Applying Reinforcement Learning to Competitive Tetris

Max Bodoia (mbodoia@stanford.edu)
1618 Sand Hill Rd
Palo Alto, CA 98498

Arjun Puranik (apuranik@stanford.edu)
1618 Sand Hill Rd
Palo Alto, CA 98498

Introduction

For our project, we attempt to apply reinforcement learning to the game of Tetris. The game is played on a board with 20 rows and 10 columns, and each turn a player drops one of seven pieces (the seven tetrominoes) onto the field. Points are gained by dropping pieces in such a way that all ten squares in some line (row) are filled, at which point the line is cleared. Multiple lines may be cleared with a single piece drop and will result in more points being earned. A player loses if the height of the pieces on his board exceeds the height of the board itself.

The fully observable nature of the Tetris board and the simple probabilistic transitions from state to state (i.e. adding a randomly selected piece to the end of the piece queue each turn) naturally suggest the use of reinforcement learning for Tetris. Specifically, Tetris can be modeled as a Markov Decision Process. However, the state space of Tetris is extremely large - the number of ways to fill in a 20×10 board is 2^{200}, and the Tetris requirement that no row be completely filled only reduces this to $(2^{10} - 1)^{20}$. As a result, the complete MDP for Tetris is entirely intractable. In our paper, we will outline the different approaches we took to dealing with this intractability. We begin by describing our initial, unsuccessful attempts, and commenting on the reasons why they may have failed. Next, we describe the approach that eventually suceeded - fitted value iteration - and give a detailed analysis of its results. Finally, we conclude by considering the strengths and weaknesses of our implementation of fitted value iteration, and highlight other potential approaches that we did not try.

Initial Attempts

In this section we will outline our early approaches to the problem. In each case we describe the motivations for the approach, the extent to which it was successful, and the reasons why it was ultimately unsuccessful.

Block Drop

As an initial pass at the problem of Tetris, we chose to implement a simple game which we will call Block Drop. Block Drop is comparable to Tetris in that players drop pieces onto a rectangular grid, and gain points for filling up an entire row (clearing a line). The main difference is that in Block Drop, the only type of piece available to players is a single unit square. Although the relationships between states in Block Drop are much more simple than in Tetris (and thus the optimal policy is much easier to learn), the size of the state space for Block Drop and Tetris are similar. In this sense, the tractability of Block Drop can be used as a rough lower bound on the tractability of Tetris.

We wrote a program to formulate an $m \times n$ game of Block Drop as an MDP. Note that a valid Block Drop gameboard never has any holes (where a certain grid square is not filled but a higher up square in the same column is filled). This allows us to represent each state by a set of n values corresponding to the height of the pieces in each of the n columns. These column heights can take on values from 0 to $m-1$ independently of each other (columns heights are bounded by $m-1$ since the game ends if a column reaches height m). We also added a single terminal state to represent losing the game, so the total state space of Block Drop is $m^n + 1$. One possible action exists for each column (since the player chooses which column to drop a piece in) so there are a total of n actions. Transition probabilities are completely deterministic - dropping a piece in a particular column either increments that column's value by 1, or decrements the value of every column by 1 (in the case when a line is cleared). Our reward function assigns a reward of 1 for clearing a line and a cost of m for losing the game; all basic moves had 0 cost.

In order to solve the Block Drop MDP, we used the ZMDP package written by Trey Smith (Smith & Simmons, 2012). This package is designed primarily for finding (approximate) solutions to Partially Observable MDPs, and as a result it requires that the input problem be described as a POMDP rather than an MDP. Unfortunately, we were unable to find any other software that works for pure MDPs, so we decided to convert our problem into a POMDP by adding a single observation that is always made. We assumed that this modification would not affect the tractability of the problem significantly.

