Learning to play Tetris with Big Data: Project Report

CS3243 - Introduction to Artificial Intelligence (Group 14)

Yogesh Kumar **Nhan Nguyen** **Kristoffer Lerbæk Pedersen** **Gustav Kvick**

April 16, 2016

INTRODUCTION

- The purpose of the project is to create a utility based agent whose goal is to maximize the number of rows removed up to termination of the game.
- Tetris is a video game which was originally invented by Russian programmer Alex Pajitnov in 1985, and has been popular ever since. The blocks, usually called *Tetrominos*, fall from the top of the board to the bottom, and the player tries to clear the rows by filling them with no holes.

Fig 1: Tetrominos

- The agent uses a heuristic function to approximate the utility of a state. The agent's strategy to play Tetris is improved after every move while in learning mode, using the Least-Squares Policy Iteration algorithm (LSPI).
- In this report we have given a gist of our design, the agent's salient features and its working.

STRATEGY OF THE AGENT

- Given the current piece to place, the agent will make the decision, which it deems will result in the best possible state. When a piece is to be placed, the agent simulates and evaluates all the possible boards that would result from all the possible moves(orientations and positions) of the piece, and then chooses the action that leads to the best valued game board (greedy strategy).
- We note $s \in S$ a state of the game, where s represents the configuration of the board at a given time. The evaluation function $V : S \to \mathbb{R}$ maps a state s to a number that estimates the value of the state s.
- The evaluation function V is a combination of the feature functions, using a weighed, linear sum. If we let f_i and w_i, with $i = 1...n$, denote the n feature functions and their respective weights. Each f_i maps a state s to a real value. The evaluation function V is then defined as:

$$V(s) = \sum_{i=1}^{n} w_i f_i(s) \tag{1}$$

FEATURE FUNCTIONS

- The following features are evaluated in the heuristic function.
  1. **Landing height**
     Height at which the current piece will land. Implemented by checking the column height where a piece may fall and adding the piece height of that specific column, returning the highest value.
  2. **Eroded pieces**
     Contribution of last piece to the cleared rows times the number of cleared rows. Implemented by comparing the total number of deleted rows after the move with the current state, returning the difference.
  3. **Row transitions**
     Number of filled cells adjacent to empty cells, summed over all rows. Implemented by traversing each row, checking whether each filled cell is different from the cell next to it, or the border is next to an empty cell on that row, and counting up if it is.

4. **Column transitions**
   Number of filled cells adjacent to empty cells, summed over all columns. Implemented by traversing each column, checking whether each filled cell is different from the cell above it, and counting up if it is.
5. **Number of holes**
   Number of empty cells with at least one filled cell above. Implemented by traversing each column, checking if we have an empty cell directly under a filled cell, and counting up if it is.
6. **Cumulative wells**
   Sum of the accumulated depths of the wells. Implemented by traversing through each column, comparing the height to the height of the lowest of the two neighbors. Boundary columns are always compared to the neighbor towards the middle, as the edge is considered a column of maximum height.
7. **Covered empty cells**
   Number of empty cells below one or more filled cells in the same column, summed over all columns. Implemented by traversing each column, counting any empty cell lower than its column top.
8. **Average column height**
   The average height of all the columns. Implemented by adding the top of each column, dividing by the number of columns.

◇ The following features were also implemented, but removed again due to them being either unnecessary or counterproductive.

1. **Holes depth:** Depth of covered holes, summed over all columns.
2. **Rows with holes:** Number of rows with a covered hole in it.
3. **Pieces accommodated:** Number of pieces which fit in to the surface without creating a hole.
4. **Moves accommodated:** Total number of possible moves not creating a hole.
5. **Total height differences:** Summed heigth difference of all columns
6. **Max column height:** Heigth of the highest column
7. **Min and max column difference:** Difference between the highest and the lowest column
8. **Standard deviation:** Standard deviation of the column heights
9. **Deepest well:** Depth of the deepest well
10. **Total well depth:** Summed depth of all wells
11. **Filled cells:** Total number of filled cells

ADJUSTING THE WEIGHTS

We use the Least-Squares Policy Iteration algorithm ($LSPI$) to adjust the weights of the agent. LSPI is a reinforcement learning algorithm designed to solve control problems. It uses value function approximation to cope with large state spaces and batch processing for efficient use of training data. The learning is executed as follows:

repeat

$w = w'$

for each $(s, a, r, s') \in S$

$$\boldsymbol{B} \leftarrow \boldsymbol{B} - \frac{\boldsymbol{B}\phi(s,a)(\phi(s,a) - \gamma \sum_{s' \in S} P(s,a,s')\phi(s', \pi(s')))^\intercal \boldsymbol{B}}{1 + (\phi(s,a) - \gamma \sum_{s' \in S} P(s,a,s')\phi(s', \pi(s')))^\intercal \boldsymbol{B}\phi(s,a)}$$

