Tetris

Mishari Alarfaj
Forrest Rogers-Marcovitz
Bryan Wagenknecht

February 3, 2009
Adaptive Control and Reinforcement Learning

Introduction

Tetris was created in 1985 by Alexey Pajitnov, and since then it is one of the most popular computer games. Despite its simple rules, playing the game well requires a complex strategy and lots of practice. Furthermore, finding the optimal strategy is NP-hard even if the sequence of tetrominoes is known in advance.

We created an algorithm to autonomously play tetris based of rules of the RL-competition: pieces drop straight down and score is based off the number of cleared rows before the last piece goes above the top row. The game is played on a board with ten columns and twenty rows. Each board combination was scored by a set of features and corresponding weights. We will first describe our algorithm that determines the next move that maximizes the board value. Next, the machine learning technique used to adjust the weights for maximal rows cleared will be explained. Lastly we we present results and a conclusion.

Basic Algorithm

Rather than trying to learn a policy for the entire state space of ~2^{200} possible board arrangements, our Tetris player employs an algorithm to choose its next move based on an evaluation of nine different features on the board. Ideally, the algorithm would evaluate the state of every possible game board resulting from the current piece and all possible future pieces, for each legal move. This strategy would lead to potentially infinite expansions, so the search must be restricted for feasibility. Our implementation enumerates board states only to a depth of *d* steps into the future. The search time is further reduced, by pruning this tree search before each new expansion, so that only the futures of the *b* leaves with the best scores for any given round are enumerated and evaluated. Once the set of best possible moves are evaluated, the position and orientation of the current piece that branches to the highest mean score *d* steps in the future is selected. The piece is place accordingly, a new next piece is randomly generated by the game, and the decision process starts over again.

The board state evaluation features are an attempt to characterize any given board's potential for leading to infinitely many lines cleared in the future. Intuitively, a board with fewer holes (empty cells covered by filled cells above) or a board with lower maximum block height should be more favorable for continued player success. This intuition was fleshed out using a set of nine features borrowed from Böhm, Kokai, and Mandl [1] and are as follows:

1. Pile height: the row of the highest occupied cell in the board.
2. Holes: the number of empty cells that have at least one occupied cell above them.
3. Removed lines: the number of lines that were cleared in the last step to get to the current board.
4. Altitude difference: the difference between the highest occupied and lowest empty cell that are directly accessible from the top.
5. Maximum well depth: the depth of the deepest well (depression of width one) on the board.

6. Sum of all wells: sum of the depths of all wells on the board.
7. Weighted blocks: the number of occupied cells on the board, weighted by height so that a block in row *n* counts *n*-times as much as a block in row one (counting rows from bottom to top).
8. Row transitions: the sum of all occupied-to-unoccupied transitions on the board as it is scanned horizontally by row. The outside to the left and right of the board is counted as occupied.
9. Column transitions: the sum of all occupied-to-unoccupied transitions on the board as it is scanned vertically by column. The outside below the board is counted as occupied.

Each feature establishes a quantitative feature score f_i for the board. The overall score V for a given board state is then taken as the weighted sum of each feature score:

$$V = \sum w_i \cdot f_i \qquad (1)$$

Some features are given negative weights to penalize the creation of undesirable board characteristics, and some are given positive weights to encourage good characteristics. Given the proper weighting values for all features, the board state with the highest overall score V should have the greatest chance for long-term game success. The challenge after having identified these scoring features is to learn proper weights.

Learning the Weights

Our program's learning procedure consists of a numerical gradient descent algorithm which attempts to learn the optimal weights for the evaluator's features. The algorithm starts with our initial feature set, and generates a new set by selecting one of the nine feature weights at random and increasing it's magnitude by 10%. Two games with an identical set of pieces are then run, one using the original weights and one with the new modified set. After the conclusion of both games, a new "initial" set is created using gradient descent as follows:

$$w_{t+1}^{i} \leftarrow w_{t}^{i} + \epsilon \frac{\delta L}{\delta w_{t}^{i}} \qquad (2)$$

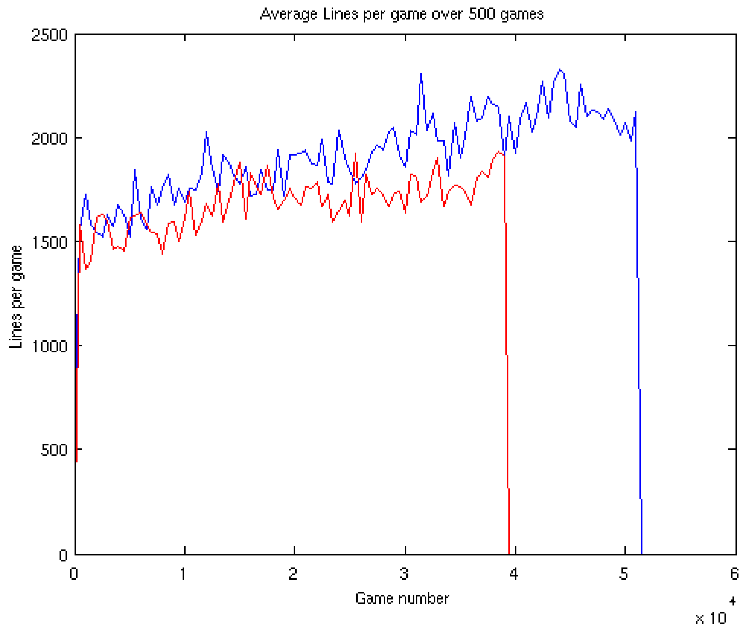
where L is the number of lines generated by a game, w_i is the i^{th} weight of the feature set used, and ϵ is a very small learning rate. Since the true gradient of the value function is unavailable to us, we use numerical differentiation between the two games as follows:

$$\frac{\delta L}{\delta w^{i}} \approx \frac{L_1 - L_c}{w_1^{i} - w_c^{i}} \qquad (3)$$

Subscript *1* and *c* represent the games using the initial weight set and modified weight set respectively.

