LEARNING TO PLAY TETRIS WITH BIG DATA

CS3243 - Introduction to Artificial Intelligence (Group 39)

Nguyen Cong Thanh A0161307N
Huynh Thanh Duc Anh A0161312X
Le Trung Hieu A0161308M
Agus Sentosa Hermawan A0161315R
Do André Khoi Nguyen A0161346J

April 21, 2018

1 Introduction

Tetris is a tile-matching puzzle game played on a two-dimensional grid. In each turn, an object of different shapes, called Tetriminos, will fall from the top, which the player can move or rotate. The objective of the game is to manipulate these Tetriminos to create full horizontal lines so that these lines get destroyed.
We have successfully built a Tetris agent that **can clear millions of lines** before the game terminates. This report includes the description of our agent and its performance analysis.

2 Strategy of the Agent

At each turn, based on a given Tetriminos, the agent evaluates the utility of all reachable states from the current state and then selects the move that leads to the state with the highest utility. Utility of a state is the weighted sum of all features, which are explained below.

3 Features

We implement 8 features modified from the features introduced by Dellacherie (2003) and Thiery (2009) [1]:

(a) **Landing bottom:** The height where the **bottom** of the current piece fell (row 6 in Fig 1).

(b) **Landing top:** The height where the **top** of the current piece fell (row 10 in Fig 1).

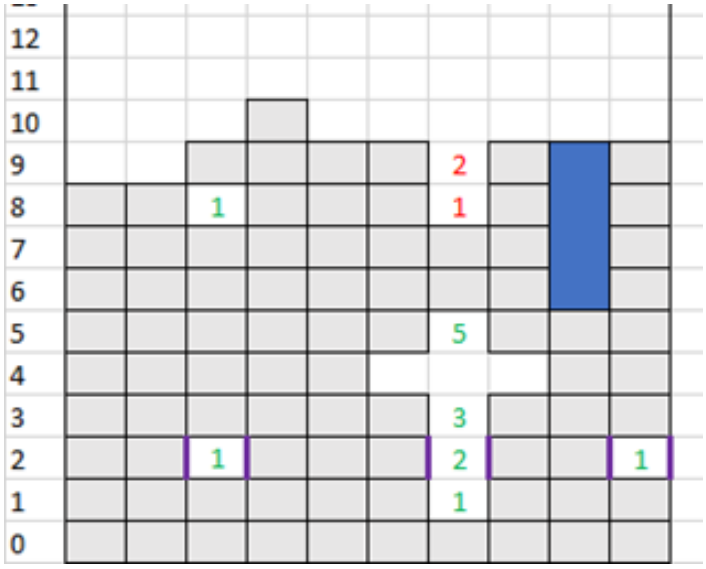

Figure 1: Example of a Tetris grid at one turn

(c) **Line cleared:** Number of lines cleared in one move (2 in the example: rows 6 and 7).

(d) **Number of holes:** Total number of holes in the grid. A hole is defined to be an empty cell with at least one filled cell above it in the same column. There are 10 holes in Fig 1.

(e) **Sum of outer wells:** A well is a series of empty cells counting down from an empty cell that is horizontally adjacent to two occupied cells. The sum of outer wells is the accumulated well depths of all wells that are **above** the top of a column. Outer well depths are numbers in red color. In Fig 1, the outer well sum is $2 + 1 = 3$.

(f) **Sum of inner wells:** The sum of inner wells is the accumulated well depths of all wells that are **below** the top of a column. Inner well depths are numbers in green color. In Fig 1, the inner well sum is 5+3+2+1+1+1+1 = 14.

(g) **Row transitions:** The number of occupied cells adjacent to empty cells, summed over all rows. The two borders are also counted as occupied cells. Take row 2 for example, the row transitions are purple thick borders.

(h) **Column transitions:** Same as row transitions, but summed over all columns.

4 Teaching the Agent

4.1 Training Strategy

We start training with genetic algorithm (GA) using quasi-random weight vectors (called genes from now on for ease of reference) until we have a good set of genes that can clear hundreds of thousands of lines. After that, we continue training using the noisy cross-entropy (NCE) method. The cross-entropy (CE) method is an algorithm to solve global optimization. At each iteration, we select $\lfloor \eta n \rfloor$ best performing genes, calculate the mean μ and deviation σ of the selected genes, then generate a new generation of genes from a multivariate normal distribution $N(\mu, \sigma^2)$ (refer to Fig 2 for pseudo-code).

