

Designing an AI Agent for playing Tetris using Reinforcement Learning

Sam Sarjant

Supervised by Associate Professor Bernhard Pfahringer

11th March 2008

Abstract:

When designing AI for game playing, it can be quite difficult to write AI for games that have a large number of states. A fixed policy will work in some cases, but won't allow the agent to properly overcome unseen obstacles. Using reinforcement learning for an agent's strategy allows the agent to overcome unseen obstacles by testing the result of interacting with said obstacles and learning from the outcome by modifying its policy of play.

Tetris is one such game that has a large amount of states and thus cannot easily be played by a computer. Teaching a computer how to play by other methods of machine learning (such as supervised learning) can be very difficult as finding the 'right' action to take at any given state is expensive. The aim of this project is to create an agent capable of playing the game of Tetris well enough to compete in a worldwide competition with a focus on reinforcement learning as its guide.

Introduction

Tetris has been around for over 20 years now and is still tremendously popular. The simplistic nature of the game hasn't diminished the fact that it is still an incredibly addictive and popular game. However, even though it is easy to implement (so much so that it has been used as a 'Hello World' program for learning programmers), creating AI to play it has proven to be a difficult task as the problem of Tetris has been proven to be an NP-hard problem [1]. As such, it is hard to design an algorithm for playing perfectly. So, an approach of letting the computer learn the game itself is better suited to the task.

This method of learning is known as reinforcement learning and allows an agent (computer-controlled AI) to learn how to do something via trial-and-error. In the early stages of machine learning, reinforcement learning received surprisingly little attention but lately has become a thriving avenue of research. Reinforcement learning is based primarily on letting the agent find its own method of doing things by returning rewards to the agent based on the action it took. The rewards may be immediate, or be delayed. This is one of the main problems in reinforcement learning and proves to be a problem in Tetris.

This project aims to explore the benefits of reinforcement learning used in the game of Tetris and create an agent capable of learning and playing the game well enough to compete against other agents in competition. Several methods of implementing the learning will be explored by drawing upon other AI Tetris agents (both reinforcement learning based and otherwise) as well as successful applications of reinforcement learning on other games.

Tetris Specification

The standard implementation of the game of Tetris involves a 'well' of size 10x20 in which a random piece (or 'tetromino') chosen from a set of 7 possible unique tetrominoes falls down from the top of the well at a set speed until it rests on the bottom of the well or another tetromino. The piece may be moved around horizontally or rotated by 90° in either direction so as to let the piece settle in an ideal position. Pieces may also be 'dropped' when an ideal horizontal position and rotation are decided upon and the player wishes the piece to rest on the lowest point possible.

Each tetromino is unique to one-another and each is made up of 4 'units' linked by a side, as seen in Figure 2.

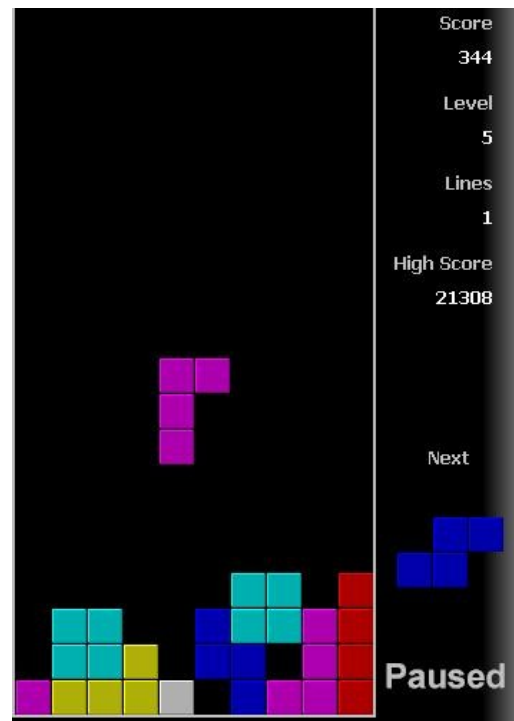


Figure 1: An example Tetris state

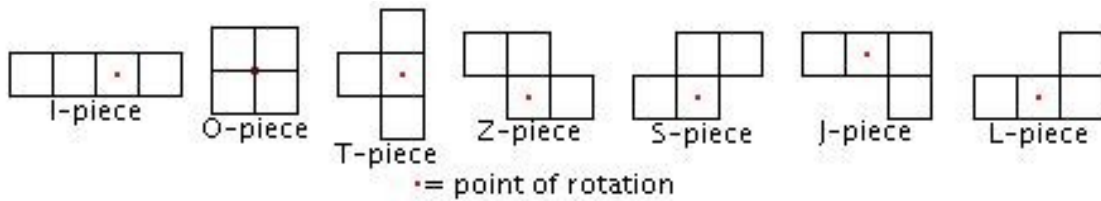


Figure 2: Tetromino structures

If the piece becomes fixed to create a completely filled horizontal line, the line disappears, the pieces above move down 1 unit, and the score is increased. The score is also increased when a piece lands, but not by as much as when making a line.

