino is randomly selected by the game from a uniform distribution. The player must then decide the rotation and horizontal position to place the tetromino, as the vertical position is dependent on these values and the board state.

Abstractly, we defined a state as a matrix of Boolean values, representing whether or not a piece on the board was occupied or not, along with a tetromino piece. More formally, since the Tetris board has 20 rows and 10 columns and seven possible tetrominoes:

$$\forall i State_i \in \{0,1\}^{20\times 10} \times \{1-7\} \tag{1}$$

Accordingly, the initial state was defined as a zeros matrix of the previously defined size, along with a randomly selected tetromino.

$$State_0 = ([0]_{20x10}, t \in \{1-7\}) \tag{2}$$

A terminal state test would simply test whether or not the piece is unable to fit on the board with any rotation.

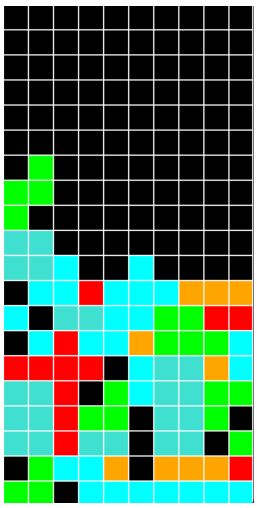
It follows that the state transition model is defined as the set of all possible states that can be achieved given a board configuration and a current piece. A state transition simply represents a possible move, per the usual convention for game search-spaces. Therefore, since there are 10 possible horizontal positions, and a maximum of 4 possible piece rotations for each horizontal position, there are forty possible state transitions.

Formally, we can assume we have a function that returns the board position after placing a piece at an x position with a specific rotation (along with a random tetromino).

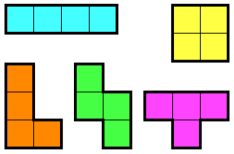
$$next\_state((board, piece, position, rotation)) \tag{3}$$

Finally, we can define a function that returns the set of all possible successor states given a board and a piece.

$$next\_states((board, piece)) = \bigcup_{x,r} next\_state(board, piece, x, r) \tag{4}$$



(a) A single state of the Tetris environment. Colored squares represent a "1" value in the current state matrix, while black squares represent a "0" value



(b) The seven possible tetromino shapes.

Figure 1: Pygame Tetris environment

4.3 Naïve Greedy Search

It is intractable to explore all possible states reachable from the current state. If we consider all possible tetromino shapes that may be selected, the branching factor is $4 * 10 * 7 = 280$. If we wish to generate all states that are less 50 moves away from the current state, we would need to traverse 280^{50} nodes, which is greater than the estimated number of atoms in the observable universe. The simplest possible way to systematically explore the state space is to explore all possible states reachable from the current state with a single move. We are essentially performing a depth limited search, with a maximum depth of one.

This method is tractable, as for each given board configuration and given piece, there are a maximum of 40 possible states that can be reached within one move. After exploring all boards resulting from all possible moves, we evaluate each board using a heuristic function calculated from a collection of features, and greedily choose the highest scoring board state. The downside to this method is that its success is heavily dependent on the quality of the heuristics chosen, as much of the state space is never explored. Fortunately, many heuristics have been developed for evaluating a single board state. In the context of game AI searchs, heuristics are functions that take as input features of a state/position and return an estimate on how successful the state is.

Feature Set	Features
Bohm et. al	Max height, holes, removed lines, well depth, altitude difference, well sums, blocks, row/column transitions
Bertsekas et. al	Aggregate height, holes, bumpiness, cleared lines
Lagoudakis et. al	Max height, holes, bumpiness, mean height, removed lines
Dellacherie et. al	Max height, holes, landing height, cummulative wells

Table 1: Tetris Feature Sets

For the purpose of this analysis, we will test the following feature sets.

