Learning the game of Tetris using Batched Temporal Difference Learning

Joris de Keijser, Herbert Kruitbosch, David Otterbein & Martien Scheepens

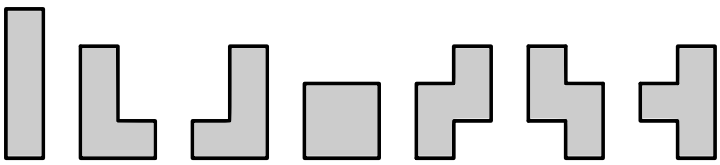
University of Groningen, January 2013

Abstract

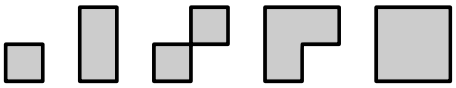
This research uses a batched version of temportal difference learning with V-values to train an agent for the game of Tetris. Moves are batched in a training set to prevent overfitting and learn more from such moves. The V-values are trained with a neural network for which different features from [1] are used. The agent is trained on the standard tetromino set, and a simpler one. During training, the agent is given negative rewards when it causes a game over. In conclusion, the agent performs better if higher punishments are given for rewards for both tetromino sets. All experiments were run in Java, the source is available on request. The results were plotted using Matlab, also these scripts are available on request.

1 Introduction

Tetris was invented by Alexey Pazhitnov in the mid-1980s and became a popular video game. Tetris has been implemented many times since then, sometimes with altered rules. This paper discusses an artificial agent to play two versions of the game, one with the standard set of tetrominoes (displayed in figure 1a), and one with a set of simpler tetrominoes (displayed in figure 1b). The simpler set of tetrominoes was suggested by [2]. Further rules of the Tetris game are as specified in [3], a summary of the rules can be found in [4].



(a) standard set, from left to right: I, $L1$, $L2$, B, $S1$, $S2$ and T.



(b) simpler set, from left to right: One, Two, $Skew$, $Three$ and $Four$

Fig. 1: Tetrominoes used in different versions of the Tetris game.

The success of the game might be due to the difficulty of finding an optimal location and orientation for a given tetromino. This makes Tetris an interesting game to be played by a computer. Ideally an algorithm for playing tetris looks like this:

Algorithm 1.1: PLAYTETRIS()

```
while ¬game over
  do play most optimal move
```

The offline version of the game, where the complete sequence of tetrominoes is known in advance, was proven to be NP-hard [5]. This proof was not about playing the game of Tetris,

but finding the minimum height of filled blocks on the playing field needed to play the given tetromino sequence.

Considering an online version of the game, where the next tetromino is only known if the previous was already played, is also complex. Mainly because it is hard to describe what an optimal move is, since given different sequences of tetrominoes that follow, different moves might be optimal. This paper discusses a method of playing online Tetris and defines a criterion for quantifying how *optimal* a certain *playing field* is. This criterion is based on heuristic measures and should take into account possibilities for future moves to clear lines and prevent a *game over*.

2 Application

Reinforcement learning is used to take decisions in Markov decision process, for example financial industries might use reinforcement learning to evaluate how well certain investment strategies work out or to decide how profitable buying certain shares are expected to be. In this paper it is used to play Tetris.

3 Method

This paper discusses a reinforcement learning (RL) approach to playing Tetris. Figure 2 shows the schematics of RL. The environment is the actual Tetris game itself, which keeps track of the state of the play field, provides tetrominoes and updates the state whenever the agent does a move. An important feature of the environment is that it can give the agent feedback about a move, using a reward. The agent can use this information to learn a policy that maximizes the cumulative reward of a sequence of actions. This paper considers a Tetris environment which rewards nothing (0), except if one or more lines are cleared or the move caused a game over. In the first case the reward is positive, in the latter it is negative. This will be further discussed later.

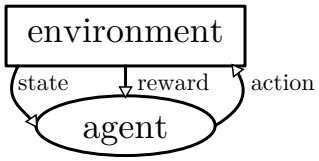


Fig. 2: Schema of reinforcement learning.