The next piece is shown at the side of the field to indicate what piece is next so the player can use this information to place pieces strategically.

If the player fills the field with tetrominoes and a new piece is spawned on top of an existing static piece, the game is over.

Differences in the RL2008 Tetris specification

- At each step, an action must be given to the game to let the piece drop; Idle, move left, move right, rotate left, rotate right, or drop.
- Only the current state of the field is given with no indication as to the location of the current piece, only what piece is falling.
- There is no next piece shown, so the player can only use the current piece for strategy.
- It is not said if score is given when a piece has rested, regardless of whether it has created a line or not. It is likely that score is given when a line is completed though.

Reinforcement Learning

‘Reinforcement learning is learning what to do--how to map situations to actions--so as to maximize a numerical reward signal.’[2] This statement essentially captures what reinforcement learning is all about. The agent is required to learn what to do by testing actions through trial-and-error and seeing what works by receiving a reward based on the action. Over time, the agent will learn an ideal set of actions to conquer a problem by storing the rewards gained from the actions and following the best path of rewards.

Reinforcement learning differs from supervised learning in that it doesn’t know if the action taken is right or not. Supervised learning can give an answer if an action is ‘right’ or not by helping the agent out with prior knowledge. Although this is a perfectly acceptable method of learning, often it isn’t known what the right action to take is, or at least it is very hard to determine the best action.

An agent utilising reinforcement learning always has a specific goal in mind. Some sort of task that it tries to constantly work towards, such as locating an object, or in the Tetris case, obtaining the highest score possible without losing the game. To get to this goal, the agent needs to take the best possible path towards it. However, in order to find the best path, the agent must explore first. This is the main problem in reinforcement learning: Exploration vs. Exploitation. An agent must explore to find a good path to take, but must also exploit that path to achieve its goal. The balance of these factors is usually dependant on the problem and this project aims to find the best possible balance for it.

RL2008 Competition

The RL Competition [12] is an annual competition which is based on using reinforcement learning on tricky problems by providing several domains for competitors to download and create agents for. The agents are used as their entry into the competition and tested against each other in the final proving run. Throughout the competition, a competitor may, once a week, run a proving run for their agent to see how well it does against other competitors by viewing the result of the run on an online leaderboard.

The software provided is based on RL-Glue [13], which is a ‘test-harness’ of sorts used for running reinforcement agents and environments around. It is multi-language and can be used for whatever reinforcement learning purposes required.

The competition timeline is as follows:

- **October 15th, 2007:** Official competition announcement.
- **November 1st, 2007:** Release of competition training software. Competitors can now begin training their agents.
- **December 1st, 2007:** Release of competition test software. Competitors can now do official competition proving runs.
- **June 1st, 2008:** Test runs begin.
- **July 1st, 2008:** End of competition.
- **July 6th-9th 2008:** Competition event at ICML.

Related Work

Reinforcement Learning has been applied to Tetris and other NP-hard games and has been met with some success. TD-Gammon, which is a backgammon playing agent using reinforcement learning and a large neural network, was one of the most successful applications, where the agent is ranked as one of the best backgammon playing entities around. It is able to compete against the best human players and is nearly equal in skill with the best fixed policy algorithm around [3]. KnightCap, which is a chess playing agent, learned its strategy by training against human players and gradually became more intelligent. It ranked as a master chess player after playing 308 games against online opponents [4].

Other work in Tetris AI includes the fixed-policy algorithm designed by Pierre Dellacherie (2003, France), which is regarded as the best one-piece AI algorithm to date, and averaged 650000 completed rows per game, with some games scoring over 2 million completed lines [5]. A reduced version of Tetris was used in the testing of an RL algorithm which had no vertical limit to the well, but limited the number of pieces [6]. The agent managed to learn well how to keep the height low using the reduced pieces. Later, another algorithm built on this one was created that achieved even better results [7].

However, full versions of Tetris using reinforcement learning haven’t achieved as well as fixed policy methods yet. A method of using noisy cross-entropy to play achieved fairly well with number of lines at about 350000 [8]. Another, using relational reinforcement learning, which saves relationships between elements rather than explicit state-action pairings, did rather poorly achieving less than 50 rows. [9]

Another paper discusses a possible application of reinforcement learning, which has a goal very similar to this project [10]. The results were quite dismal, with the agent not even achieving 20 rows [11]. The results were lacking information however, so it’s possible they may differ from shown.