$$\boldsymbol{b} \leftarrow \boldsymbol{b} + \sum_{s' \in S} P(s,a,s')\phi(s,a)R(s,a,s')$$

$$w' \leftarrow \boldsymbol{B}\boldsymbol{b}$$

until $(w \approx w')$

Where $\phi(s,a)$ denotes the feature vector of state s after taking action a

$\pi(s')$ returns the best action in state s' according to current policy

γ is the discount factor for future rewards

SCALING THE ALGORITHM TO HANDLE BIG DATA

Parallelization can be used to increase learning speed. Each complete Tetris game will then run on an individual thread. The speedup is dependent on the number of physical cores present on the machine running the learning algorithm.
Parallelisation on 8 threads over 4 physical cores led to a 4.13x speedup over running the algorithm on one thread. The processor in the machine used for this benchmark was an Intel Core i7-2670QM @ 2.2 GHz.

1 Thread	8 Threads	Speedup
1,480s	358s	4.13

Table 1: Average time taken to complete 6,720,000 states with 8 features

We let the agent learn only 6,720,000 states because the weights converge really quick and produce good results (mean: ~2,000,000 rows) after learning as low as 6,720,000 randomly generated states.

EXPERIMENTAL RESULTS AND ANALYSIS

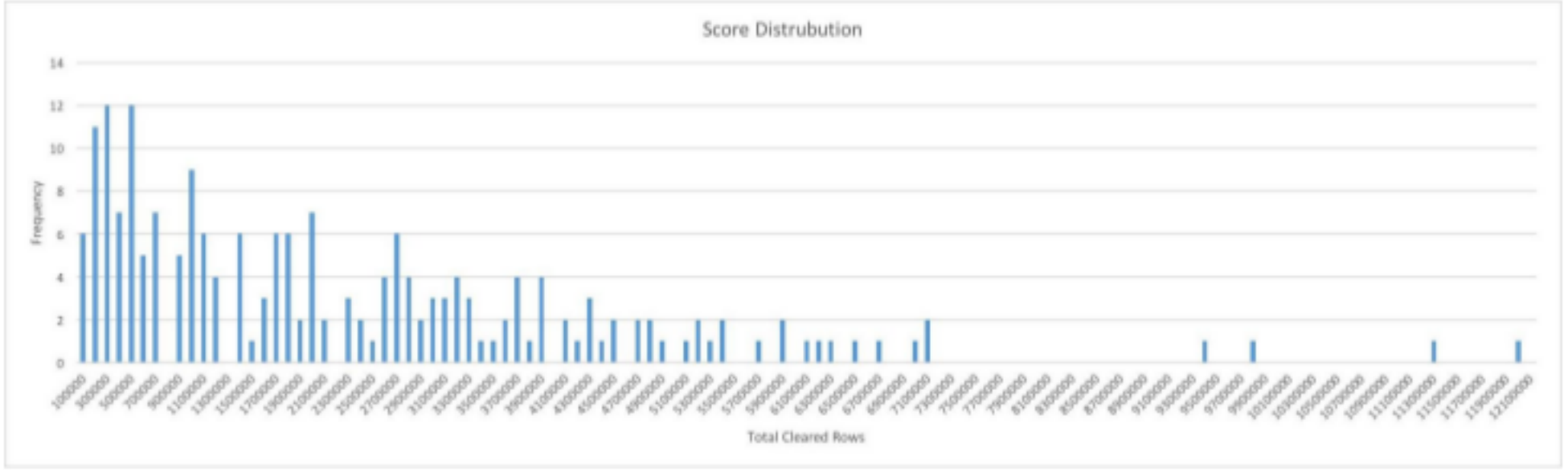

Fig 2: Total number of rows cleared vs. frequency graph for 200 games played

Games Played	Mean	Median	Maximum	Minimum
200	2,248,824	1,703,188	11,955,679	15,526

Table 2: The Mean, Median, Max. and Min. number of rows cleared using the best policy

As we can see from Fig 2 most of our values are distributed between 0 to 4 million, and 50% of the total outcome clears above 1.7 million rows. Fig 2 indicates as well that the score distribution exponentially decreases, it is more common that our solution clears 200 thousand rows instead of 3.2 million rows, even though they are as far way, based on the number of cleared rows, from the median value.
Given if our score distribution is exponentially distributed our solution would have an expected value, E(x) of **2.45 million** and a standard deviation of 1.7 million.

