Learning to play Tetris applying reinforcement learning methods

Alexander Groß, Jan Friedland, Friedhelm Schwenker

University of Ulm - Institute of Neural Information Processing
James-Franck-Ring, 89069 Ulm - Germany

Abstract. In this paper the application of reinforcement learning to Tetris is investigated, particulary the idea of temporal difference learning is applied to estimate the state value function V. For two predefined reward functions Tetris agents have been trained by using a ϵ-greedy policy. In the numerical experiments it can be observed that the trained agents can outperform fixed policy agents significantly, e.g. by factor 5 for a complex reward function.

1 Machine learning for game playing

Playing games, such like Chess, Go, Checkers, Backgammon or Poker, is always a great intellectual challenge to humans, and therefore game playing is a scenario to test and evaluate artificial intelligence methods, in particular machine learning aspects have been taken more and more into account during the last years. Many methods derived from the fields of traditional artificial intelligence and mathematical game theory have been utilized in computer games, for instance, game trees are one of the most popular tools. A game tree represents all the possible states of a game in its nodes. Starting point is the root node representing the starting configuration of the game. Children nodes within the tree are representing all states that can be reached for the current node following the rules of the game, and leaf nodes are representing the terminal states of the game. In almost all (interesting) games the complete tree is too large to search all the possible pathes, and therefore the space to be searched must be reduced by applying heuristics which were tailored by human experts. A popular attempt in this direction is to estimate the winning chance of the player by so-called evaluation functions, and learning such evaluation functions utilizing machine learning techniques became an important challenge of research in modern artificial intelligence.

Artificial neural networks have been successfully used in many scientific and real world applications, for instance in pattern recognition, data mining, time series prediction. In recent years some attempts have been made to train artificial neural networks for game playing tasks. Tesauro [1] has applied feedforward neural network models to play Backgammon where the artificial neural net was used together with reinforcement learning (RL) algorithms. Here in this paper a RL algorithm which is known as temporal difference learning has been investigated for playing Tetris. The Tetris board is a grid of 10 columns and 20 rows of cells, which totals to 200 cells. Every cell can be in two possible states called

empty and *filled*. A Tetris gaming piece consists of 4 cells, therefore they are also known as *tetrominos*.

A Tetris gaming piece is drawn randomly at the beginning of the game and placed in the top area of the board. Possible players turns are *rotations* of the piece and *dropping* the piece. In fixed time intervals the piece is moved automatically *down* by a row. If it hits prior pieces or the bottom line of the board, either after some time or by a dropping turn of the player, it stays there and cannot be moved further. Then the next piece is randomly drawn. As the player places the pieces on the board, rows will get filled more and more. If a row is completly filled it is removed from the board and all rows above are moved one row down. The goal of the game is to avoid hitting the top line of the board for as long as possible. A detailed description of the game can be found in [3]. Our implementation, particularly the GUI, is based on [4].

2 Reinforcement Learning

Tetris has been proven to be NP-complete [2]. The consequence of this result is that it is impossible to find an optimal policy in the policy space effectively, thus RL methods could be of interest to find approximating solutions. The basic RL scenario contains two interacting parties: an agent and its environment. Assuming the enviroment at time t is in a current state (s_t). In this particular state the agent can select an action a_t from a set of possible actions $\mathcal{A}(s_t)$. After the agent has performed action a_t the environment sends a reward $r_t(a_t, s_t)$ to the agent and executes a state transition $s_t \mapsto s_{t+1}$.

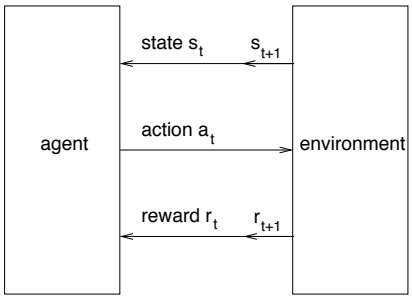

Fig. 1: RL scenario showing the interaction between agent and environment.

The agent's goal is to maximize the sum of rewards over time. The agent estimates the values of states it is able to achieve or the values of actions in states he encounters. These are called state value functions (in the following denoted by V) and action value functions respectively. Using this information allows the agent to choose better actions in respect of solving the task. A comprehensive guide on reinforcement learing is given in [5].

The *greedy* action a_t^* is determined by taking the one with a maximum sum

of reward and value of the following state.

$$a_t^* := \underset{a_t \in \mathcal{A}(s_t)}{\operatorname{argmax}} \; r_t + \gamma V(s_{t+1})$$

here $\gamma \in (0, 1)$ is a discounting factor.

Obviously, only these greedy actions are used for testing. In training, it's often useful to explore other states and actions. To allow other actions and states to be reached, a random action is taken with a rate of ϵ.