3.1 Action

When presented with a human player, typical Tetris environments will provide the vertical movement of a tetromino simulating the *falling* movement of the tetromino. The human player can alter the trajectory of the tetromino by moving it horizontally or rotating it.

Regardless of how to choose the *best* or *most optimal* action, the agent must know what possible moves there are. Simplifying the possible actions will simplify choosing an action for an agent. We assume a machine player will be able to both rotate a tetromino in the desired orientation fast enough and move it to the desired horizontal location. Therefore the environment allows the agent only to specify the orientation and the column where the leftmost block of the tetromino should be placed. This means that the environment does not simulate any movement (trajectory) in the field. Side effects of the simplification are that

tetromino	I	$L1$	$L2$	B	$S1$	$S2$	T	One	Two	$Skew$	$Three$	$Four$
# rotations	2	4	4	1	2	2	4	1	2	2	4	1
# actions	17	34	34	9	17	17	34	10	19	18	36	9

Tab. 1: Number of possible rotations and actions for a certain tetromino.

the agent cannot shove a stone into a cave with an opening on the side or perform special moves like the T-Spin[6]. Sometimes tetrominoes placed near or at the top of the play field can block the vertical movement of a tetromino, such that it can not be dropped at certain columns. These subtle differences are ignored and strictly our agent is not trained to play tetris as described in [3].

The play field has 10 columns, and thus the leftmost block of a tetromino can be placed at most in 10 columns, in some cases less, since the tetromino is not allowed to be placed outside of the play field. Moreover each tetromino has at most 4 orientations, as they are rotated in angles of $\frac{1}{2}\pi$. Table 1 states how many rotations and possible actions each tetromino allows for.

3.2 State

The state the environment keeps track of consists of three things, the 10×20 blocks which either contains (part of) a tetromino or nothing, the tetromino which should be played next in this field and whether the the game is over or not.

The set of all possible states is quite big, namely $2^{10\times 20} \times (7 \text{ or } 5)$. The $2^{10\times 20}$ part is for the filling of the play field, then 7 or 5 part is for the tetromino to be played. Then there should be twice as many states since the game can either be over or not. Fortunately a game over is not possible for each play field, only those where the topmost row is reached can be game over. This is a small part of the total state space, and is therefore neglected.

The filling of the play field is quite large, but also describes almost all information needed to decide which action is best. The next tetromino to play, is provided by the environment and selected randomly from a uniform distribution. This distribution is independent of the previous action and therefore the next tetromino is less relevant when deciding which action is best.

This paper takes several suggestions to *subtract* certain information from the state space of the play field into account. The first was suggested in [7], they only provided the contour of the top of the play field by specifying the difference in height between adjacent columns.

Another set of features are pile height (PH), number of holes (H), the sum of all wells (W), the number of row transitions (RT) and the number of column transitions (CT). These are defined in [1]. Another feature is the minimum over all columns of the topmost filled block in the columns, the pile bottom (PB). An example of a play field and these features is displayed in figure 3. Together with five other features, Pierre Dellacherie managed to clear on average around $650 \cdot 10^3$ lines in a game [1]. The algorithm to play tetris using these features boils down to:

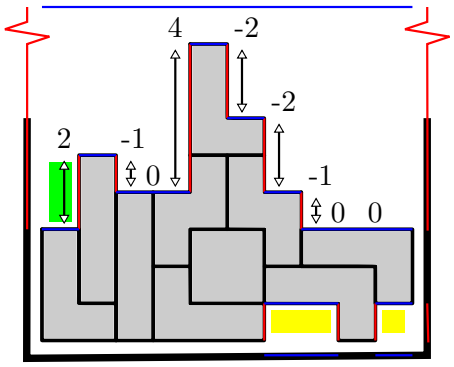


Fig. 3: A play field, the red lines indicate column transitions (there are 50), the blue ones row transitions (there are 26), the numbers are the height differentials, the green blocks are wells (there are 2) and the yellow blocks are holes (there are 3). Furthermore the pile height is 8 and the pile bottom is 3.