We can define a heuristic function as a weighted sum, or linear function, of all these variables.

$$h(n) = \begin{bmatrix} max\_height \\ \#holes \\ ... \\ removed\_rows \\ row\_transitions \end{bmatrix}^\top \begin{bmatrix} w_1 \\ w_2 \\ ... \\ w_3 \\ w_4 \end{bmatrix} \tag{5}$$

In short, the greedy search involves generating all possible board states from all possible horizontal positions and rotations of the current piece. The algorithm then evaluates each generated board state with a heuristic function and greedily chooses the move associated with the highest scoring board position.

This heuristic function is simply a linear function of a set of features of the Tetris board. However, the introduction of a linear function gives rise to another problem that commonly haunts AI and machine learning algorithms: a large hyper-parameter search-space. If we score the board states using the dot product between the vector of features and a weight vector, the weight vector must be either predefined or learned.

For the time being, we can predefine weights so that weights that are associated with features that indicate a "good" board position, like the number of removed rows, will be assigned a +1 weight, and weights associated with features that indicate a "bad" position, such as the maximum height and number of holes, will be assigned a -1 weight.

$$h(n) = \begin{bmatrix} max\_height \\ \#holes \\ ... \\ removed\_rows \\ row\_transitions \end{bmatrix}^\top \begin{bmatrix} 1 \\ -1 \\ ... \\ 1 \\ 1 \end{bmatrix} \tag{6}$$

Obviously, this approach is naïve and has many flaws. While the weights have the correct

sign, they do not take into account the importance of these features. Having a large number of holes may be more detrimental to a player than having a large number of row transitions. In the next section, we will address this issue by systematically exploring the hyper-parameter space of the weight vector using a genetic algorithm. By doing so, we will drastically improve our heuristic function and thus improve our agent by many orders of magnitude.

4.4 Genetic Algorithms

In the previous section, we saw how we can build a naive agent that selects actions based on greedy searches with a simple rating function. Additionally, we saw that we can define an aggregation function as the dot product of our features and a weight vector. This a good start in constructing our agent, however, if we want to improve our performance we need a strategy for finding the optimal values of our weight vector.

In order to do this, we will use a simple genetic algorithm. As previous described, genetic algorithms are a strategy for optimization inspired by natural selection (figure 2). The algorithm begins by spawning a population of agents, each of which has a genotype i.e. a vector that will be used as our weights in our aggregation function. In the first epoch, the weight will be generated from a uniform distribution on the interval [-1, 1]. After the population has been constructed, the fitness of each agent is computed as the arithmetic mean of a performance measure on a series of Tetris games.

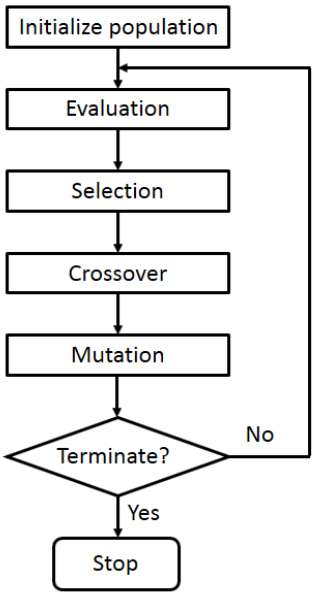


Figure 2: Evolution

We then move on to the repopulation phase where we construct the next generation of agents. For repopulation, we begin by directly replicating our best preforming agents (without mutation) into the next generation. This ensure that our best preforming genotypes are retained. We then select a percentage of our top preforming agents (based on survival rate) to serve as parents for crossover. We fill out the rest of our population by completing crossover (with mutation) on our pool of parents. To complete crossover, we begin by randomly selecting pairs of parents from the parents pool. Then, for each weight in their genotypes, we randomly select one of the parents weights to add to a newly constructed genotype. In order to improve crossover, we weight our sample so the probability of selecting from a parent's genotype corresponds to the proportion of the relative fitness between the parents, where relative fitness is the proportion of the agents fitness over the accumulated fitness of all agents. Once the new genotype is complete, mutation is applied and an agent with the new genotype is added to the next generation. For mutation, we simply scale each each weight be a value from a normal distribution with mean 1 and low standard deviation.