By modelling the reinforcement learning scenario as an Markov decision process through $\mathcal{P}_{ss'}^a$, namely the propability of changing from state s to s' under action a, and $\mathcal{R}_{ss'}^a$, the respective reward, one could formulate the relationship between values of an optimal V-function:

$$V^*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \left(\mathcal{R}_{ss'}^a + \gamma V^*(s')\right)$$

These are called *Bellman equations*, see [6] for a mathematical analysis on dynamic programming.

There are many approaches for estimating such optimal solutions. In this work, we will use the *temporal difference learning rule*

$$V(s_t) := V(s_t) + \alpha \left[r_t + \gamma V(s_{t+1}) - V(s_t)\right]$$

here $\alpha > 0$ is a small learning rate.

3 Reinforcement Learning for Tetris

A tabular representation of the V-Function is too large to be stored in any available memory. Just take into account every one of the 200 cells is allowed to be in 2 different states. So the upper bound for the whole Tetris board is 2^{200} possibilities. This can be reduced by a few percents through some observations. For example, a row can never be totaly filled or empty. So there are $2^{10} - 2 = 1022$ possible states per row, this mades up a number of 1022^{20} states for the whole board, which is roughly 96% of 2^{200}, and therefore the problem of storing such immense amounts of data cannot be solved this way.

In a first attempt, we limited our view to the two topmost used rows which are not empty. Current architectures easily deal with the resulting $2^{20} \approx 10^6$ state values. However, these experiments did not lead to any utilizable results. We traced this back to the inability of such a representation to map the state transitions triggered by placing pieces. As very different states were assigned to same table positions, their conflicting estimates did not allow to archieve any generalization.

In the next attempt the agent's view was limited to the top four used rows. If every cell is represented by two states, four rows of ten cells would have $2^{40} \approx 10^{12}$ possible state values, which is too large. We tried an other encoding

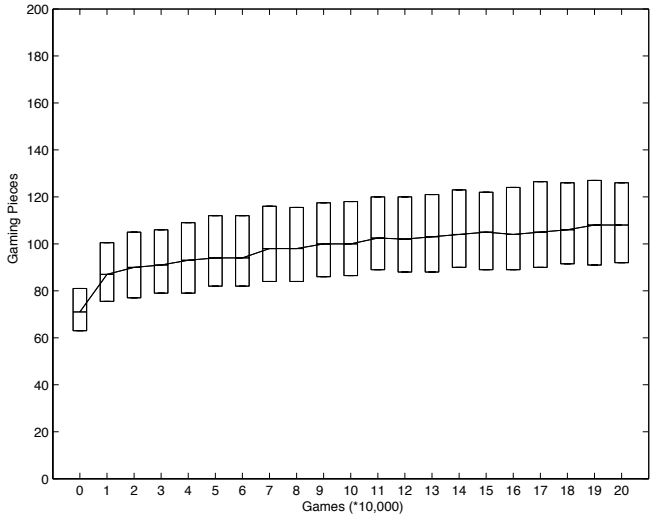

Fig. 2: Median, upper and lower quartile values for a simple reward function. The agent was trained over 200,000 games. First column, labelled by 0, represents the untrained agent using the fixed policy.

of the state to reduce this number to a feasible size. The agent's view was divided into ten columns each one consisting of four cells. The colums' states were coded separately by noting the top used cell height. This resulted in five possible states per column. As there are ten columns the agent's view can be represented by $5^{10} \approx 10^7$ different states. This state representation led to utilizable results. It was used in this work. The drawback of this approach is the inability of the agent's view to represent free cells under used ones. We tried larger views with heights of five or even six rows and an analogical state representation. They did not bring further substantial improvement, because many of the state transitions could be represented in the four row view. Instead, the results of our tests were disappointing. As a reason for this behavior we found that the increased number of state values required a corresponding number of episodes to be played until usable state value estimates were collected. So we did not pursue this way.

Tetris cannot be won, as shown in [7]. Therefore it is less promising to give some rewards at only at the end of the game. To avoid such weak rewards to the agent, a heuristic evaluation function for all the possible states are defined to get some more valuable rewards at any time step t. These functions have been constructed by a linear combination of weighted features.

Figure 2 shows the performance for a simple reward function. It is determined by the highest used (not empty) row of the board before and after taking an action. This difference (multiplied by a scaling factor 10) has been used as evaluation fuction for an action and sent to the agent as reward $r(a, s)$ Using this fixed function to select greedy actions and testing 1000 games, leads to the