We found that the solution to Block Drop matched the intuitively obvious optimal policy, and provided a good sanity check for our use of MDPs to model the problem. The resulting policy drops a single block into each column (from left to right) until a line is cleared, at which point it repeats. However, the Block Drop problem also illustrated the tractability issues that we faced: 4×4 Block Drop took less than a second to solve, but 5×4 Block Drop took 16 seconds and 6×4 Block Drop took over a minute; all sizes above 7×5 were unable to complete a single solver round. Figure 1 shows the time in seconds required for a solution to Block Drop as a

function of the board size (in terms of total number of grid squares). These results made it clear that the actual game of Tetris would be intractable without significant approximations; our next task was to decide how to make these approximations.

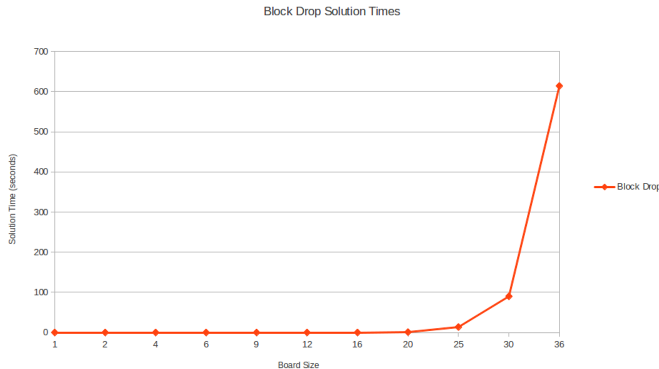

Figure 1: Solution times for Block Drop

No-Holes Tetris

In Tetris, unlike in Block Drop, we cannot represent the states of the board by a single height for each column. It is possible that holes - where a column has an empty block below the highest filled block - are formed in the board when pieces are dropped. In fact, the management of these holes is one of the more difficult aspects of the game. However, we decided to approximate board states in this manner by emulating the state representations for Block Drop and tracking only the height of the highest block in each column. We refer to the version of Tetris that results from this approximation as No-Holes Tetris. The number of possible boards for No-Holes Tetris is significantly smaller than for Tetris (m^n compared to around 2^{mn}) and was the main motivation for making this approximation. We attempted to counterbalance the inaccuracy of this simplification by also modifying the reward function of No-Holes Tetris, so that the player loses points upon taking actions which produce holes. This causes the optimal policy for No-Holes Tetris to avoid actions that lead it to states that differ from the corresponding true Tetris state. Furthermore, we observed that competitive Tetris players almost never leave holes in their stacks. As a result we hoped that the optimal policy for our approximate No-Holes Tetris would remain close to the optimal policy of true Tetris.

We then wrote a program to formulate an $m \times n$ game of No-Holes Tetris as an MDP. As noted above, each board of No-Holes Tetris is represented by a vector of n values between 0 and $m-1$, representing the heights of the blocks in each column. In addition, each state of No-Holes Tetris also includes the next piece available to the player. For p possible pieces, this gives a total state space of $p \cdot m^n + 1$ (including the terminal state). The set of actions available to the player at each state is the set of ways (including rotations) that the given piece can be dropped onto the board. Transition probabilities are deterministic to the extent that they affect the board and uniformly random to the extent that they affect the piece, so each state/action pair leads uniformly to one of p possible next states. The reward function assigns a cost of m to losing the game, and the reward for a given state/action pair is the number of lines cleared minus the number of holes created.

We found (unsurprisingly) that No-Holes Tetris is significantly more difficult to solve than Block Drop. 4×4 No-Holes Tetris took around 20 seconds to solve, while on larger boards our solver was unable to find good policies. The best policies found had regret values greater than 0.5 on a 5×4 board and greater than 4 on a 6×4 board, even when the solver is allowed to run for long periods. We tried varying the number of pieces to reduce complexity and found that the difficulty of the problem scaled extremely quickly with the number of pieces. No-Holes Tetris with only one piece could be solved about as quickly as Block Drop, but attempts to solve 6×4 No-Holes Tetris with two pieces were unable to reduce regret below 1. Figure 2 shows the solution times of No-Holes Tetris as a function of board size when different total numbers of pieces are used.

