COMP9417 - Machine Learning & Data Mining
Session 1, 2013

Mareike Picklum, z3446393
Johan Klokkhammer Helsing, z3432903

Project Report Rest-It

Due: Sun, 09 Jun 2013 23:59

1 Introduction

The project *Rest-It* is an implementation of a reinforcement learner for the popular puzzle game Tetris. The core idea of the game is to place Tetronimoes (pieces consisting of four connected cells) in the way that the complete as many lines as possible in a 10 columns by 20 rows playing field. Each completed line is removed from the field and the cells above are moved one cell down. Removing a line adds one to the players score. Tetris is a common benchmark problem for testing machine learning algorithms, as it uses comparably simple rules although it requires complex strategies to perform well.

In Related work we will give a short overview about existing reinforcement learning approaches for Tetris and give information about what our project is based on and influenced by. We will then describe the algorithms we implemented in Method and present our results in Results. We will give a short summary and wrap up the results in Conclusions.

2 Related work

There are different approaches for tetris learners. Szepesvári (2010) describes different reinforcement learning algorithms and their core ideas and discusses their theoretical limitations.
Szita & Lörincz (2006) apply noise to improve the cross-entropy method by preventing it from early convergence and achieve a policy that outperforms previous algorithms by almost two orders of magnitude.
Carr (2005) introduces reinforcement learning for Tetris and introduces the Formal Tetris Specification by Fahey (2003).
Groß et al. (2008) investigate the idea of temporal difference learning applied to the Tetris problem and train agent using an ε-greedy policy.

Our code is based on an open source Qt Project[1] implementation of tetris.
The core idea for the reinforcement learning algorithm we use is described by Zucker & Maas (2009), which will be investigated in more detail in Method.

3 Method

Following the remarks in Zucker & Maas (2009), we first defined the properties of the game needed to calculate the quality. According to the common Tetris rules, a player can rotate, move and drop a Tetronimo. As it is an easy representation, we describe an action u as a position and rotation of a piece: $u = (p, r)$. The state x of the game consists of the current board occupancy, the current Tetronimo to be played, and the next Tetronimo to be spawned (in time step $t + 1$). To end up in a successor state x', an action u has to be applied being in state x. Whenever one or more lines have been removed applying that action u, the player receives the reward r, which is dependent on the current state and the action to be applied: $r(x, u)$.

[1] Qt Project

After investigating different variations of features describing the properties of a current state, we revisit the proposed feature combination mentioned in Zucker & Maas (2009), which have been proven to work best with our implementation. The feature vector thus contains the following information:

0-9: the height of each cell

10-18: absolute height difference between adjacent cells

19: overall maximum height

20: number of holes in the board

21: number of removed lines (≙ reward)

While the number of removed lines denotes the obvious quality of an action, or the action-transition from one state to a successor state, the other features constitute important quality attributes too. The height of each cell as well as the absolute height difference between adjacent cells characterize the "smoothness" or "evenness" of the columns' surface. A state in which the columns are very low and even is to be preferred from a bumpy surface with single columns almost reaching the top of the board. An increasing number of holes in the board implies increasing complexity of the game, as it makes it harder to complete several lines at a time, and therefore reducing the overall height.

Adding further features such as the maximum height difference or the number of holes per column do not add relevant information and therefore do not lead to significantly better results.

The quality of a state is calculated from a parameter (= weight) vector θ within a quality function $Q(\theta, x)$. The weights in θ denote the "importance" or "good-/badness" of the corresponding values in the feature vector. This function is used calculate the quality of each successor state x' of the current state x. The action u leading to the state with the highest quality will then be the preferred action to be applied. However, this action is only chosen by a certain probability. This is used to prevent getting stuck in a local maxima, which can happen, when the learner chooses an action that leads to a good result in that particular case, but is not necessarily a good action in different situations. If for example, a line has been completed dropping a Tetronimo in the first column, the learner receives a reward for this action and learns that it is a good idea to drop pieces in the first column. Of course, this is not a clever strategy in general. It is important to set the learning rate to a value that is low enough to prevent a single positive action from having too much influence on the overall learned rules. Using a momentum also ensures more actions than just the last one have an impact on the changes in θ.

The weights are updated by adding a step value δ, which is determined by the addition of the previous step value and the quality of the next action. This quality, again, is dependent on the immediate reward (number of removed lines after applying that action) and the quality of the successive state.

More precisely:

$\Delta_{t+1} = \Delta_t + \frac{t}{t+1}(r(x_{t+1}, u_{t+1})z_{t+1} - \Delta_t)$

where z can be understood as a momentum which is defined by

$z_{t+1} = \beta z_t + \frac{\bigtriangledown q(\theta, x_{t+1}, u_{t+1})}{q(\theta, x_{t+1}, u_{t+1})}$.