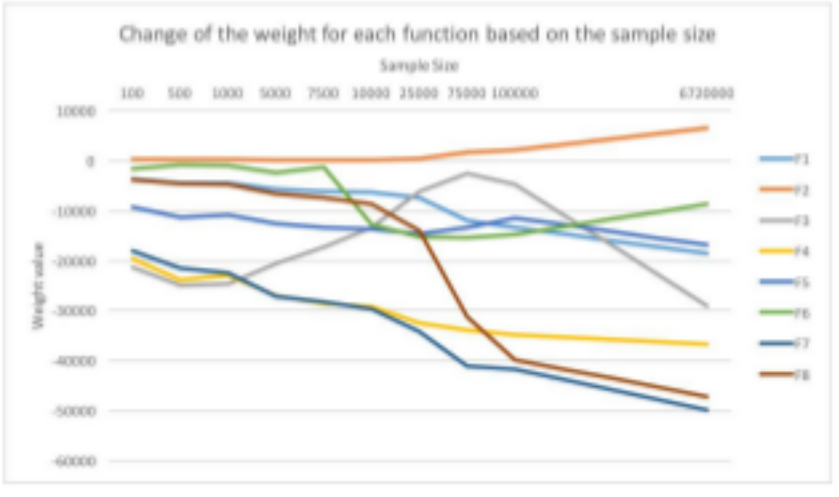

Fig 3: Learning of Weights

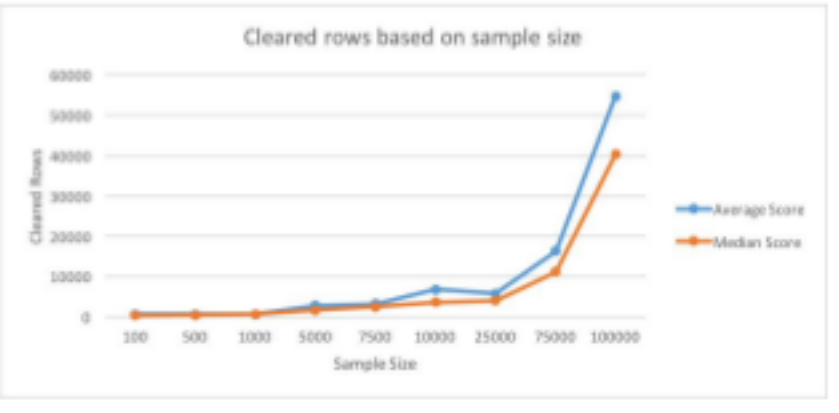

Fig 4: Average rows cleared vs games played

From Fig 3 we can see the function weight evolve based on how big the learner sample size is. One can clearly see that all functions weight, except function 3, move towards our best policy weights the larger the learning sample size is.
As we can see in Fig. 4, it shows an average improvement of performance by the learning agent . This was expected as the learning agent's weights get better with the amount of states learned.

Feature #	Heuristic	Weight	Contribution
F1	Landing Height	-18,632.774652174616	8.72%
F2	Eroded Pieces	6,448.762504425676	3.02%
F3	Row Transitions	-29,076.013395444257	13.61%
F4	Column Transitions	-36,689.271441668505	17.18%
F5	Number of holes	-16,894.091937650956	7.91%
F6	Cumulative Wells	-8,720.173920864327	4.08%
F7	Covered empty cells	-49,926.16836221889	23.38%
F8	Average column height	-47,198.39106032252	22.10%

Table 3: Best policy learned by the agent on sample space of 6,720,000 states

Table 3 tabulates the weights of the different feature functions, as well as a breakdown of their respective contributions. It is worth mentioning that contribution does not necessarily correlate with importance, as some functions may produce much lower values than others, which is why they have to be weighed higher, and vice versa.

CONCLUSION

In conclusion, we that believe our solution, which uses Least-Squares Policy Iteration (LSPI) is generally better than a Genetic Algorithm (GA), as our learner manages to converge to a relatively good policy using only a few iterations of learning. Since LSPI does not rely on the mutation chance, we believe that it is more deterministic than GA. Therefore, the results from our solution is quite satisfactory, despite the shorter time of learning.

Another distinguishing point about our solution is that we are able to generate random unique states to feed to the learner, so that it can learn from a bigger sample space and hence produce a better policy, as seen in Fig. 3. A fully generated state is compressed into an array of numbers so it does not take up too much memory to store the state during learning.

In addition, we implement parallellization in both our learner and evaluator/player to speed up the learning process. We were able to implement our solution to synchronise on multiple cores, making it easy to integrate into to a distributed system and facilitate handling of big data. We are able to allow our learner to learn from a big sample space of 6,700,000 random states in less than 15 minutes. Moreoever, we are able to allow a number of players to evaluate a given policy and automatically write back a report file.

Last, but not least, we believe that it is noteworthy that our agent, in a few cases, managed to clear almost **12 million** rows cleared in just 200 test cases, which is a lot higher than we had dared hope for, when we started this project. The average case of more than **2 million** rows cleared has also exceeded our expectations by a factor of more than 10, which leaves us no choice but to conclude that this projects has been a success.

REFERENCES

1. Michail G. Lagoudakis and Ronald Parr. Least-Squares Policy Iteration. *Journal of Machine Learning Research 4 (2003) 1107-1149.*
2. Amine Boumaza. How to design good Tetris players.