While the procedure may seem complicated, the intuition is quite simple. By creating new agents with genotype based on the best agents of the previous generation, fitness should

improve across epochs. Importantly, we should note that successful convergence of genetic algorithm is highly dependent on the hyper-parameters for training. For instance we have: survival rate, population size, number of epochs, standard deviation of mutation distribution, crossover procedure, and many more. Often, finding the hyper-parameters that preform well is more a matter of trial and error rather than analytic considerations.

4.5 Expectiminimax

The final stage in constructing our agent is to use the optimal weights learned with the genetic algorithm to guide a tree search, expectiminimax. Although before this stage we have constructed an agent that uses a heuristic function with (near) optimal weights, at the end of the day, we are still greedily choosing a move, which may not be a globally optimal solution.

To mitigate this issue, we can take approximate the "real" game/state tree of Tetris by increasing our depth limit in our depth limited search to two. We can generate all possible moves by the agent, and treat the game as a pseudo-adversary, which aims to maximize our cost function. In the end, the agent still minimizes our cost function, i.e. minimax.

However, this approach brings another problem. Tetris is a stochastic game: the next tetromino piece is not known ahead of time. Therefore, in order to implement the pseudo-adversary in a realistic way, we must iterate through all possible tetromino pieces that can be generated by the game, and calculate the expected cost. In other words, we are implementing expectiminimax, treating the randomly generated piece as a chance node.

5 Methods and Results

5.1 Environment and Algorithm Implementation Details

For the purpose of this analysis, we constructed a Tetris environment using the python programming language and the Pygame python module.

The greedy algorithm, genetic algorithm, and tree search were all implemented by us using the python programming language using the Tetris environment.

Tetromino pieces were generated from a uniform random distribution. While traditional Tetris allows a one piece look ahead mechanism, our implementation does not include this mechanism for simplicity. Additionally, our environment does not allow for fancy Tetris moves such as the T-spin. In other words, the agent is only able to place a piece by deciding a piece rotation and x-position, and dropping the piece vertically.

5.2 Metrics and Data Collection

The performance of our agents is measured by two main values. The first is the number of pieces placed, and the second is the number of rows cleared. While the number of rows cleared generally sets the standard for Tetris performance, we found that assessing performance

based on number of pieces placed, and representing lines cleared as a rating function, helped with performance. This is especially true in the case of genetic algorithms. The reason for this, we believe, is that agents tend to learn to that clearing lines is an important feature fairly quickly, but learning to place pieces in an efficient way is more complicated.

The Tetris environment that we have created directly computes the number of pieces placed and the number of rows cleared in any game.

5.3 Naïve Greedy Search Results

To estimate the performance of each feature set using the naïve greedy search, we ran one hundred trials for each feature set using the naïve weights and recorded the statistics for the number of pieces placed and number of rows cleared.