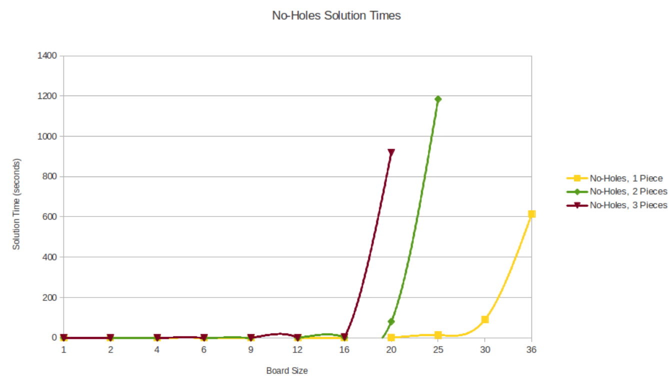

Figure 2: Solution times for No-Holes Tetris

These results were disappointing, to say the least. We knew from our analysis of Block Drop that solving No-Holes Tetris would be intractable for board sizes above 7×5. However, our hope was that we could find a "good enough" policy for the full 7-piece No-Holes Tetris on a "large enough" board. This policy would serve as a starting point for local piece placement that we could use to build policies for larger boards. We planned to do this by dividing larger boards into smaller sub-boards and choosing actions based on some function of the sub-board policies. Unfortunately, No-Holes Tetris proved to be too computationally difficult to find suitable sub-board policies. On the one hand, the 4×4 board was small enough that the outputted policy for No-Holes Tetris was determined primarily by height constraints, and did not generalize well to larger boards. On the other hand, the best 5×4 policy performed poorly even on the 5×4 board. This left us with no good starting point from which to construct a policy

for larger boards, and no obvious way of applying our results effectively to the general problem of playing Tetris.

Factored MDPs

We briefly considered representing Tetris (or No-Holes Tetris) as a factored MDP. Some of the existing literature on MDP and POMDP solution algorithms explores ways of optimizing the solution process by representing the state space in factored form (Guestrin, Koller, Parr, & Venkataraman, 2003), (Poupart, 2005). That is, we would represent each state in terms of some finite number of state variables, and describe the transition probabilities and rewards as functions of these variables instead of defining a particular value for every state. Tetris, like most games, is highly factorable and therefore well suited to this kind of approach. Possible factorizations of an $m \times n$ board include using a binary variable for each of the mn grid squares (which encodes the full version of Tetris), and representing each of the n columns as a variable that takes m possible values (which encodes No-Holes Tetris).

Ultimately, however, we decided not to pursue this approach. The benefits of representing MDPs in factored form come primarily from the ability to represent particular transition probabilities and rewards as functions of strict subsets of the full set of state variables. In other words, in order for a factored MDP to be more efficiently solvable than its unfactored counterpart, the state variables of the factored form must be independent to some extent. At first, No-Holes Tetris seems to exhibit a great deal of independence: each time a piece is dropped, at most four state variables (i.e. columns) will change in value, and these changes will depend only on the values of the other three state variables. However, the fact that state transitions must also take into account line clearances eliminates this independence. Line clearances only occur when a particular row is filled in at every column, and as a result, the transition probabilities for each state variable depend on the value of all n state variables. For this reason, we judged that the use of a factored representation would not provide significant reductions in tractability and decided against this approach.

Fitted Value Iteration

The next approach that we tried (and the one that ultimately proved most successful) was fitted value iteration. The basic principle behind fitted value iteration is to choose a small set of features and represent each state in terms of this feature set. In this respect, it is reminiscent of the factoring approach. Importantly, however, the state space of a factored MDP is necessarily the same size as the original state space, while the set of possible combinations of feature values may be much smaller. Furthermore, fitted value iteration does not consider this full set of possible feature values, but instead restricts itself to the feature representations of a sampled set of states. This means that the tractability of fitted value iteration depends on the number of samples rather than the size of the state space.