Design

Bdolah and Livnat [7] received excellent feedback from their reduced Tetris strategy, so design was originally focused on a modified Top-Two-Rows strategy for full Tetris. However, the state space associated with such a strategy proved to be too large. Instead, the field could be split into several smaller tiled substates of the field stored as height mappings of the state area. Also, by truncating the contours to a maximum constant, the state space can be reduced to 5^3 (125) possible field states. However, this presented another problem of states that span over a large vertical height and could result in badly placed tetrominoes. So, instead of having fixed size substates, variable sized substates were introduced that cut off the substate when a large climb or drop occurs (see Figure 3). Special states are needed for any chasms in the playing field to ideally put I-shaped tetrominoes down.

For the agent to learn how to play, the policy of its play included a substate and a tetromino position within the substate and the reward associated with such a state. So when the piece lands, its position would be stored in the policy using the substate(s) it landed in and the agent would learn a new policy for that particular piece. An advantage of using variable sized substates is that small substates can fit inside bigger substates and so multiple updates to the policy can occur.

Because Tetris is a delayed reward problem in reinforcement learning, storing only the substate and tetromino that create a line would result in slow and short-sighted. By keeping an eligibility trace of substate-tetromino values, the line creating state and the states leading up to the line creation could be updated. To properly balance the learning though, the oldest states in the eligibility trace would receive a fraction of the reward gained based on the distance from the front of the trace.

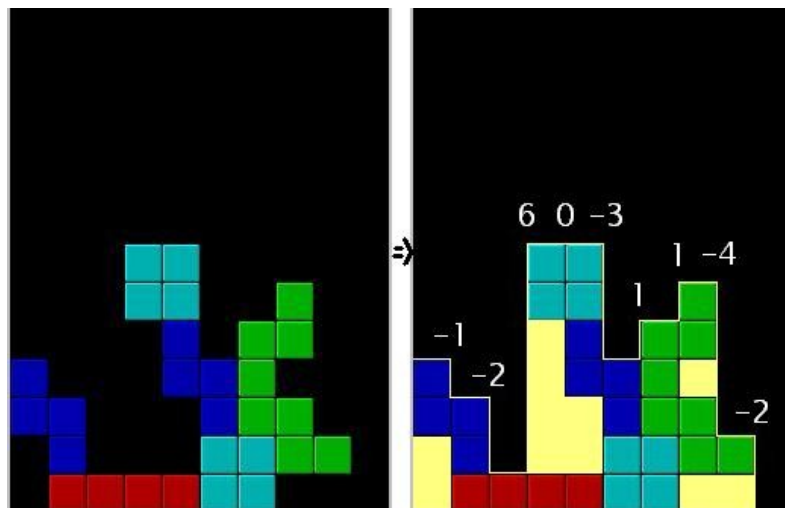
Another problem that the delayed nature of Tetris gives, is receiving any lines at all in the early stages of play. If an agent is exploring randomly and placing pieces anywhere, the probability of receiving a line-based reward is minute. However, if the agent were guided during exploration, and biased towards placing pieces in smart position in which the pieces fit well, lines could be made early on and lead to faster learning.

Another area of the problem to consider is the balance between exploration and exploitation. At the beginning, the agent would naturally be exploring, but should take greedy paths to maximise points also. The methods of ϵ -greedy, Softmax and other methods will be explored. Learning rate is another value to modify for optimal play. There are many other variables that can be changed which will need to be checked against each other.

Figure 3: As shown in the figure, the field can be represented as a series of height differences.

These differences can be split up into substates, broken by height differences of 3 or more.

$\{-1, -2, 6, 0, -3, 1, 1, -4, -2\}$
 $\Rightarrow \{-1, -2\}, \{0\}, \{1, 1\}, \{-2\}$



Completed Thus Far

As the competition started in October 2007 and finished July 2008, it was tactful to start work early on. Work began on the 1st January 2008 and the timeline of things completed up to then are as follows:

- **8-1-08:** Contoured Sub-States idea settled upon. Details of implementation drafted.
- **11-1-08:** Comprehended observations given by environment.
- **16-1-08:** Program pseudocode outlined. Basic structure complete.
- **24-1-08:** Finished JUnit test cases for all currently realised functions.
- **1-2-08:** Version 1.0 complete! A basic agent can run on the environment.
- **5-2-08:** First test run on lab machines. Results slightly buggy, but acceptable.
- **8-2-08:** Theorised variable sized substates split by large climbs or drops. Also theorised guided exploration to give the agent a jump-start.
- **13-2-08:** Variable substate capturing complete.
- **15-2-08:** 'Superstate' (a substate that encompasses a smaller substate) updating theorised.
- **18-2-08:** Superstate updating complete.
- **21-2-08:** Version 1.1 complete! Differs from 1.0 by using variable sized substates and superstates.
- **1-3-08:** Version 1.2 complete! Differs from 1.1 by adding in guided exploration (placing pieces in smart positions while exploring).
- **3-3-08:** Eligibility trace theorised for path updating.
- **10-3-08:** Modified ϵ -greedy strategy to one that uses simulated annealing.