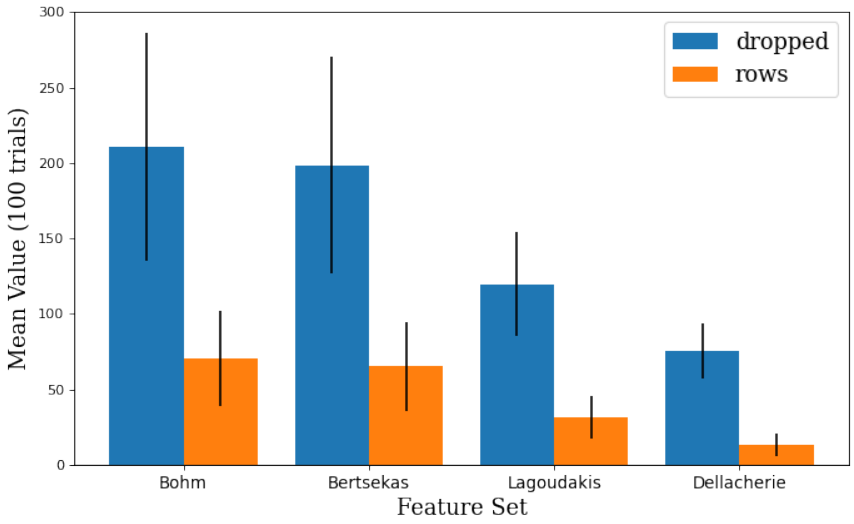


Figure 3: Results of naïve greedy search after 100 trials

The Bohm feature set performed the best, with an average of 211 pieces dropped and 70 rows cleared in each game. Because the Bohm feature set performed the best with naive weights, we selected this feature set for our final heuristic function. Using a genetic algorithm to improve our heuristic function weights, we can drastically increase these metrics.

5.4 Genetic Algorithm Results

Genetic algorithms are very computationally expensive. In order to train a genetic algorithm, you must run many epochs with many agents, each with a large number of trials. Furthermore, each game takes longer to terminate as agents improve. Optimal Tetris agents can take hours or even days to finish a single game, depending on the hardware. Due to limited time and computational resources, we imposed some limitation on training. We ran

relatively small populations over a relatively small number of epochs. We also did not run exploratory trials to optimally tune the hyperparameters – which, as we mentioned, requires significant trial and error. Because of this, the performance of our agents is substantially below what is possible for this method. Nevertheless, we believe our results are still illustrative of the power that genetic algorithms hold and serve their intended purpose in our project.

To test our genetic algorithm, we ran a series of 15 epochs with 20 agents and a survival rate of .35 (more hyper-parameters can be seen in table 2).

parameter	**value**
epochs	15
population size	20
trials	3
survival rate	.35
elite	.2

(a) training parameters

group	**lines**
population	3681
elite agents	9010
top agents	9688

(b) final epoch fitness

group	**fitness**
population	1830
elite agents	4453
top agents	4390

(c) final epoch lines cleared

Table 2: parameters and results

Fitness of our agents was measured by the average number of pieces placed in a series of 3 games, with the top 15 percent of agents classified as 'elite' and directly copied into the next generation. We collected data each epoch recording average fitness and average genotype of 3 groups: the top performing agent, the group of elite agents, and the entire population. Performance for each group continued throughout training – the average performance value of the last epoch can be seen is show in table 2. Additionally, the fitness and average genotypes were graphed in figure 4.

After obtaining the optimal weight vector using the genetic algorithm, we created a final heuristic function using the feature set of Bohm *et al.* and the learned weight vector. Using this final heuristic function, we ran our greedy search algorithm for 100 more trials. As shown in fig. 5, the greedy search algorithm vastly outperforms the naive ones. On average, our genetically learned greedy search algorithm places 5621 pieces per game, and clears 2330 rows, outperforming the naive algorithms by nearly 30 times.

5.5 Expectiminimax Results

To analyze the performance of expectiminimax, we ran 10 trials using our expectiminimax algorithm with our genetically learned heuristic function. However, due to the large branching factor, it is not feasible to repeatedly calculate all game states at a depth of 2.

To solve this, we developed a sampling scheme for each position. For each possible move by the agent, we sampled a subset of the possible game states resulting from any randomly generated tetromino, at any rotation and horizontal position. From this subset, an expected cost was calculated. We varied the size of this subset to quantify the positive effects of observing deeper states on the performance of our agent. As shown in figure 6, the expectiminimax agent is able to out perform the greedy agent by a factor of three. With

