Tetris Agent 06

Inaba Kazuhiro (A0140136W)

Julian Chan Siu Wah (A0135810N)

Kwok Jun Kiat (A0140114A)

Praveer Tewari (A0140124B)

Teo Ming Yi (A0146749N)

1. Agent’s strategy

1.1 Playing strategy

We used a utility-based agent in which we approximated the utility function with a heuristic function. The heuristic function is a weighted linear function that evaluates a move based on some characteristic features (x_1 to x_7) of the next state of the Tetris field:

$$\boxed{\begin{aligned} f(n) = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 \\ +w_5x_5 + w_6x_6 + w_7x_7 \end{aligned}}$$

The features we chose are as follows.

x_1 : **Sum of the absolute differences in height between adjacent columns**

$x_{2:}$: **Maximum column height**

x_3 : **Number of rows cleared**

x_4 : **If the game has been lost** - +1 If the game has not ended, -1 otherwise.

x_5 : **Number of holes** - a hole is an empty slot with a non-empty slot above it in the same column.

x_6 : **Sum of pit depths -** pit depths measure the height difference between a column and its shortest adjacent column. Pit depths are only taken into account if they are more than 2.

x_7 : **Average difference between the height of a column and the mean column height.**

In addition, we used the minimax algorithm as our playing strategy. Our “adversary” in this case is the program that generates the next Tetris piece. Our minimax playing algorithm probed two layers, returning the minimax (scored with the heuristic function) of all possible states following a certain move. The move associated with the minimax choice is then chosen as the next move.

Our minimax algorithm also made use of alpha-beta pruning.

1.2 Learning strategy

To learn the weights, we chose to run a genetic algorithm similar to [1] on 8 populations, each containing 8 randomly initialized sets of weights. For every generation, we played the game with each weight set and obtained its fitness value. We used the number of lines cleared as the fitness value. Then, we used the roulette wheel selection algorithm, where each set is selected to be the next candidate of its respective population with a probability proportionate to its relative fitness value.

To promote diversity and avoid local maxima, the selection process was conducted between all 8 populations every 100 generations. After selection, a random crossover point between each adjacent pair of sets would be chosen, from which they will perform an exchange of weights. Each weight also had an independent probability of 0.07 to mutate into a random value.

2. Agent’s performance

The weights learned from our genetic algorithm are as follows:

w1:-0.08075476343626997
w2: 0.06512710745251571
w3: 0.026528301544375688
w4: 0.9841333927818585
w5: -0.529638056344585
w6: -0.1136445616744991
w7: -0.27466748067531854

With these weights, we experimented with 3 types of playing strategies: local search, expectimax and minimax.

2.1 Local Search

We used our heuristic function to score the next state from all the possible moves. The move with the highest score is then made. The results showing the number of line cleared are tabulated below:

Number of games played	100
Mean	192,451.25
Median	130,328.5
Highest	753,811
Lowest	5,122

2.2 Expectimax

The expectimax playing algorithm that we used probed two layers, returning the sum of the scores (scored with the heuristics generated by the learning algorithm) of all possible states following a certain move. Conventionally, expectimax playing algorithms take the mean of the scores. However, since the number of possible Tetris pieces that can be generated next is constant, comparing the sum allowed our program to have 1 less division operation.

The best of all the cumulative scores is then chosen, and the associated move is made.

However, the runtime of the expectimax algorithm was too slow to gather any real data, with a single game taking up to 21 hours and still not reaching completion. This playing strategy was therefore deemed infeasible.

2.3 Minimax

This was our chosen playing strategy, as mentioned earlier. We found the minimax algorithm to be about 8 times faster than expectimax due to alpha-beta pruning. The results for our minimax algorithm are tabulated below:

Number of games played	15
Mean	1,510,327.5
Median	1,198,257
Highest	3,562,143
Lowest	337,524

As shown above, our minimax strategy performed way better than local search strategy. In addition, it runs significantly faster than the expectimax algorithm, hence we decided to choose it as our playing strategy.

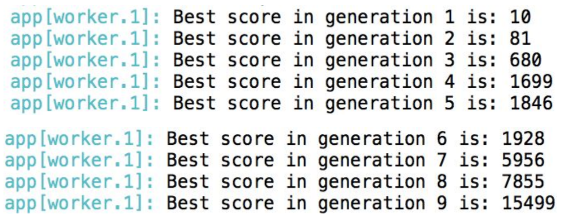
3. Discussion

3.1 Analysis of genetic algorithm

We measure the learning performance of our genetic algorithm by monitoring 2 aspects: the rate of improvement, and the number of games played within a given time.

