A Reinforcement Learning Algorithm to Train a Tetris Playing Agent

Patrick Thiam, Viktor Kessler, and Friedhelm Schwenker

Ulm University, Institute of Neural Information Processing
89069 Ulm, Germany
friedhelm.schwenker@uni-ulm.de

Abstract. In this paper we investigate reinforcement learning approaches for the popular computer game *Tetris*. User-defined reward functions have been applied to $TD(0)$ learning based on ε-greedy strategies in the standard Tetris scenario. The numerical experiments show that reinforcement learning can significantly outperform agents utilizing fixed policies.

1 Introduction

Tetris is a popular computer game originally invented by the Russian mathematician Alexey Pajitnov in the mid 1980's, and nowadays it is implemented on almost all operating systems and hardware platforms. The standard Tetris board has a size of 200 cells (arranged in 10 columns and 20 rows), where each cell has two states: free or occupied, and thus 2^{200} gaming board configurations are possible. During the game, gaming pieces (shapes of four connected cells, also called *tetrominos*) are dropped from the top of the gaming board into the board and stacked upon occupied cells or the bottom line of the gaming board. In the standard Tetris seven different tetrominos exist, and pieces to be dropped are selected with equal probability. The player can select one out of the ten columns and can rotate the current tetromino before dropping it. When a line of cells is occupied the line is removed and all cells above it are moved down by one line. Each removed line adds to the player's score, and multiple lines can be removed at the same time. The game is over when a cell in the top row is occupied by the current tetromino. The goal is to maximize the score. Because of its complex nature Tetris has been proven to be NP-complete [3]. The consequence of this result is that it is not possible to find an optimal policy effectively, and thus artificial intelligence methods could be of interest to find approximating solutions. Because of its popularity, standard Tetris [8] as well as variations, such as SZ-Tetris [7] have become popular benchmark tests for various machine learning algorithms during the last years.

Neural networks have been successfully applied to numerous real world applications, for instance in pattern recognition, data mining, time series prediction. In recent years several attempts have been made to train artificial neural networks for game playing tasks. For instance, Tesauro [10] has successfully applied feedforward neural networks to play Backgammon. In this scenario artificial neural networks are applied in conjunction with reinforcement learning (RL) algorithms. Combinations of reinforcement learning with artificial neural networks

N. El Gayar et al. (Eds.): ANNPR 2014, LNAI 8774, pp. 165–170, 2014.

and ensemble learning have been successfully applied to board games such as *Connect Four* or *English Draughts* [6,4,5]. Here in this paper we apply temporal difference learning - a well know RL algorithm - to train a Tetris playing agent.

The major goal of this work is to explore and evaluate the effectiveness of reinforcement learning techniques to train a Tetris playing agent. The paper is organized as follows: In Section 2 a brief introduction to RL is presented, then in Section 3 the $TD(0)$ implementation for the standard Tetris application is described. The numerical experiments are shown in Section 4, and finally we discuss results and draw conclusions in Section 5.

2 Introduction to Reinforcement Learning

A reinforcement learning scenario contains two interacting parties: an agent and its environment. We assume that the environment can be completely observed, so for any time step t the environment is in a particular state s_t. Given this sate, the agent can select an action a_t out of a set of possible actions $\mathcal{A}(s_t)$. After the agent has performed an action a_t the environment gives a particular reward $r_t(a_t, s_t)$ to the agent and performs a state transition $s_t \mapsto s_{t+1}$.

The agent's goal is to maximize the sum of rewards over time, for this, a state value function (in the following denoted by V) has to be estimated. Using this information allows the agent to choose appropriate actions with respect to the given task. A comprehensive guide on reinforcement learning can be found in [9].

The *greedy* action a_t^* is determined by taking the one with a maximum sum of reward and value of the following state.

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

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

The *policy* defines the strategy used by the agent to choose its next action. Obviously, only these greedy actions are used for testing. In training, it is 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. In this work an ε-greedy policy will be use, with $\varepsilon \in [0, 1)$. The agent plays a *greedy* action with a rate of $1 - \varepsilon$ (*exploitation*) and a *random* action with a rate of ε (*exploration*).

In case a *greedy* action is chosen, the value of the current state $V(s_t)$ has to be adjusted according to the *temporal difference learning rule* (see Eq. 2). The *greedy* action a_t^* is determined by taking the one with a maximum sum of reward and the weighted value of the following state $r_{t+1} + \gamma V(s_{t+1})$. The reward is a function that assesses the configuration of a state at each given time t giving it a numerical valuation r_t. This function is used to evaluate the next state s_{t+1} and the value obtained r_{t+1} is used in combination with the weighted value of the next state $\gamma V(s_{t+1})$ as a comparison parameter to select the *greedy* action.

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 $\mathcal{R}^a_{ss'}$, the respective reward, one could formulate the relationship between values of an optimal V-function:

$$V^*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma V^*(s') \right) \tag{1}$$

These conditions are called *Bellman equations*, please see [1] for a detailed mathematical analysis.