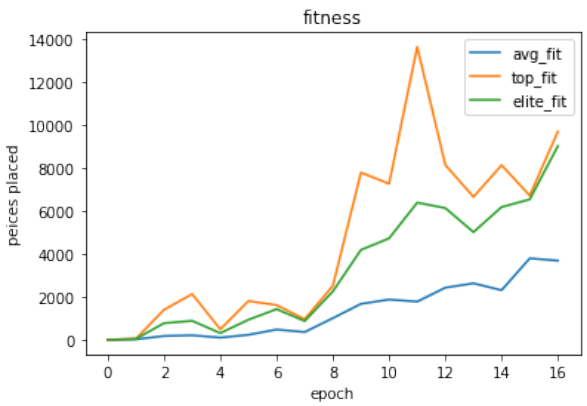


figure 4a

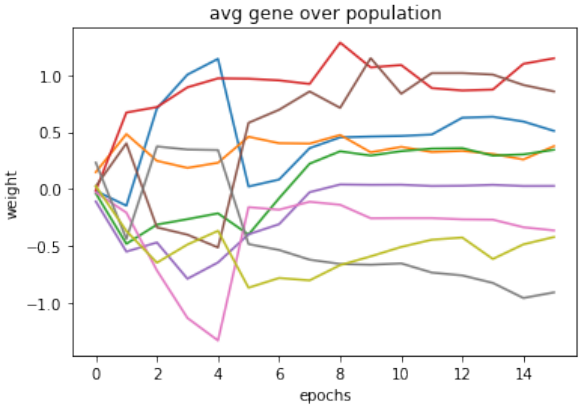


figure 4b

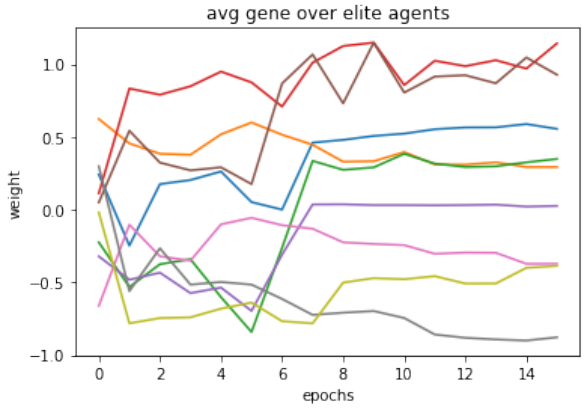


figure 4c

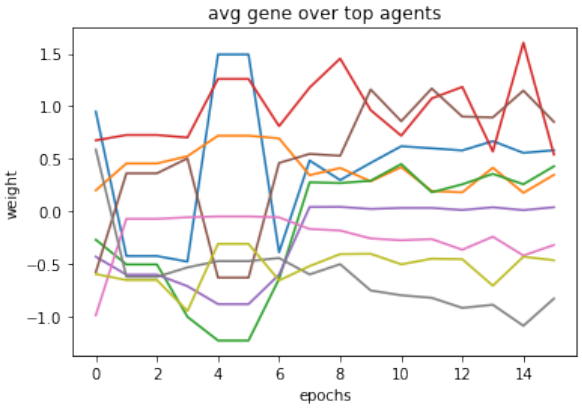


figure 4d

Figure 4: Genetic Algorithm Training

only sampling 30 states of depth two, our final agent was able to, on average, place 16136 blocks and clear 6705 lines.

6 Discussion

6.1 Greedy Search

The results of the naive greedy search shows that the Bohm feature set outperforms the other feature sets. To understand why this is case, let us consider features in Bohm feature set and how they are different from the other features. The first thing we notice is that the Bohm feature set contains more features than other feature sets, and thus is richer than other feature sets. The quality of the heuristic function highly depends on the collections of features that the heuristic uses. Therefore the heuristic function that used the Bohm feature set dropped more pieces as well as cleared more rows than other heuristic functions.

This result is important because it suggests that number of features in the feature set

is important. That is, a heuristic function that uses feature sets with a smaller number of features will not be as efficient as a heuristic functions that uses a feature set with many features. This result, however, does not tell us what an ideal number of features in a feature set is.