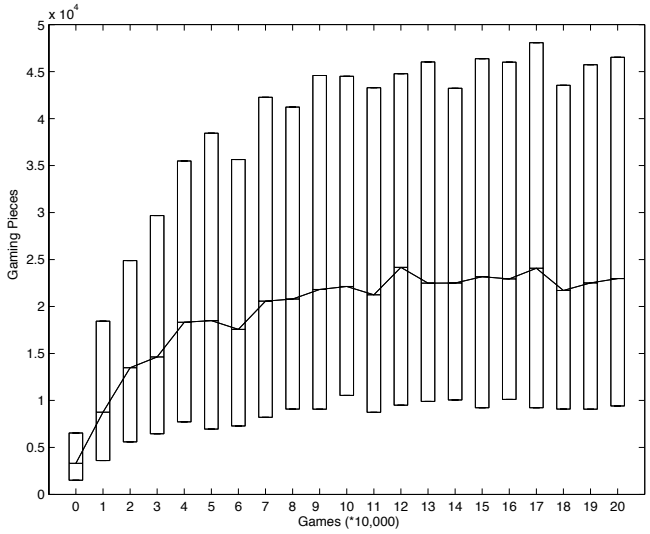

Fig. 3: Median, upper and lower quartile values for a complex reward function. The agent was trained over 200,000 games. First column, labelled by 0, represents the performance of the untrained fixed agent.

results plotted in the first boxplot (labelled by 0 to index the untrained agent), Then, in the following the agent has been trained through 20 episodes each of 10,000 games. After each episode the agent has been tested again by 1000 games. Numbers of played gaming pieces per test game are collected and plotted in a simple boxplot, showing median, upper and lower quartile. We ran experiments with different values for α und ϵ. For smaller values the achieved results were similar to the ones shown here. The drawback was their time consumption, which could be reduced by choosing larger values. As we let these values grow, the learning became more and more unstable until the point where no positive effects were seen at all. This was at values roughly ten times the ones mentioned at the end of the paragraph. So we came up with values which have a good learning effect. The parameters include a constant learning rate $\alpha = \frac{1}{100}$, and a slowly falling random action rate $\epsilon = \frac{1}{1+15\ln[n]}$, where n is the number of the games used to train the Tetris agent. It can be observed that the fully trained agent (after 20 episodes) outperforms the untrained agent slightly.

Figure 3 demonstrates an experiment using the same training parameters, but utilizing a more informative reward function. At designing this function we made several considerations how to improve the simple reward function. First of all, looking at only the topmost used row renders state transitions below it invisible to the reward function. Therefore we exchanged this feature against the average of the heights of all ten columns. Our second thought concerned the inclusion of holes between pieces as they were placed. This should be rated as negative in regard to the aim of removing rows from the game board. Further ideas we

had included a measure for the shape of the pile that was formed. A proposal comprised simply summing up the absolute differences of neighboring columns. That showed some promising results but did not satisfy our expectations. So we refined this by squaring the differences before summing them up as we thought larger steps should be counted more severe as small ones. We called the resulting shape measure *quadratic unevenness*. Some more features were formulated and experiments were ran with a range of weighted sums of these to find a suitable reward function. Finally, we came up with the function used for figure 3. It constists of a weighted sum of three features. The first feature is the average of the heights over all 10 columns. This value is weighted by factor 5. The second feature is the number of holes appearing between the pieces, this number is then multiplied by 16. The last feature is the discussed criterion to measure the roughness of the shape, the quadratic unevenness. This feature is weighted by factor one. These three terms are summed up to the reward function. For this reward fuction the performance is shonw in Figure 3, again the agent has been trained 20 episodes each with 10,000 games. Testing results - based on again 1000 testing games - are shown for the untrained agent (column 0) and after each of the 20 episodes in Figure 3.

4 Conclusions

In this study reinforcement learning has been used to improve hand made fixed policy strategies. In both cases the performance of the fixed policy was improved significantly. Using this complex reward function as a fixed strategy to evaluate states during the game, an average duration of 5,000 pieces per game corresponding to 2,000 deleted rows per game was achieved. Adapting the state value function V by a simple temporal difference learning procedure together with the exploration of the state space using an ϵ-greedy policy the agent's performance increased on average to more than 20,000 pieces, see Figure 3. Similar results, but with smaller performance gain, were achieved for the simple reward function. Ongoing work includes more elaborated reinforcement learning techniques and the application of meta learning heuristics.

References

[1] Gerald Tesauro. *Temporal Difference Learning and TD-Gammon*, Commun. ACM, 38(3), p. 58.68, 1995.

[2] Erik D. Demaine, Susan Hohenberger, David Liben-Nowell. *Tetris is Hard, Even to Approximate*, Technical Report MIT-LCS-TR-865, MIT Computer Science Laboratory, 2002.

[3] Colin Fahey. *"'Standard Tetris"' specification*, `http://colinfahey.com/tetris/tetris.html`, 01.12.07

[4] Nick Parlante. *Stanford Tetris Project*, `http://cslibrary.stanford.edu/112/`, 01.12.07

[5] Richard S. Sutton, Andrew G. Barto. *Reinforcement Learning: An Introduction*, The MIT Press, Cambridge, 2002.

[6] Richard E. Bellman. *Dynamic Programming*, Dover Publications, New York, 2003.

[7] Heidi Burgiel. *How to Lose at Tetris*, Mathematical Gazette, July 1997, pp. 194-200.