For further updates, watch the blog progress at <http://super-sanity.com/category/honours-project-progress/>. Use the password 'waikato36' to view the posts.

Proposed Timeline

- **25-3-08:** Fix all currently known bugs and implement eligibility trace.
- **18-4-08:** Select best exploration strategy from previous lab tests. Run proving run for leaderboard.
- **18-4-08 – 30-5-08:** Prepare for interim report and presentation. Modify strategy to achieve a higher position on the leaderboard.
- **1-6-08 – 1-7-08:** Tune agent to the best it can be for competition.
- **1-7-08:** Competition closes.
- **10-7-08:** Based on results, either further improve agent, or begin final report. Also prepare for 520 Conference.
- **15-10-08:** Hand in report.

Required Resources

This project requires the use of a Linux system to easily run the RL Competition software. The software itself is needed from the RL Competition website and the Java programming language is required also. Though not absolutely required, a Java programming IDE such as Eclipse would be helpful to development. Some form of data graphing software would help in algorithm comparison also. All of the above can be obtained freely from the internet, but a pre-set up Linux OS would be helpful.

All of the above have already been supplied, and should suffice for the project's needs and requirements.

Evaluation

The success of the project can be evaluated by answering these questions:

- How did the agent place on the RL Competition leaderboard?
- How did the agent's results compare to average human performance on Tetris?
- How did the agent's results compare to existing AI Tetris agent's results?

Conclusion

Though others have attempted to write AI agents for Tetris before, and some have succeeded, this project's research will not only further progress into AI Tetris players, but also into applications of reinforcement learning. Regardless of whether the agent of this project wins the RL Competition 2008, it will promote research into new methods of reinforcement learning for all competitors and add to the collective pool of knowledge for all to later utilise.

References

- [1] Ron Breukelaar, Erik D. Demaine, Susan Hohenberger, Hendrik Jan Hooeboom, Walter A. Kusters, David Liben-Nowell. Tetris is hard, even to approximate. *International Journal of Computational Geometry & Applications*, April 8, 2004. URL: http://www.liacs.nl/~rbreukel/publications/tetris_is_hard.pdf
- [2] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. 1998. URL: <http://www.cs.ualberta.ca/%7Eesutton/book/ebook/the-book.html>
- [3] Gerald Tesauro. Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, March 1995 / Vol. 38, No. 3. URL: <http://www.research.ibm.com/massive/tld.html>
- [4] Jonathan Baxter, Andrew Tridgell, Lex Weaver. KnightCap: A chess program that learns by combining TD(λ) with game-tree search. 10 Jan, 1999. URL: http://arxiv.org/PS_cache/cs/pdf/9901/9901002v1.pdf
- [5] Colin P. Fahey. Tetris AI : computer plays Tetris. 2003. URL: <http://colinfahey.com/tetris/tetris.html>
- [6] Stan Melax. Reinforcement Learning Tetris Example. 1998. URL: <http://www.melax.com/tetris.html>
- [7] Yael Bdolah and Dror Livnat. Reinforcement Learning Playing Tetris. 2000. URL: http://www.math.tau.ac.il/~mansour/rl-course/student_proj/livnat/tetris.html
- [8] István Szita and András Lörincz. Learning Tetris Using the Noisy Cross-Entropy Method. 12 Jan, 2006. URL: <http://eotvos.elte.hu/~szityu/papers/SzitaLorincz05Learning.pdf>
- [9] Kurt Driessens. Relational Reinforcement Learning. PhD thesis, Catholic University of Leuven, 2004. URL: <http://www.cs.kuleuven.ac.be/kurtd/PhD/>
- [10] Donald Carr. Applying Reinforcement Learning to Tetris. May 30, 2005. URL: http://colinfahey.com/tetris/ApplyingReinforcementLearningToTetris_DonaldCarr_RU_A_C_ZA.pdf
- [11] John Åsberg. Artificial Tetris Players. November 14 2007. URL: <http://frit.fy.chalmers.se/cs/cas/courses/seminar/071126/asbergpdf>
- [12] RL2008 Competition. URL: <http://rl-competition.org/>
- [13] RL-Glue. URL: <http://rlai.cs.ualberta.ca/RLBB/top.html>