Algorithm 3.1: PLAYTETRIS()

while ¬game over
do
Observe the current tetromino t
for each possible action a for t
do try a and calculate a $V(s_a)$ from the features of the state s_a to which a has led
take the action for which V is maximal

The value V_a can be calculated as a linear combination of the features with some predetermined weights or a more complex formula combining these features. In the case of a linear combination, this calculation can be done by a neural network with linear activation nodes and no hidden layers. Such a network is displayed in figure 4.

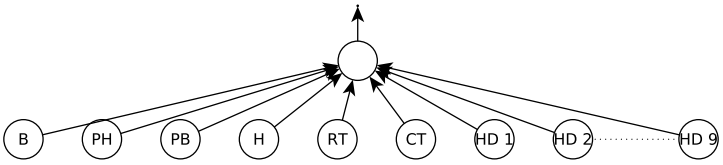


Fig. 4: Neural network for implementing a linear combination of the features discussed in this section.

3.3 Temporal Difference

Most of the time the reward for a certain action will be 0, as no lines will be cleared and the game is not yet over. In such cases it is not possible to evaluate how well a move helps towards the goal of clearing lines and preventing game over. Therefore the agent is trained using temporal difference learning (TDL)[8], which will learn it to evaluate actions based on states that follow.

If the agent uses a policy $a = \pi(s)$ to choose an action for a state s, then

$$V^{\pi}(s) = E\left(\sum_{i=0}^{\infty} \gamma^i r_i | s_0 = s, \pi\right) \tag{1}$$

is the value of a policy. Here $\gamma \in [0, 1)$, which ensures the convergence of the series whenever there is some upper (and lower) bound on r_i and is called the discount factor. This V^{π}-value takes a weighted cumulative of the rewards for each action done according to π. Since we play the online version of Tetris, there is a probability distribution for different states to follow s, as the next tetromino is selected randomly. Therefore we need to take the expected value over all possible sequences of tetrominoes. Finding a policy π which leads to a highest $V^{\pi}(s)$ for all states is hard. The concept is that TDL allows for a policy π to be found, such that $V^{\pi}(s)$ is sufficiently high for all or most of the possible states.

The temporal difference learner discussed here implements $V^{\pi}(s)$ as a neural network (NN) as shown in figure 4. TDL clasically works like this:

Algorithm 3.2: TRAINV(policy π, NN V : state $\to \mathbb{R}$)

Initialize V with random weights, typically in $[0, 0.01]$
$s_{prev} \leftarrow$ current state
according to π select an action a and play it.
$r_{prev} =$ the reward for this action.
while computational time left
do
- $t \leftarrow$ current tetromino
- $s \leftarrow$ current state
- according to π select an action a.
- $s_{after} \leftarrow V(s_a)$, where s_a is the state after playing a.
- train $V(s_{prev})$ with the target V-value $r_{prev} + \gamma V(s_a)$
- $s_{prev} \leftarrow$ current state, where a is not yet played
- play a
- $r_{prev} =$ the reward for this action.

3.4 Policy

The previous implementation of TDL tries to approximate $V^{\pi}(s)$. Each time an action is played, the state is trained by the reward just before that state and the V-value of the state reached by the next action. Allowing π to take random actions will allow the agent to explore strategies that might work. When V is sufficiently trained and π selects an action which leads to a state with the highest V-value, the agent will train on strategies that turned out to work. Both have advantages, but relying on only one is a bad idea. When selecting a random action, it is very unlikely that a line is cleared, and the NN will learn for many moves how bad they are, but only for few how well they work. By only selecting the actions that were successful in the past, the process will fail to try different strategies and chances of getting stuck in an unfortunate local optimum are high.

The solution used here is called ϵ-greedy, where $\epsilon \in [0, 1]$. An action is selected with chance ϵ as the action leading to the highest V-value. The other portion is selected randomly. For the experiments of this paper ϵ starts as 0.5 and is slowly decreased each iteration until it is 0.1.

3.5 Reward

The reward which the environment gives for certain actions is important when training the V-values. They are the only information, besides the discount factor and values from the NN itself, that is train the NN. Different versions of the Tetris games have different scoring mechanisms. Often they give points for each dropped tetromino, each cleared line or both. Obviously both clearing lines and placing more tetrominoes delays game over, and thus both have a *positive effect* on the game.