Input: population size n, proportion η, generation g, noise function Z
Initialize: Set μ and σ to the mean and deviation of the genes got from GA.
while true **do**

1. Generate a random sample of n genes from $N(\mu, \sigma^2 + Z(g))$
2. Evaluate the fitness score for each gene i
3. Select $\lfloor \eta n \rfloor$ best-performing genes
4. Update μ and σ:
   $\mu(j) = \frac{1}{\lfloor \eta n \rfloor} \sum_{i=1}^{\lfloor \eta n \rfloor} w_i(j)$
   $\sigma^2(j) = \frac{1}{\lfloor \eta n \rfloor} \sum_{i=1}^{\lfloor \eta n \rfloor} [w_i(j) - \mu(j)]^2$

Figure 2: NCE algorithm pseudo-code

In this project, we use $n = 100$, $\eta = 0.1$, $Z(x) = e^{-\frac{x+100}{10\pi}}$.

4.2 Considerations

4.2.1 GA - NCE Hybrid

CE is a very promising training strategy. However, it has two large drawbacks. Firstly, with a small population size, it can be very hard for CE to get a good sample. Secondly, CE usually converges early into a bad peak. To counter these problems, we run GA, which usually produces an acceptable range of good genes but converges more slowly, to get a good sample, then pass that sample to CE. Furthermore, to prevent CE from premature convergence to local optimum [3], we inject some noise into the deviation. After some trial-and-error, we find out that a decaying noise (as shown in the formula) results in the production of the best performing agents.

4.2.2 Gene Normalization

At each state, the agent is only interested in finding which move is better than the rest, so multiplying a gene by a constant will not change the ranking of the moves. This may introduce a standstill where the training strategy just keeps adding on value to the weights without actually changing its performance. To counter this, we normalize all genes (dividing a gene's components by its vector length).

4.2.3 Higher Game Difficulty

As the agent gets more competent, the fitness evaluation time becomes longer, getting up to several hours or even a day per iteration. To reduce the evaluation time, we increase the frequency of Z and S pieces as these pieces are the hardest to play. Making Z and S appear three times more often compared to other pieces **reduces the evaluation time by at least 3000 times** while keeping the same agent performance. An agent that can clear 800 lines on average in a game with many Z and S can clear several million lines with uniform piece distribution.

4.2.4 Sub-random Initialization for GA

Instead of using pseudo-random numbers to imitate random points for initialization, it is more important that **the points are as evenly distributed as possible**. Hence, with library support, using Quasi-random generator so that the

points are designed to maximally avoid each other is a more appropriate choice for initialization as it obtains a better starting point. [2]

4.2.5 Fitness Landscape

The score of each bot follows an exponential distribution, so it is **necessary to re-evaluate** the same bot multiple times in each generation and takes the average out of them to get the correct estimation of its score. Our chosen **number of evaluation in training is 15 times**. Besides, we save the seed for the Random controller to ensure that **different bots play exactly the same game within an iteration**.

4.2.6 Look-Ahead Mechanism

Look-Ahead utility value of a state is the number of subsequent legal moves that result in a loss and the sum of utility values of all of its subsequent legal moves. To save computation time, initially we only used look-ahead when the highest column of a state is over 14. However, we notice that this implementation still drags the running time of our bot down heavily, so we decide to disable this functionality as we believe our agent is competent enough without look-ahead. The functionality is still included in our source code for your reference.

5 Scale to Big Data

5.1 Multi-threading

To speed up learning speed, we introduce parallelization to the fitness evaluation step. In each generation, the performance of many genes are gauged concurrently, and each runs on an individual thread. The results will then be combined together at the end to select the best genes. On an AWS c5.4xlarge instance, running on 15 threads results in a 4.55 speedup over running on a single thread.

1 Thread	15 Threads	Speedup
686s	151s	4.55

Table 1. Average time taken to complete the first 20 generations with GA

5.2 Vectorization

Vectorization is a type of parallel processing which makes use of single instruction, multiple data. The idea is to replace for loops with vectorization to increase computational speed:

- Loop: **for** i **in** 0...4 **do** z[i] = x[i] + y[i]
- Vectorization: z[0:4] = x[0:4] + y[0:4]

We experimented with ND4J library - which is similar to Numpy package in Python - which supports vectorization. Due to the lack of time, we could not fully vectorize all for-loops. However, by trying to vectorize some programs, we notice that it does help make the running time at least two times faster.