In general, two basic properties must hold for a particular MDP in order for fitted value iteration to be effective. First, it must be possible to approximate the true value of a particular state using only a small set of information about that state. This ensures that the "featurization" of the MDP is reasonably representative of the original. Second, the number of samples from the original state space needed to represent the relationship between state values and state features must be relatively small. This ensures that an accurate function from features to values can be learned using the sampled set of states. Intuitively, Tetris seems to satisfy both properties, so the application of fitted value iteration is a natural choice.

Our algorithm is roughly identical to the fitted value iteration algorithm presented at the end of the Reinforcement Learning handout, with a few small caveats. It begins by sampling a set of states S_s from the state space S and initializing a parameter vector θ to zero. Then, it alternates between two steps. In the first step it calculates, for each state, the maximum over all actions of the expected utility of being in that state:

$$y_s = \max_{a \in A} R(s) + \gamma E_{s' \sim P_{sa}}[V(s')] \text{ for each } s \in S_s.$$

In the second step, it fits the values of the parameters to this set of maximum expected utilities:

$$\theta := \arg\min_\theta \frac{1}{2} \sum_{s \in S_s} (\theta^T \phi(s) - y_s)^2.$$

Here, A is the set of all actions, $R(s)$ is the reward received from being in state s, P_{sa} is the distribution over possible transition states resulting from taking action a in state s, and $\phi(s)$ is the vector of features values for state s.

The main difference between our algorithm and the one presented in the handout is that instead of sampling from the state space as a whole, we take samples by randomly placing pieces around the board. This seems reasonable, since a truly random Tetris board is unlikely to look anything like the kinds of boards found in Tetris gameplay, and the set of states that can be generated using this sampling process is identical to the set of states it is possible for our agent to encounter during test time. The other difference is that because we already know the transition probabilites in Tetris, we can compute $R(s) + \gamma E_{s' \sim P_{sa}}[V(s')]$ directly instead of having to sample it.

Once the algorithm converges, the parameter settings can be used by an agent to play Tetris. The agent chooses moves in a manner similar to the first step of the algorithm. When in state s, it chooses action a' according to:

$$a' = \arg\max_{a \in A} E_{s' \sim P_{sa}}[V(s')].$$

Feature Selection

The fitted value iteration algorithm given above assumes the existence of a function ϕ from states to feature values. A full implementation therefore requires the choice of a feature set and the definition of ϕ. Note that ϕ can be any conceivable function $\phi : S \to \mathbb{R}^k$ for some $k \in \mathbb{N}$, where k is the number of features. This means that the "feature space" of possible functions ϕ is both enormous and difficult to define. Although techniques exist for automating the process of feature selection, it is far more common to define features manually using

domain knowledge (Hall, 1999).

We considered a variety of possible features based on our personal experience playing Tetris. The first kind of features we considered were "summary" features that are functions of the whole board. The most straightforward summary features we examined were Max-Height, the maximum height of all the columns, and Num-Holes, the total number of holes on the board (where a "hole" is an unfilled grid square for which there exists a filled grid square higher up in the same column). Other more complicated features included: Avg-Diff, the average of the absolute values of differences between adjacent columns; Max-Diff, the maximum of these absolute values; and Num-Covers, the total number of covers on the board (where a "cover" is a filled grid square for which there exists an unfiled grid square lower down in the same column). For each proposed feature, we conducted preliminary tests of the feature by running our algorithm using only this feature and a constant feature with value 1 for every state. We then measured the average lifespan (i.e. number of moves made before losing) for an agent that plays using the learned parameters and compared it to the average lifespan of an agent that places pieces randomly. Only the features Max-Height, Num-Holes, and Num-Covers led to better than random performance.