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

$$V(s_t) := V(s_t) + \alpha \left[r_t + \gamma V(s_{t+1}) - V(s_t) \right] \tag{2}$$

where $\alpha > 0$ is a small positive learning rate.

3 $TD(0)$-Learning for Tetris

It has been shown in [2] that Tetris cannot be won. 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.

The reward functions used in this work have been designed through linear combinations of weighted features. The first two features consist of the value of the highest used column (max^t_{height}) and the average of the heights of all used columns (avg^t_{height}) at each given time t. The next feature consists of the total number of holes between pieces at each given time t (cnt^t_{holes}). The last feature consists of the quadratic unevenness of the profile (U^t_{Pro}). This feature results from summing the squared values of the differences of neighboring columns.

$$r_{t+1} = 5 \times (avg^{t+1}_{height} - avg^t_{height}) + 16 \times (cnt^{t+1}_{holes} - cnt^t_{holes}) \tag{3}$$

$$r_{t+1} = 5 \times (avg^t_{height} - avg^{t+1}_{height}) + 16 \times (cnt^t_{holes} - cnt^{t+1}_{holes}) + (U^t_{Pro} - U^{t+1}_{Pro}) \tag{4}$$

Both reward functions take both next state s_{t+1} and current state s_t into consideration. They describe how good is the transition from the current state to the next state, whereby the higher the returned value the better the state. Furthermore the second reward function (cf. Eq. 4) uses the quadratic unevenness as an additional feature. Later we will see the impact of this particular feature in the performance of the agent.

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 gives 2^{200} configurations. In order to tackle this problem and reduce the state space to a usable size, the height difference between adjacent columns was used to encode each state. For a given state the height difference between successive columns is computed. Prior to that, a threshold is set to limit the maximum and minimum height difference. In this work the

threshold was set to ± 3. The possible height differences form a set of 7 values: $\{+3, +2, +1, 0, -1, -2, -3\}$. All height differences outside this range are truncated to ± 3. Subsequently, each state is represented as a 9-tuple of values taken from the previous set. Using this method results in reducing the state space to $7^9 \approx 40 \times 10^6$ possible states.

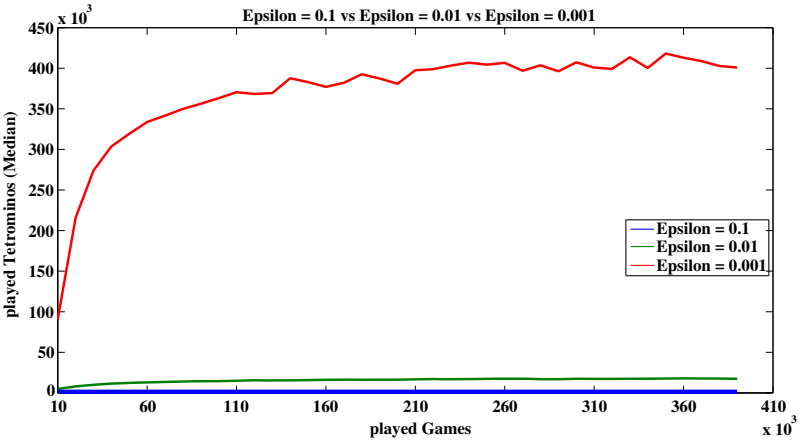

Fig. 1. Learning curves of the RL agent trained through $TD(0)$ with ε-greedy policy and reward function as defined in Eq. (4) $\varepsilon = 0.1$ (green) vs $\varepsilon = 0.01$ (blue) vs $\varepsilon = 0.001$ (red)

4 Numerical Evaluation

Several experiments were undertaken to assess the performance of the implemented agents. They consist primarily of alternating learning and test phases. Prior to that, values for the learning rate $\alpha = 0.1$ and the discount factor $\gamma = 0.9$ were set for the entire experiments. The total number of played gaming pieces per game is used as the performance indicator. At first, the agent is trained through a fixed number of games which depends on the experiment being undertaken. Subsequently, a test phase follows in which the agent is tested by 1000 games. These episodes of alternating learning and test phases are repeated several times in order to achieve a robust estimate of the agent's performance.

Figure 1 shows the performance of the reward function defined in Eq. (4). The agent is trained through 37 episodes of 10000 games each. The total number of played gaming pieces is collected for each played game. The median of these values is plotted at the end of each learning phase. This experiment is repeated for three different ε-values (0.1, 0.01, 0.001).

In order to perform a fair comparison between different ε-values a second experiment was conducted. Here the agent's performances are compared on the basis of *greedy* played gaming pieces instead of the number of played games. A test phase of 1000 games follows each training phase, and again the median value

of the total number of played gaming pieces has been taken as performance measure. Figure 2 shows the performance of the second (cf. Eq. 4) reward function. For this experiment a threshold of 10^8 *greedy* played gaming pieces is set. During each training phase, the agent is trained with so many games until this threshold is reached. The experiment is repeated for two different values of ε (0.1, 0.01). The abscissas depict the total number of *greedy* played gaming pieces during the training phase, and the ordinates depict the median value of played gaming pieces during the test phase, whereby the value labeled zero shows the result of the untrained agent. Untrained agents take actions according to the evaluation function given in Eq. (4).