Default heuristics - our initial attempt was to learn the optimal weights for the default heuristics given in the project document. The algorithm was able to learn by playing a large number of games. In particular, we used a population of size 100 and it clears 100 generations within a couple of minutes. However, the rate at which it improved every generation was minimal and it did not go past the score of 600 even after 300 generations.

Our 7 heuristics - after a few trial and errors, we chose this set of features as the rate of improvement was drastically better than the default heuristics. With a population of 20, there were sets that exceeded the previous limit of 600 by the 3rd generation, subsequently scoring over 150,000 in the 9th generation. By the 100th generation, there were already sets that scored above 700,000.



```
app[worker.1]: Best score in generation 1 is: 10
app[worker.1]: Best score in generation 2 is: 81
app[worker.1]: Best score in generation 3 is: 680
app[worker.1]: Best score in generation 4 is: 1699
app[worker.1]: Best score in generation 5 is: 1846
app[worker.1]: Best score in generation 6 is: 1928
app[worker.1]: Best score in generation 7 is: 5956
app[worker.1]: Best score in generation 8 is: 7855
app[worker.1]: Best score in generation 9 is: 15499
```

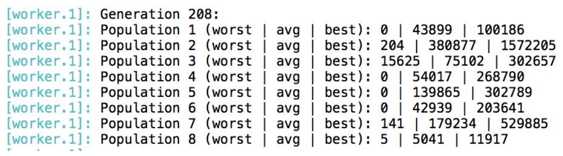
However, each generation took significantly longer as the games lasted longer, taking anywhere from 40 to 60 minutes to complete. Given a day, it played approximately 576 games.

Parallel learning - to increase the number of games it plays within a given amount of time, we utilized 8 cores of the server we ran our code on, running the genetic algorithm on each set of weights in parallel. With the same population size of 20, the algorithm ran in 8 to 12 minutes, a 5 times improvement in speed. Given a day, it played approximately 2,880 games.

With an increase in the number of games played, we were convinced that our scores were stuck at a local maximum of 1,000,000 after around 150 generations.

Distributed selection - we felt that early convergence was largely due to the population gradually losing diversity. To address this issue, we ran the genetic algorithm on 8 different populations in parallel. After every 100 generations, selection was done in between these populations to promote diversity.

As a result, improvement rate was better and we began to observe sets that scored above 1,000,000.



```
[worker.1]: Generation 208:
[worker.1]: Population 1 (worst | avg | best): 0 | 43899 | 100186
[worker.1]: Population 2 (worst | avg | best): 204 | 380877 | 1572205
[worker.1]: Population 3 (worst | avg | best): 15625 | 75102 | 302657
[worker.1]: Population 4 (worst | avg | best): 0 | 54017 | 268790
[worker.1]: Population 5 (worst | avg | best): 0 | 139865 | 302789
[worker.1]: Population 6 (worst | avg | best): 0 | 42939 | 203641
[worker.1]: Population 7 (worst | avg | best): 141 | 179234 | 529885
[worker.1]: Population 8 (worst | avg | best): 5 | 5041 | 11917
```

The number of games it plays within a given amount of time was also relatively large. Each population was of size 8; 64 sets of weights in total. Each generation completed within 20 to 30 minutes, and approximate 6x speedup compared to our non-parallel implementation. Given a day, it played approximately 3,686 games.

3.2 Ways to increase rate of learning

As mentioned in 3.1, we managed to run the algorithm on each set of the population in parallel through multithreading. As a result, it managed to play approximately 2,880 games in a day.

The number of games it plays within a given amount of time can be further improved if we run this algorithm on different populations, implemented on a network of distributed systems. Suppose we ran it on a distributed system of 5 servers, we could potentially achieve a 5 times increase in speed.

3.3 Analysis of playing performance

As far as we have experimented, the set of features that we are using yields the best results. One might be inclined to think that adding more features to the heuristic function would lead to better results as more factors are being taken into account. The learning algorithm would eventually “figure out” the optimal weights for all the features, even if the feature prove to be not useful, the learning algorithm would eventually learn to assign it with a weight of close to zero. However, while this might work in theory, our experimentations found that this was not the case as most features we added did not improve performance and some even led to a worse performance.

One possible reason for this is that in the shorter run, introducing more features would lead to a greater chances of error, caused by assigning the wrong weights to features. Perhaps if we let the learning algorithm run for a longer period of time, maybe a few weeks to even a few months, it might eventually learn the optimal weights for the features and improve performance.