In addition to the summary features, we also considered sets of non-summary features that were functions of particular columns. These feature sets corresponded to different summary features: examples include i-Col-Height, the height of the ith column, and i-Col-Diff, the absolute value of the difference between the ith and $(i-1)$th columns. However, when we tested each of these potential feature sets individually, we found that an agent playing with the learned parameters performed no better than random play. This is most likely because the relationships between these feature sets and the true value function are non-linear. Since our algorithm finds parameter settings using linear regression, it does not have the potential to learn non-linear relationships well and typically ends up converging to arbitrary parameter values. In theory, the parameter update step could use a wide variety of machine learning algorithms, and using a more complex regression algorithm could allow it to learn more intricate relationships between the features and the state values. Without such modifications, however, our algorithm cannot perform well using the non-summary feature sets considered and so we ultimately chose not to consider them further.

Results

After this preliminary testing, we were left with three features - Max-Height, Num-Holes, and Num-Covers - that seemed promising. We then conducted a series of more rigorous tests to determine which combinations of features led to the best performance. For each possible combination of these three features, we ran our algorithm to convergence 5 times and tested an agent for each of the resulting parameter vectors. The algorithm used 1000 sampled states and a discount factor of 0.9 on a full 20×10, 7-piece board, and iterated until the maximum difference between corresponding parameter values from one iteration to the next dropped below 0.01. In testing, each agent played 100 games and the average number of moves per game was computed. Table 1 shows the average of the parameter vectors and average lifespans over all 5 agents for each possible feature set. The first value in each parameter vector is the value of the constant feature, followed by the values of the parameters corresponding to the other features in the order listed.

Table 1: Average agent lifespans and parameter vectors

Feature Set	Avg Lifespan	Parameter Vector
Constant	25.8	(-13.8)
Max-Height	56.8	(21.8, -2.8)
Num-Holes	47.7	(17.6, -2.5)
Num-Covers	40.5	(12.6, -1.0)
MH, NH	126.0	(8.2, -0.4, -0.7)
MH, NC	72.1	(63.1, -3.7, -0.2)
NH, NC	42.0	(-15.1, -5.5, -0.1)
MH, NH, NC	129.4	(-17.0, -1.0, -0.9, -0.1)

Max-Height seems to be the most informative feature since it gives the best single-feature performance and the two feature pairs that use it perform better than the third. Num-Holes likely comes in second given the stark difference between performance with Max-Height, Num-Holes and with Max-Height, Num-Covers. Num-Covers has the worst individual performance and it makes a difference only when combined with Max-Height, so it seems to be the least informative of the three. These rankings seem to make intuitive sense since looking at the maximum column height and total number of holes on a Tetris player's board is one of the most obvious ways to judge whether they are doing well.

On the other hand, however, the differences in performance given by these three features may not be due to inherent differences in their informativeness about the true value function, but rather a product of the random sampling process we used. In general, random play clears lines rarely but loses often, and thus the set of sampled states contains many more examples of terminal states than of line clearances. Since Max-Height is especially indicative of whether the player is near a terminal state, while the other two features relate more to how easily lines can be cleared, it is possible that the correct value of Max-Height is simply the easiest to learn from random play. To test this hypothesis, we conducted another set of trials for the three-feature set using a modified version of the algorithm with multiple training rounds. In the first round, we learn parameter settings for the three features using a set of states sampled from random play. In subsequent rounds, the set of states is sampled from play using the parameter settings from the previous round, with a probability of 0.3 to choose a random action. However, this algorithm showed no significant change in parameters from round to round (even though subsequent rounds reinitialized the pa-

rameters to 0 after collecting the samples). This suggests that the parameter settings for the three-feature set given in Table 1 are likely the optimal values achievable by our algorithm.