The experiments in this paper will give a small reward for each played tetromino (0.2). For one, two, three or four cleared lines at once, respectively 1, 4, 8 or 16 points are given. The reward for an action that caused game over is different for different experiments, it varied between -5 and -100 for different experiments.

The environment could also give (negative) rewards for created holes, column transitions and the pile height, since intuition will state that holes are bad since they prevent lines from being cleared until the hole is *opened*. Unfortunately, in doing so, one would clearly state how *bad* or *good* such features are, whilst this is what the agent should learn from experience. Therefore these kind of rewards are not given by the environment.

3.6 Batched

An alteration to the TDL algorithm was used for the experiments. Instead of training the network each move, first a batch of 10^4 actions and there *target* V-values were calculated. The advantage is that training the 10^4 actions and their target V-values at once, instead of just one, decreases the chance of overfitting, thus allowing to train longer on such a batch, by applying more back-propagations on the NN. The algorithm then is:

Algorithm 3.3: TrainV(policy π, NN V : state $\to \mathbb{R}$)

Initialize V with random weights, typically in $[0, 0.01]$
while computational time left
do
- $S \leftarrow$ empty training set
- **for** 10^4 played actions
  - **do**
    - $s_t \leftarrow$ the state to which a led
    - determine $V'(s_t) = r_t + \gamma V(s_{t+1})$ and add $(s_t, V(s_t))$ to S
- train V using S

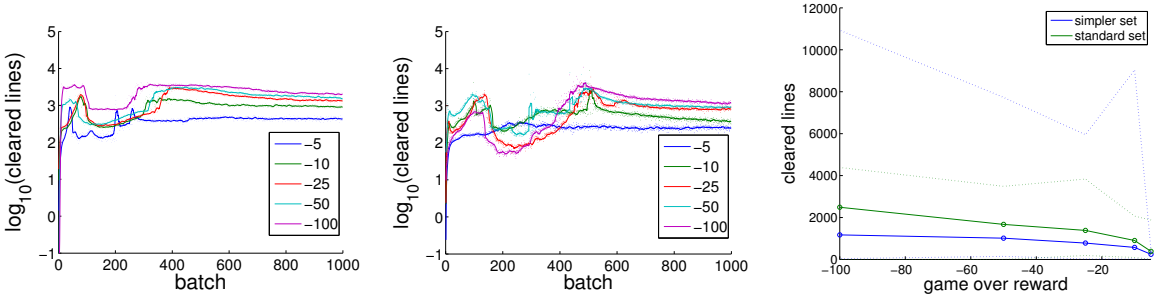
4 Results

Summarized the training method was tested for both the simpler set of tetrominoes and the standard set. For both the network in figure 4 was used and both were trained with -5, -10, -25, -50 and -100 as game over rewards. This accounts for 10 experiments. Each experiment was executed 5 times, resulting in 50 networks. All these networks where then tested by using them to play 50 games of Tetris like this:

Algorithm 4.1: PLAYTETRISBYV-VALUES(NN V : state $\to \mathbb{R}$)

while ¬game over

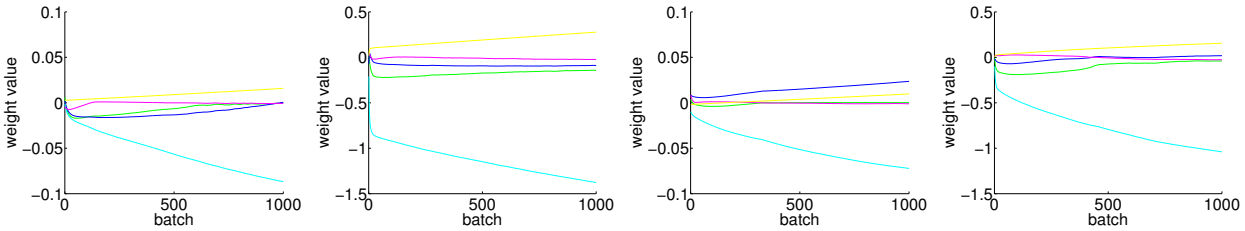
do { from all actions that do not lead to game over
play the one which leads to state s with the highest $V(s)$
if all actions lead to game over, play a random one



(a) standard set (b) simpler set (c) standard set, −5

Fig. 5: (a and b) Total amount of cleared lines for the 50 games (logarithmic scale). The dots represent the real data, where the curves are an approximation. (c) Average and extrema (dotted) of the total amount of cleared lines (50 games) over all batches.