Here, β is a "momentum-influence-factor", $\frac{\bigtriangledown q(\theta, x_{t+1}, u_{t+1})}{q(\theta, x_{t+1}, u_{t+1})}$ the score ratio.

4 Practical Issues

As we use softmax, some of the values can become very small and occasionally be rounded to zero. To prevent this, we switch temporarily to hardmax whenever softmax would cause a floating point error.

5 Results

To be able to compare the results of our implementation and see if different changes have significantly different outcomes, we extended the base implementation with some GUI widgets that provide useful information such as the number of played games, the maximum number of removed lines and the average number of removed lines. As the first pieces are randomly placed, it happens by chance, that a line is removed. This is the moment, in which the learner receives a reward for the first time. Often the learner has to play 50 or more games before it finally removes a line. In that case, the average number of removed lines is not very meaningful. Also, after running the learner for a while, this value does not change significantly, and does not show significant short-term changes. That is why we introduced another value, denoting the average number of removed lines *over the last n games*, the "moving average".

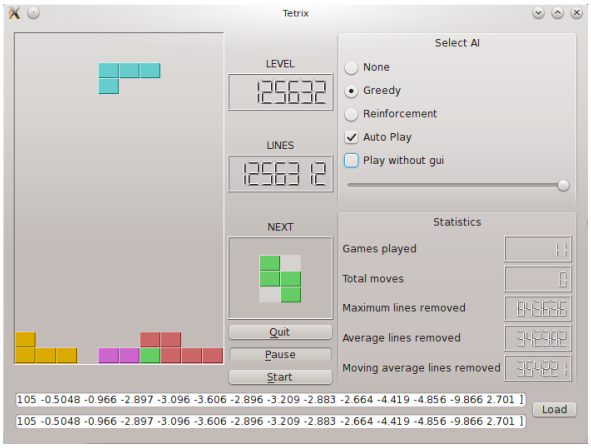

Figure 1: The current number of removed lines already exceeds one million. Note, that the difference between the total and the moving average is not significant, as the game has been initiated with excellent weights, and the learning period was left out.

We also implemented an agent, which uses hardcoded rules to play Tetris. These rules are what we consider to be the "naive" way to play the game, comparable to an unexperienced player. The results of this rather unintelligent approach are used as a baseline. The algorithm with a learning function should by all means outperform the naive Agent.

We also use a greedy implementation, which simply uses given weights for the features to play the game, without learning from it. This is used to compare the quality of the learned weights (from the reinforcement learner implementation). These weights can simply be fed in through a text field in the GUI.

Figure 1 shows the greedy algorithm using obviously excellent values previously learned by the reinforcement learner. The number of removed lines already exceeds 1,250,000 after 11 played games.

The performance of the reinforcement learner usually does not outperform the greedy algorithm. To prevent getting stuck in a local maximum, the learner only chooses the "best" action (based on what it has learned before) to a certain probability. Otherwise, it just performs a random action, which can be interpreted as "exploring" new options without being too dependent on the possibly disadvantageous rules it has learned.

The difference (exemplified with a rather old implementation of the reinforcement learner) can be seen in Figure 2.

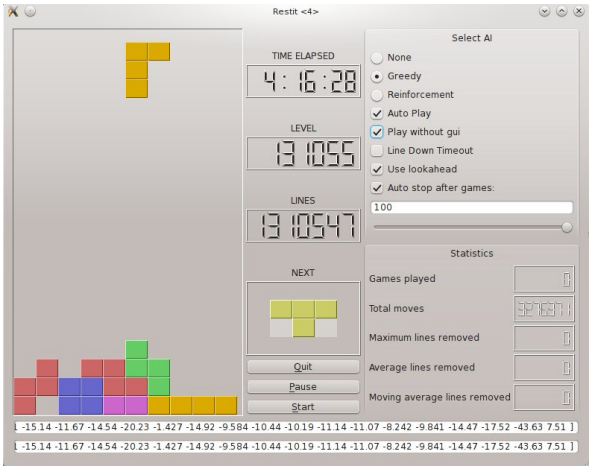

Figure 2: More recent values: The greedy algorithm runs since more than four hours and has not lost one single game yet. The number of removed lines exceeds 1.3 million

Surprisingly, the weights learned by the reinforcement learner are actually worse when using lookahead compared to only using the current piece to determine the best action. While the greedy agent, which just uses a given weight vector to play, improves using the lookahead, the reinforcement learner keeps on learning disadvan-

Run	Lines removed in first game	Maximum Lines Removed	Games played
1	336,989	630,468	6
2	659,616	659,616	1
3	703,426	703,426	2
4	1,700,103*	1,700,103*	1
5	1,743,241*	1,743,241*	1

* interrupted before game ended

Table 1: 5 different instances of the reinforcement learner trained with 5000 games each. Evaluated with learning turned off, using lookahead.

tageous weights, which can, for example, result in stacking pieces on both edges of the board. Using a momentum adds to this effect. We do not really have an explanation for this, so this might be a subject for further research.