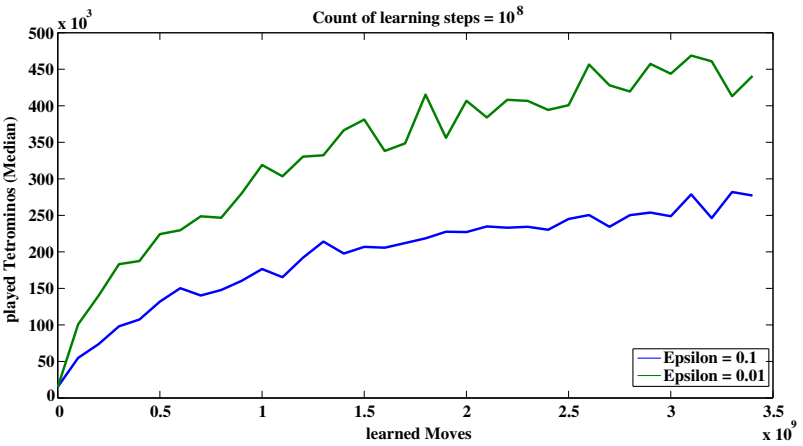

Fig. 2. Test curves of the RL agent trained through $TD(0)$ with ε-greedy policy ($\varepsilon = 0.1$ (blue) vs $\varepsilon = 0.01$ (green))

5 Discussion and Conclusion

After taking a closer look at the results plotted from the first experiment, it is clear that exploration increases with increasing ε values. Thus the number of played gaming pieces decreases. As the number of games per learning phase is constant, the number of *greedy* played game pieces decreases with increasing Epsilon. It follows that the agent needs to be trained with more games to achieve results comparable with those obtained with lower ε values. However, it is not possible through this experiment to determine the effect of ε on the learning performance of the agent. The next experiment serves this purpose.

Through the second experiment a comparison is done between the number of *greedy* played gaming pieces during the training phase and the number of played gaming pieces during the test phase. This comparison depicts at which extend the agent is able to learn when he follows solely its *greedy* policy. Thereby it is also possible to determine the effect of the *greedy* parameter ε on the learning performance of the agent. The results of the experiment for the second reward

function (cf. Eq. 4), shows clearly that the agent yields better performances for lower values of ε. Both plotted curves seem to reach a saturation point after playing approximately 2.5×10^9 *greedy* gaming pieces. Consequently, Figure 3 seems to show that larger exploration doesn't seem to help the agent play better. At this point it has to be pointed out that the agent had not been trained further due to the time consumption of both training and test phases. Thus this observation could not be proven throughout further investigation.

For the first reward function given in Eq. 3 learning results are very poor (only approximately 2000 pieces after training with 3.5×10^9 *greedy* gaming pieces) at least in comparison with the reward function given in Eq. 4 (approximately $400,000$ pieces after training with 3.5×10^9 *greedy* gaming pieces). The untrained agent using the second reward function reached a median value of 14000 played gaming pieces, while this value lied by 60 played gaming pieces using the first reward function. But in both cases, the overall performance has been significantly improved through $TD(0)$ learning utilizing ε-greedy policy. Ongoing research focuses on time-depending exploration in $TD(0)$ learning and applications to real world scenarios.

References

1. Bellman, R.E.: Dynamic Programming. Dover Publications (2003)
2. Burgiel, H.: How to lose at Tetris. Mathematical Gazette 81, 194–200 (1997)
3. Demaine, E.D., Hohenberger, S., Liben-Nowell, D.: Tetris is hard, even to approximate. In: Warnow, T.J., Zhu, B. (eds.) COCOON 2003. LNCS, vol. 2697, Springer, Heidelberg (2003)
4. Faußer, S., Schwenker, F.: Neural approximation of Monte Carlo policy evaluation deployed in Connect Four. In: Prevost, L., Marinai, S., Schwenker, F. (eds.) ANNPR 2008. LNCS (LNAI), vol. 5064, pp. 90–100. Springer, Heidelberg (2008)
5. Faußer, S., Schwenker, F.: Learning a strategy with neural approximated temporal-difference methods in English Draughts. In: 2010 20th International Conference on Pattern Recognition (ICPR), pp. 2925–2928. IEEE (2010)
6. Faußer, S., Schwenker, F.: Ensemble methods for reinforcement learning with function approximation. In: Sansone, C., Kittler, J., Roli, F. (eds.) MCS 2011. LNCS, vol. 6713, pp. 56–65. Springer, Heidelberg (2011)
7. Faußer, S., Schwenker, F.: Neural network ensembles in reinforcement learning. Neural Processing Letters, 1–15 (2013)
8. Groß, A., Friedland, J., Schwenker, F.: Learning to play Tetris applying reinforcement learning methods. In: ESANN, pp. 131–136 (2008)
9. Sutton, R.S., Barto, A.G.: Introduction to Reinforcement Learning. MIT Press, Cambridge (1998)
10. Tesauro, G.: Temporal difference learning and TD-Gammon. Commun. ACM 38(3), 58–68 (1995)