Figure 5ab shows the amount of cleared lines for the total of 50 games. A batch indicates how many training sets of 10^4 moves have been used to train the NN. Between 150 and 400 batches, both tetromino sets perform worse. The weights for some networks are displayed in figure 6. The figure shows that a lower game over reward will lower the weights for *intuitively negative features*, like holes and column transitions faster, allowing the network to learn faster. Column and row transitions weights decrease at first. After around 150 batches the agent increases them again, indicating that *better* V-values can be found when they are bigger then their minimum around 150 batches. The fact that the network lowered these weights too much, may explain why the performance between 150 and 400 batches is significantly lower.



(a) standard set, −5 (b) standard set, −100 (c) simpler set, −5 (d) simpler set, −100

Fig. 6: Weights of the neural for both tetromino sets and game over rewards of −5 and −100. The lines are: CT (green), RT (dark blue), H (light blue), W (pink), whether the game is game over (1) or not (0) (yellow). These are the weights for the first trained network with the given settings and may differ on different runs.

5 Conclusion

In this paper we compared two versions of the Tetris games, using the standard tetromino set and the simpler one. An interesting conclusion is that a lower game over reward (−100) will

outperform a higher one (-5). This is shown in Figure 5c, it shows the average, maximum and minimum test results over all batches for different game over rewards. For this test set and for the simpler tetromino set the maximum of cleared lines is order 100 bigger for a game over reward of -100 compared to a reward of -5. Roughly a game over reward of -100 is trains to clear 10 times as many lines as a game over reward of -5.

Future work may include more experiments using different methods, since this might give better results. Results from the internet post on *El-Tetris* [1] use the landing height of a tetromino and the amount of cleared lines as features to determine the quality of an action. Using these in TDL with Q-values [9] might give similar or better results.

Using NN with different activation functions or with hidden layers might be capable of representing more complex models, allowing for more cleared lines. It seems obvious to try different types of NN, but the success of Pierre Dellacherie and El-Tetris suggest that a NN without hidden layers and linear activation nodes should be able to perform well.

The V-values should have a symmetry property: if the play field and the tetromino that is being played are mirrored, the V-value for such an action should stay the same. Exploiting this doubles the training data and might lead to faster learning and more cleared lines.

In the implementation used for the experiments, most computational time is spend calculating training sets, instead of actually training. Faster code may solve this.

References

[1] N. Böhm, G. Kókai, and S. Mandl, "An evolutionary approach to tetris," in *The Sixth Metaheuristics International Conference (MIC2005)*, 2005.

[2] S. Melax, "Reinforcement learning tetris example." `http://www.melax.com/tetris`, 1998.

[3] C. Fahey, "Tetris specifications & world records." `http://colinfahey.com/tetris/tetris.html`, 2003.

[4] D. Carr, "Applying reinforcement learning to tetris," *Department of Computer Science Rhodes University*, 2005.

[5] R. Breukelaar, E. Demaine, S. Hohenberger, H. Hoogeboom, W. Kosters, D. Liben-Nowell, *et al.*, "Tetris is hard, even to approximate," *International Journal of Computational Geometry and Applications*, vol. 14, no. 1, pp. 41–68, 2004.

[6] "T-spin on wikia.com." `http://tetris.wikia.com/wiki/T-Spin`, 2012.

[7] Y. Bdolah and D. Livnat, "Reinforcement learning playing tetris." `http://www.math.tau.ac.il/~mansour/rl-course/student_proj/livnat/tetris.html`, 2000.

[8] R. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, pp. 9–44, 1988.

[9] M. Wiering and M. Van der Ree, "Reinforcement learning in the game of othello: Learning against a fixed opponent and learning from self-play," 2012.