In addition, our group decided to employ the minimax algorithm, even though the Tetris game is not adversarial in nature, with an opponent trying to minimize the player’s payoff. Expectimax would have been a better choice given that the next Tetris piece is generated randomly with each Tetris piece having an equal chance of being the next piece. However, expectimax does not allow pruning and its long running time makes it infeasible to test given a limited amount of time. Minimax proved to be our next best option as it allowed pruning. In addition, in minimax even if the opponent does not minimize the payoff of the player, the player would never do worse than the case where his opponent minimizes his payoff.

3.3 Ways to increase playing performance

The main issue we have with in-game decision making is due to run time. The time taken for minimax and expectimax is significantly longer as compared to local search algorithm. Even with the pruned minimax algorithm, the time taken to play a move is still only about 100 lines per second and takes a few hours just to play a single game.

To improve the runtime of our playing algorithm, one solution would be to use multithreading. Each thread can be used to determine the heuristics of each possible move. Although this will allow multiple heuristics to be calculated simultaneously, pruning of the minimax will no longer work which may result in an increase in run time.

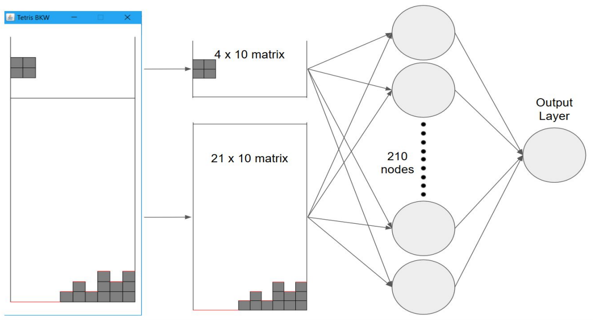
3.4 Why we chose the genetic algorithm

Other than the genetic algorithm, we tried to implement Deep Reinforcement Learning to train our agent to play Tetris, similar to [2]. More details about deep reinforcement learning will be provided in the next section.

We chose to use our genetic learning algorithm instead of Deep Reinforcement Learning due to a number of reasons. Firstly, the genetic algorithm performed much better than this learning algorithm in terms of the performance measure. Secondly, the time taken to train a decent set of weights for a neural network is much slower, as there are much more weights in a neural network than the 7 heuristics for genetic. It has also been shown by [3] that an agent using a heuristic function hand-crafted to the Tetris game consistently performs better than an artificial neural network because of the neural network's generalization. That is, playing the game of Tetris only based on visual input and not being 'guided' by any heuristics. Lastly, we are unable to implement a multithreading algorithm that can work with Deep Reinforcement Learning, thus there will be no speed up as compared to our genetic algorithm.

4. Experimental Methods

For Deep Reinforcement Learning, we used 3 layers of neurons. The input layer, one hidden layer and the output layer. The input layer consist of 250 nodes. The first 210 nodes corresponds to the current state of the 21 by 10 Tetris board. The next 40 nodes corresponds to a 4 by 10 matrix which would be the orientation of the and position of the next piece. All of the nodes in the input layer are connected to all the nodes in the hidden layer. The hidden layer consists of 210 nodes that connects all the inputs together. The output layer will take in input from all the nodes within the hidden layer. The value at the output layer will then go through a sigmoid function to get a value from 0 to 1.



The final output from the output layer will contain the probability of selecting the move given the current state. We will choose to make the selected move if the probability of making the move is larger than 50%. If the probability is less than 50%, we will swap the current move with another move until we attain a move that has a larger than 50% output. Suppose after iterating through all the moves and none is selected, the move with the highest value will be chosen instead. The weights are trained using a payoff function based on the number of lines cleared.

We faced many problems while trying to randomly initialise the weights as the number of lines cleared for a set of randomly generated weights will always be 0 based on the original payoff function. The weights also took significantly longer to learn and clear even a single line.

5. Conclusion

In conclusion, we have found that a utility-based Tetris agent can become significantly better than an average human at playing Tetris. Although the rules of the Tetris game in our project have been simplified (no scoring, combos, T-spins, etc), we believe that with some adjustments in our heuristic features, our agent should be able to perform similarly well in the full Tetris game.

6. References

[1] Shahar, E., West, R. (2010). Evolutionary AI for Tetris.

[2] Stevens, M., & Pradhan, S. (2017). Playing Tetris with Deep Reinforcement Learning.

[3] Ian J. Lewis & Sebastian L. Beswick, (2015) Generalisation over Details: The Unsuitability of Supervised Backpropagation Networks for Tetris.