6.2 Genetic Algorithm

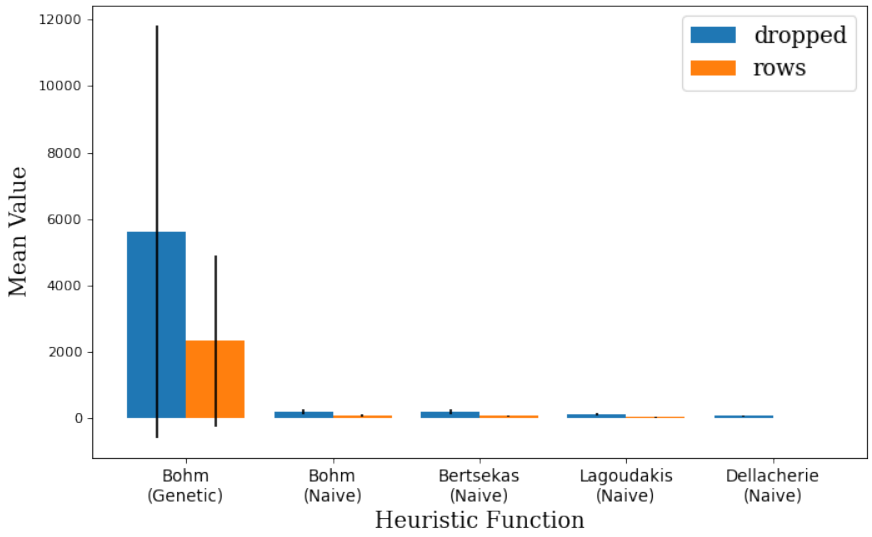


Figure 5: Comparison between the performance of naive heuristic functions naive and our heuristic function with (near) optimal weights, learned via genetic algorithm

As seen in our results, we successfully trained a genetic algorithm to improve the greedy search algorithm introduced in section 4.3. In our final epoch, the average fitness of our population reached 3681, and our top preforming agent achieved a fitness of 9688. These results greatly outperform our naive approach which saw a maximum score of 200. This is consistent with our prediction that allowing the agent to optimize the weights in its aggregation function would improve its performance. Since the inputs to our aggregation function are a set of hand crafted features, the weight vector is essentially a notion for value for each feature. The importance of a feature corresponds to its magnitude and whether or not its desirable corresponds to its sign.

Because we encoded the weights vectors of our agents as their genotype, we graphed the average genotype of our agents across epochs (figure 4b-d). This gives us a notion for how each weight changed over time and contributed to overall fitness. As we can see, each weight varies quite a bit in the early epoch but stabilizes as time goes on. This makes sense because variability tend to do down once a local optimum is reached. This corresponds to the notion of convergence, where each generation stays within a small area of the feature space and does not explore.

Due to our limited number of epochs, it is unclear whether or not our algorithms has converged or not. However, we can can see that variability has diminished. Also, we see that the performance between the top performing agent and the group of elite agents has become very similar. Both of these qualities indicate that may be approach convergence on a local optima. If we wanted to avoid finding a local optima at this point we would need to expand the search space by increasing the amount of diversity in the population. We could do that in many ways including increasing the size of population, increasing the survival rate, and increasing the magnitude of mutations.

6.3 Expectiminimax

As shown in the slight improvements with respect to the simple greedy heuristic search, when a near optimal heuristic is paired with a deeper search, the agent has a larger view of the state space and thus is able to be more informed. Due to hardware and time limitations,