What turned out to be the best strategy, was to let the agent run without lookahead, and then for the evaluation switch to a greedy method with lookahead using the learned weights. We assume this is because the learned weights for tetris without lookahead approximate a desire for "safe" states. Getting a high score only depends on staying alive for the longest possible amount of time, so essentially good weights when not using lookahead should approximate good weights when using lookahead. It is much simpler for the learner to figure out what those weights are if lookahead is not enabled.

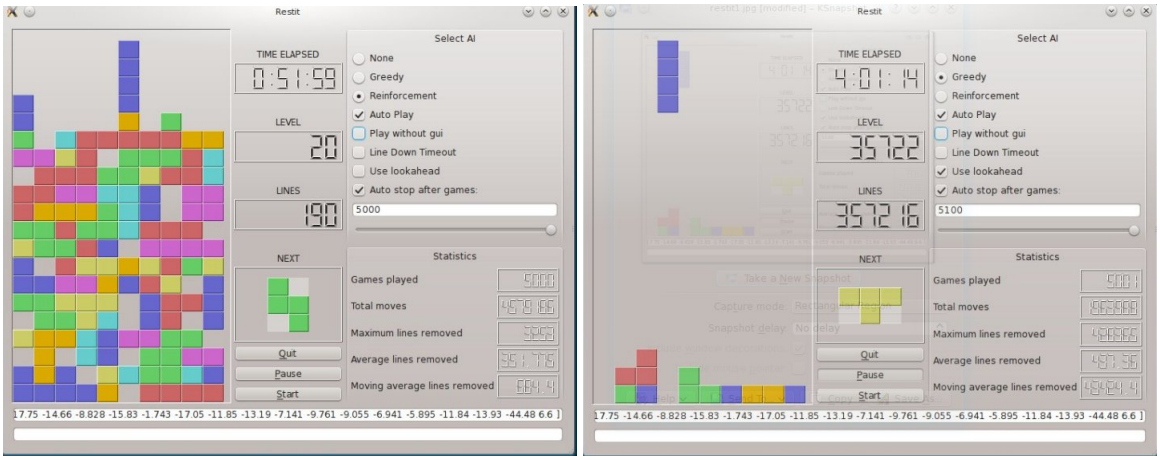

Figure 3: The learner has played 5000 games in less than one hour, the greedy algorithm takes more than 3 hours for two games

We found empirically that a learning rate, α, of 0.1 was suitable. A bigger learning rate seemed to cause the learned values to fluctuate. On the other hand a lower learning rate caused the learning process to last unreasonably long. As Zucker & Maas (2009), we found that a momentum, β, of 0.5 was suitable, as long as we don't use lookahead.

For the benchmark, we first let the learner run for 5000 games without using lookahead. Then we switched off learning (greedy algorithm) and turned on lookahead for the evaluation and planned on running 100 games. This is the same setup as Zucker & Maas (2009) used. The problem however, was that the learner performed way better than we had expected. Just a single game usually lasts over two hours, and exceeds 600,000 removed lines. This made it very impractical to do 100 tests.

We ended up running 5 parallel learners, and tested them on just one game each. The results are shown in Table 1.

6 Conclusions

With a few minor changes, we basically implemented the algorithm described by Zucker & Maas (2009), but we surprisingly get much better results. We assume that training the weights without lookahead and only using it in the evaluation phase, makes a big difference. In general, we achieve better results than we expected and except for a few minor implementation bugs we consider this project as successful. A video of the learning procedure can be found on YouTube[2].

References

Carr, D. (2005), Applying reinforcement learning to tetris, Technical Report MSU-CSE-00-2, Department of Computer Science, Rhodes University, Grahamstown 6139, South Africa.

Fahey, C. P. (2003), 'Tetris specifications & world records, 2003', *URL http://www. colinfahey. com/2003jan tetris/2003jan tetris. htm* .

Groß, E., Friedl, J. & Schwenker, F. (2008), 'Learning to play tetris applying reinforcement learning methods'.

Szepesvári, C. (2010), *Algorithms for Reinforcement Learning. Synthesis Lectures on Artificial Intelligence and Machine Learning*, Morgan & Claypool Publishers.

Szita, I. & Lörincz, A. (2006), 'Learning tetris using the noisy cross-entropy method', *Neural computation* **18**(12), 2936–2941.

Zucker, M. & Maas, A. (2009), Learning Tetris. School of Computer Science, Carnegie Mellon University, USA.

[2] Reinforcement Learner for Tetris Reloaded