The last method we tried for improving the effectiveness of our agent was to allow it to preview the upcoming two pieces and add a 2-step look-ahead feature. Most versions of Tetris allow players to preview several upcoming pieces, so it seemed fair to allow our agent the same privilege. During training, our agent learns a parameter vector for the three-feature set as it did before. In gameplay, however, it maximizes the expected value of the next state over all possible combinations of actions with the current piece and the two previewed pieces. In other words, if P_{sa_1,s_2a_2,s_3a_3} is the transition distribution over next states after taking actions a_1, a_2, and a_3 in s_1, s_2, and s_3 respectively, the agent chooses an action a according to:

$$a = \arg\max_{a_1 \in A} R(s_2) + R(s_3) + E_{s' \sim P_{s_1a_1,s_2a_2,s_3a_3}}[V(s')].$$

We found that this 2-step lookahead agent was a highly effective Tetris player - over the course of several thousand moves, it never lost once and rarely entered a state with a maximum column height above half of the total board height. To ensure that this performance was influenced by our parameter learning (as opposed to merely the lookahead), we tested lookahead agents that had no learned parameters and simply chose actions based on the maximum reward attained during lookahead. In this case, 2- and 3-step lookahead agents performed no better than random play and agents with larger lookaheads took dozens of seconds to choose moves. These results demonstrate how agents that utilize both low-order lookahead and parameter learning can significantly outperform agents which use only one or the other.

Conclusions

We began this project with the hopes of creating an agent capable of playing competitive Tetris against a human player. Very quickly on, however, we realized that we had underestimated the difficulty of basic, single-player Tetris. The beast of exponential size reared its ugly head, and it became clear that we could not simply encode the game as an MDP and expect it to be tractably solvable. Even reducing the board size by a factor of eight and ignoring holes was not enough to allow for the computation of an optimal policy, and the tricky nature of line clearances prevented us from taking advantage of independence between factored state variables.

This forced us to change tactics and use a less straightforward approach to find an approximate solution to the MDP. The fitted value iteration algorithm was a natural choice, since Tetris boards are well characterized by a small set of features that can be learned with relatively few sampled states. After some experimentation, we found three features - Max-Height, Num-Holes, and Num-Covers - that seemed to be roughly linearly correlated with the "true" value of Tetris states. An agent that learned parameter values corresponding to these features was able to survive for an average of around 130 moves. This was not a trivial feat: a random agent only survives for an average of 25 moves, and without the use of previewed pieces - a staple of almost all versions of Tetris - humans without plenty of experience would likely not perform much better. Furthermore, the addition of two previewed pieces and a 2-step lookahead allowed our agent to stay alive indefinitely.

This final instantiation of our agent is particularly interesting because of the way it combines fitted value iteration and simple search and ends up performing better than with either technique individually. In this case, the paradigm of uniting machine learning and conventional algorithms proved to be extremely effective. Future versions of our agent could likely be made even more effective if we incorporated other standalone techniques as well. The linear regression step could easily be replaced by a more advanced regression technique that would allow our agent to learn non-linear relationships between the features and state values, and automated techniques for feature selection might be able to find relationships that are not readily apparent to human players. Nevertheless, our final agent is a reasonably effective Tetris player as it stands, and could provide a solid foundation if we wished to extend our agent to the competitive Tetris arena.

References

Guestrin, C., Koller, D., Parr, R., & Venkataraman, S. (2003). Efficient solution algorithms for factored mdps. *J. Artif. Intell. Res. (JAIR)*, *19*, 399–468.

Hall, M. (1999). *Correlation-based feature selection for machine learning*. Unpublished doctoral dissertation, The University of Waikato.

Poupart, P. (2005). *Exploiting structure to efficiently solve large scale partially observable markov decision processes*. Unpublished doctoral dissertation, Citeseer.

Smith, T., & Simmons, R. (2012). Point-based pomdp algorithms: Improved analysis and implementation. *arXiv preprint arXiv:1207.1412*.