The feature weights were initially hand-tuned by trial and error to achieve a set that could play games with around 1500 lines cleared on average. This hand-tuned set is given below in Table 1. The hand-tuning was necessary for this learning method to be successful. Because of the stochastic nature of the game, it is difficult to distinguish between a poor weighting set leading to a bad game or an unlucky string of tetrominoes leading to a bad game independent of the weighting set. As a result, the update rule given above may not actually shift the weights in a truely favorable direction after every iteration. In order to avoid over-reacting in the wrong direction, the learning rate should be sufficiently slow to allow the weights to drift in the right direction in the long run. The consequences of a small learning rate is that the weights will not drift very quickly and little to no improvements would be seen without starting with a feasible set.

This algorithm has shown to improve the per-game average of our player, creating weights capable of completing games that average with over 2000 lines (a 50% improvement). The results of two independent learning runs starting from the same initial hand-tuned weighting set are shown below. The plot shows the average number of lines achieved for each group of 500 consecutive games throughout the learning process. The zero-point plotted at the end of each run (the steep dive in the trendline) is an artifact of data analysis and does not represent a 500 game average score.

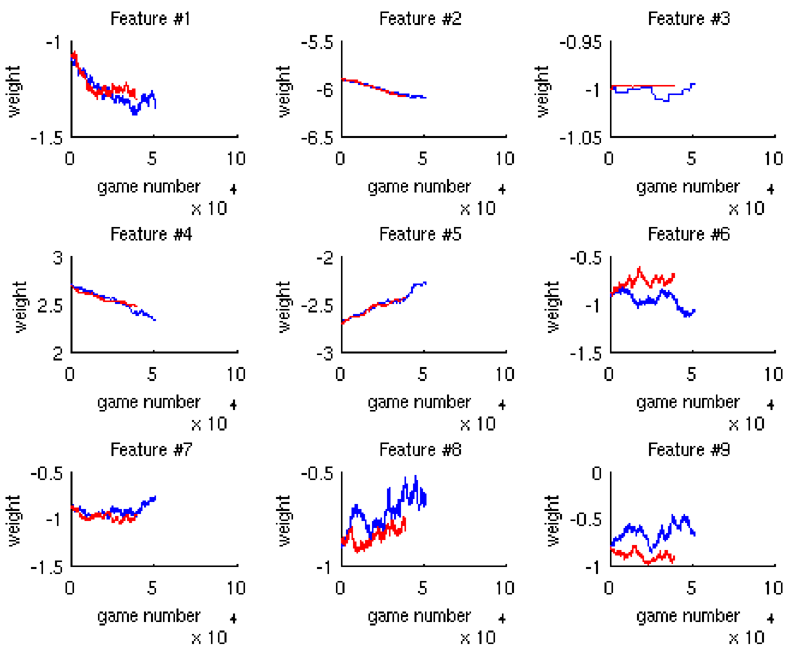


Both learning runs made improvements but did not appear to converge or plateau. The learning runs were halted so that game testing could begin with the most optimized weighting set available. Games were initiated using the final learned rates from the longest learning run (shown in blue above). These final weights are given in Table 1 below.

Table 1: Initial and learned feature weights

Feature #	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Initial weights	-1.1	-5.9	-1.0	2.7	-2.7	-0.9	-0.9	-0.9	-0.85
Final weights (from blue run)	-1.34824	-6.09312	-0.99576	2.32891	-2.29118	-1.06840	-0.79428	-0.62157	-0.65409

The trends over time for each of the weighting factors are plotted below for both learning runs. The results show that both learning runs adjusted the weights similarly for most of the features. Features 6, 8, and 9 showed some discrepancy between learning runs, which could indicate that they either have little effect on the outcome of a game, or that the initial value was already nearly at a local minimum and that the weights started to drift from there semi-randomly. It should be noted that none of the feature weights seemed to converge, indicating that a local maximum had not yet been reached when learning was halted. Better scores could perhaps be obtained by extending the duration of the learning runs, or using a larger learning rate.



Results and Discussion

To balance performance and computational speed, our algorithm predicted two moves into the future while pruning the tree to the best 5 moves from each piece. After staring a number of algorithms to run overnight, it was discovered that a null-pointer exception occurred when the board height was near the top, but legal moves still existed. This caused the games to prematurely end when there was a good chance that future moves would have reduced the board's height. The results of these 13 games can be see in the following table. The mean of the games was 188,294 lines removed with a top score of 417,214 lines.

Table 2 - Games ended due to error:

18139	97588	288552
25701	216745	304598
60834	241197	324746
69922	271008	417214
83122		

A simple fix was made, and new games started. At the time of writing none of the 5 games have ended, with the top game over 500,000. It is most likely that at least one of these games will go over a million lines cleared. Further learning should allow the algorithm to be run with only a single move look ahead, vastly improving computational speed while still clearing over a million lines. Different approaches to learning, such as genetic algorithm, simulated annealing, simplex, etc might find weights that represent a "global" optimal set rather than the (most likely) local optimal set the current weights have settled on.

References

[1] Bohm, Niko, Gabriella Kokai, and Stefan Mandl. "An Evolutionary Approach to Tetris." Proc. of MIC2005: The Sixth Metaheuristics International Conference, Vienna, Austria. 2005.