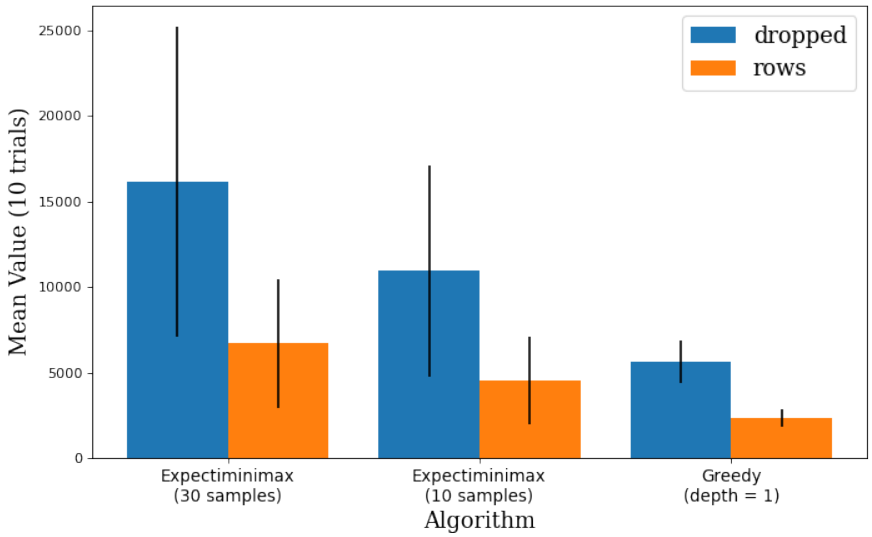


Figure 6: Comparison between the performance of the final greedy algorithm with the performance of the two expectiminimax algorithms. The number of samples refers to the size of the subset of second move possible positions when calculating expectiminimax

a full expectiminimax of depth 2 was not feasible for this project. However, seeing the promising results with the limited sampling, it is clear that this strategy of implementing chance nodes is fruitful. Further increasing the sampling number, or abandoning sampling altogether and traversing all possible depth 2 nodes, would most likely increase performance substantially.

7 Conclusion

Training game-playing agents is fundamental in the advancement of artificial intelligence. Tetris agents that have done well in the literature often have components such as hand-crafted features, aggregation functions, and search strategies. Often, the intent of these components is unexplained, which make it difficult to understand what is going on.

Moreover, ensembling different classes of artificial intelligence algorithms is an approach that is often taken, with great success, by researchers in turn based games like go and chess. However, this approach is not seen in abundance in Tetris research. For this reason, we decided to implement this approach using by incrementally adding complexity to an agent by testing different approaches and strategically increasing its ability to explore the Tetris state space.

In the first phase, we considered all the possible states in the board that can be reached from the current state. Specifically, we considered all the possible states that can be reached from the current state with a single move, which is a depth limited search with a depth limit of one, and then we evaluated each board using a heuristic function and chose the board that has the highest value. The heuristic function we used was the inner product of the features (e.g. maximum height and number of holes) and a weight vector. However, these weights were not optimal. They were predefined and did not reflect the importance of each feature.

In the second phase, we used a genetic algorithm to learn optimal weights in the heuristic function we used in phase one. The algorithm first produced a set of agents where the genotype of the agents is the weights in the heuristic function. At the beginning of the genetic algorithm, we initialized the weights with a uniform distribution on the interval [-1, 1] and calculated the fitness of each agent. From there, we ran 16 epochs and produced new agents with genotypes based on the best agents of the previous generation.

In the third phase, we built an agent that uses the heuristic function learned by the genetic algorithm from the previous phase to perform an expectiminimax search of depth 2. This approach was tricky because due to the stochasticity of Tetris, the next piece cannot be predicted, so we needed to go over all possible pieces and calculate the expected cost.

Using the number of of pieces placed and the number of rows cleared, we were able to asses the performance of the agents in the three stages. One of the things we found was that measuring the performance of agents using the number of pieces placed and representing lines cleared as a rating function improved the performance of the agents. When using the naïve greedy algorithm to compare how the different feature sets scored in terms of number of pieces dropped and number of rows cleared, we found that the Bohm feature scored the highest points. On average, the Bohm feature dropped 211 pieces and cleared 70 rows in each game, and therefore we used the Bohm features as the weights for the heuristic function. After using a genetic algorithm to improve the heuristic function, the result we reached was impressive. The newly trained agents were able to drop more than 5500 pieces and clear approximately 2000 rows. The result we achieved using the genetic algorithm is nearly 30 times higher than the score we achieved using the naïve algorithm.

Our final agent was able to on average place 16136 blocks and clear 6705 lines. Incrementally building a Tetris agent by ensembling different artificial intelligence agents has

given insight into the power of exploring a state space to approximate solutions to otherwise intractable problems.