6 Results and Observation

After running GA for an hour, we got a set of genes that can clear at most 1.5 million lines. Then we continued to train this set of genes using NCE. After 4 hours, the initial set of genes evolved through 160 generations and we got our currently best-performing gene below:

Feature	Weight
Landing bottom	-0.163239748068033
Landing top	-0.213181276719493
Lines cleared	0.141117231058684
No. of holes	-0.574190533892977
Outer well sum	-0.274276596550364
Inner well sum	-0.204178152956631
Row transitions	-0.177407261061924
Column transitions	-0.655673684049796

Table 2. Best-performing weight vector

Across 100 games played using this gene, we achieved the following impressive statistics:

Mean	Min	Max	SD
5,685,845	261,500	25,721,463	5,414,540

Table 3. Number of rows cleared distribution

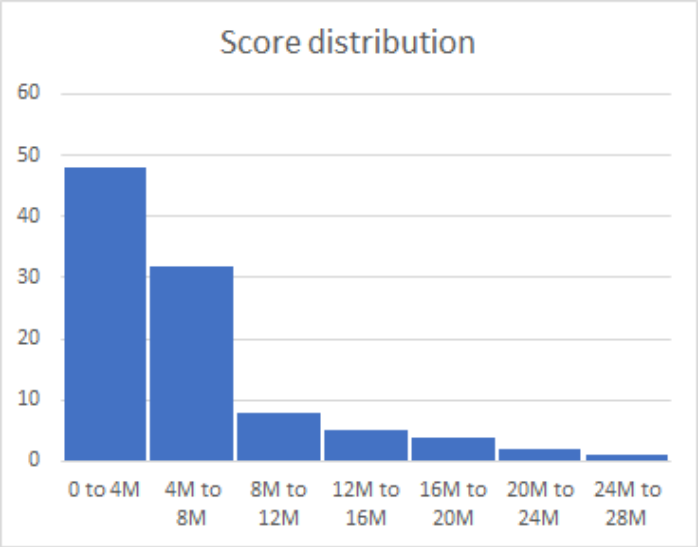

Figure 3: Score distribution across 100 games

From Fig 3 and the fact that the mean score and standard deviation have similar values, the score likely follows an exponential distribution.

It is quite impressive that while the agent can only clear 800 lines on a game with high frequency of S and Z pieces, it can go up to 25 million lines on a normal game.

While the long evaluation time prevented us from testing newer genes in normal games with uniform piece distribution, we believe that they can do at least two times better compared to this gene when we consider the training results of NCE (Fig 4).

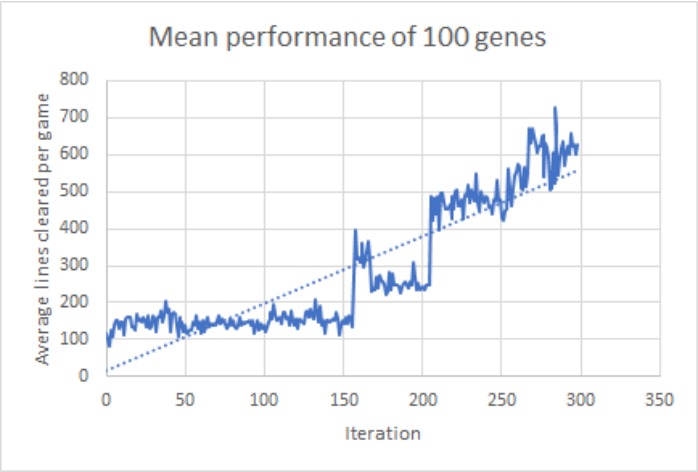

Figure 4: Learning curve of NCE

7 Conclusion

Our aim in this project is to show that a very simple optimization method, like NCE, when used in complement to another optimization method like GA can produce well-performing agents in a significantly shorter time compared to using only one optimization method.

We have shown how to use vector normalization to avoid situations where the weight vector keeps growing in size without actually changing its performance. We also discuss how increasing the game difficulty can drastically reduce the agent training time while keeping the same agent performance. Higher frequency of S and Z pieces allows the agent to learn in a very short time, up to three to four orders of magnitude (10^3 to 10^4 times) faster compared to training using uniform piece distribution.

In addition, to scale up to big data, parallelization is our main tool for this project. Parallelization is introduced in our fitness evaluation step by measuring many genes concurrently and then combining them to calculate the fitness value for each gene. We also try to vectorize some components in our training program so that we can parallelize those components. In the end, we get a speedup of 4.55.

To wrap it up, we believe this project was successful given we have produced an agent that can clear over 25 million lines with only about 5 hours of learning time.

References

[1] A. Boumaza. How to design good tetris players. 2013.

[2] H. Maaranen, K. Miettinen, and M. M. Mäkelä. Quasi-random initial population for genetic algorithms. *Comput. Math. Appl.*, 2004.

[3] I. Szita and A. Lörincz. Learning tetris using the noisy cross-entropy method. *Neural Comput.*, 2006.