8 Suggestions for Future Work

There are many ways that future work could expand on what we have done. The most obvious direction would be to optimally tune the hyper-parameters in to the genetic algorithm with longer training and increased computing power. Additionally, one might try to eliminate the need for hand-crafted features. One possible change to our heuristic function/genetic algorithm is adding the exponential aggregation function shown in Bohm et al. Another approach would be to train a neural network to extract feature automatically. Furthermore, additional non-linear aggregation functions could be explored. This too could be done with neural networks, but the issues that have been expressed regarding long term dependencies would need to be addressed. One possible solution to this would be to use a combination of monte carlo tree search along with deep reinforcement learning as seen in Silver et al. [SHS+18].

9 Contributions

Team Member	Parts
Daniel Chang	Parts 1, 2, 3.1, 4.2-3, 4.5, 5.1, 5.2, 5.3, 5.5, 6.3
Ibrahim Nour	Part 3.2, 6.1, 7
Hayden Johnson	Part 2, 3.2, 4.1, 4.4, 5.2, 5.4, 5.4, 6.2, 8

Table 3: Team members and delegated parts

References

[BKM04] Niko Böhm, Gabriella Kókai, and Stefan Mandl. Evolving a heuristic function for the game of tetris. pages 118–122, 01 2004.

[BL94] Justin A Boyan and Michael L Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. In *Advances in neural information processing systems*, pages 671–678, 1994.

[BPW+12] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods.

IEEE Transactions on Computational Intelligence and AI in Games, 4(1):1–43, 2012.

[BT95] Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Proceedings of 1995 34th IEEE conference on decision and control*, volume 1, pages 560–564. IEEE, 1995.

[Car05] Donald Carr. Applying reinforcement learning to tetris. Technical report, 2005.

[DDRD01] Sašo Džeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. *Machine learning*, 43(1):7–52, 2001.

[EDDLN04] Susan Hohenberger Erik D. Demaine and David Liben-Nowell. Tetris is hard, even to approximate. *International Journal of Computational Geometry & Applications*, 14(01n02):41–68, 2004.

[For93] Stephanie Forest. Genetic algorithms: Principles of natural selection applied to computation. *Science*, 261(5123):872–878, 1993.

[FVR06] Vivek F Farias and Benjamin Van Roy. Tetris: A study of randomized constraint sampling. In *Probabilistic and randomized methods for design under uncertainty*, pages 189–201. Springer, 2006.

[GGS13] Victor Gabillon, Mohammad Ghavamzadeh, and Bruno Scherrer. Approximate dynamic programming finally performs well in the game of tetris. In *Neural Information Processing Systems (NIPS) 2013*, 2013.

[Kak02] S Kakade. A natural policy gradient. advances in neural informa tion processing systems 14, 2002.

[LB15] Ian J Lewis and Sebastian L Beswick. Generalisation over details: the unsuitability of supervised backpropagation networks for tetris. *Advances in Artificial Neural Systems*, 2015, 2015.

[LLM+13] Rigie Lamanosa, Kheiffer Lim, Ivan Manarang, Ria Sagum, and Maria-Eriela Vitug. Expectimax enhancement through parallel search for non-deterministic games. *International Journal of Future Computer and Communication*, pages 466–470, 01 2013.

[LPL02] Michail G Lagoudakis, Ronald Parr, and Michael L Littman. Least-squares methods in reinforcement learning for control. In *Hellenic conference on artificial intelligence*, pages 249–260. Springer, 2002.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[RD04] Jan Ramon and Kurt Driessens. On the numeric stability of gaussian processes regression for relational reinforcement learning. In *ICML-2004 Workshop on Relational Reinforcement Learning*, pages 10–14, 2004.

[RL10] Maria Rovatsou and Michail G. Lagoudakis. Minimax search and reinforcement learning for adversarial tetris. In Stasinos Konstantopoulos, Stavros Perantonis, Vangelis Karkaletsis, Constantine D. Spyropoulos, and George Vouros, editors, *Artificial Intelligence: Theories, Models and Applications*, pages 417–422, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[RN10] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[SHS+18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[SP16] Matt Stevens and Sabeek Pradhan. Playing tetris with deep reinforcement learning, 2016.

[VBC+19] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.