Genetic Tetris

Thomas Young

December 2018

Introduction

A topic I have been very interested in learning, which we only touched upon in class, is genetic algorithms. I have made attempts at implementing one, though they did not go as well as I had hoped. Because of this, I wanted to find a project that would solidify my basic understanding of them and expand on it. Since most of the class focused on applying what we learned, such as Markov decision processes and q-learning, to Pacman, I thought it would be cool to apply a genetic algorithm on another older game, Tetris.

My goal for the practicum is to create a Tetris AI which can display the ability to make educated moves based on previous experiences. I define an educated move to be one that could increase the score of the game and create an optimal setup for the next piece.

The following guidelines were used for this version of Tetris[1]. Most of these rules came from the official Tetris guidelines. [1]:

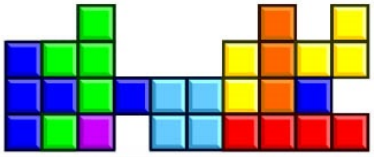
- The board is a 10x22 grid of squares
- There are the 7 normal tetrominos[2]: I, O, T, L, J, S, Z
- Each piece starts in the middle of the board at the top laying as flat as possible
- The tetrominos are picked randomly (however not using the 7 system random generator which the official Tetris uses)
- There are no look ahead pieces
- The score increases by one every time a tetromino is created and by 10 for every row cleared

Methods

My initial search for implementing a genetic algorithm for Tetris was very fruitful. I found multiple different implementations, and although there was some diversity across them, there was one thing that remained the same: The genes were used to represent weights for features of the board. With these weights, the algorithm determined the optimal next move for the tetromino. These are the features I chose to use[3][4]:

[1]Original Tetris code came from Laria Silvasur's implementation[2]. I edited her code to allow my AI to play the game.

[2]tetrominos are the shapes composed of four squares used in Tetris.



max height: 4
cumulative height: 30
relative height: 2
roughness: 7
number of holes: 2
cleared rows: 0

- Max Column Height: The height of the tallest column
- Cumulative Height: The sum of all the column heights
- Relative Height: The difference between the tallest and shortest column heights
- Roughness: The sum of the difference of heights between neighboring columns
- Holes: The number of empty squares which have a block somewhere above them.
- Rows Cleared: The number of rows that will be cleared in the current board layout

I chose these features because they represent three main strategies for Tetris which promote an increase in score, a decrease of the structure height, and thinking ahead.

1. **Clear rows** - You want to encourage the player to clear rows. You can achieve this by adding a feature for row clearing. This way the AI will be able to learn that this is beneficial
2. **Keep the structure low** - This is a pretty intuitive idea. As long as you can stop the blocks from touching the top, you will stay alive. To handle this, you can use three features: max column height, cumulative column height, and the number of holes in the board. The max column height ensures that the player doesn't stack all of the pieces in one place, resulting in them losing very quickly. Cumulative column height incentivizes the player to make moves that increase the overall height as minimally as possible. Finally, preventing holes from appearing will help the player in the long run since any row with a hole will be much more difficult to clear.
3. **Think ahead** - A player being able to look ahead and create a surface which encourages building is critical to getting a high score. To quantify the looking ahead process, the features for roughness and relative height are used. It is beneficial for a player to keep the board as flat as possible, as this sets them up to play any piece with as little drawbacks as possible. Both of these features encourage the player to keep the board in as flat as possible.

With these features to chose from, we can handpick the ones we think work best and create a tuple of weights. Each weight is a float between -1 and 1 representing how positive or negative that feature is. For example, if we were to make a gene containing the max column Height, relative Height, holes, and rows cleared with their weights we would get the following

Max column height weight = 0.1	Relative height weight = -0.254
Holes weight = 0.9	Rows cleared weight = -0.998

$$(0.1, -0.254, 0.9, -0.998)$$

The example above weighs a tall column and a board with holes positively while weighing relative height and the number of rows cleared negatively. With a set of weights, you can then apply them to the game. When a new tetromino is created, the AI tries every possible move. The score of that move is then determined by summing the product of the weights and their corresponding feature value. The move with the most optimal resulting board is chosen. This continues until the AI loses.

With this explanation of how the genes will work for the Tetris AI, the five steps for how the genetic algorithm works can be explained.[5]:

- Initial Population
- Fitness Function
- Selection
- Crossover
- Mutation

Initial Population The point of the initial population is to create a set of random genes which are possible solutions to Tetris. The two articles I read differed on how many genes to instantiate in the initial population. One of them only used 50[3]genes to start while the other had 1000. Having only 50 in any generation, let alone the initial one can often lead to a lack of diversity in the genes making it more likely to find a local minimum. Since we want to diversify our genes as much as possible, we are incentivized to have a larger gene pool. Additionally, every generation after the initial will have less variation among the genes making it more difficult to escape local maximum after the initial population. To encourage the genes to start on the right path, I set my initial population size equal to ten times the normal generation size. for example, if every generation normally has a population of 10, the initial population will be of size 100.

Fitness Function The fitness function is fairly simple. When a gene is used to play a game, it does so until it loses. Once this happens, the gene's fitness is set to its score in Tetris. This value is saved along with the gene in a dictionary for the selection step.

Selection Once every gene has played and scored a fitness value, selection occurs. This step involves taking the best genes from the current population and using them to create the next generation. The simplest way of doing this, which is what I chose to do, is to sort the genes by their fitness score and take a certain top percentile of them to pass on their feature weights. I chose the top ten percent. This ensures that the next generation will only be made of the best, guaranteeing an overall increase in average fitness per generation.

The other version of selection I saw was tournament selection[4]. For this, you sample 10% of the population and take the top 2 genes for breeding. You continue to make these samples until you have 30% of the next generation selected.

Once selection is finished, crossover and mutation are applied.

Crossover This step most resembles heredity in genetics. I again went for a more standard approach where the two genes swap random feature weights between them. The result is two child genes for the next generations. Crossover happens to all the genes which were selected in the last step. They are crossed over with their neighboring gene from the new generation.

The other method Lee uses is called weighted average crossover[4]. This crossover creates a new gene by taking the weighted average of gene1 (g_1) and gene2 (g_2) based on their fitness (f()),

$$g_{new\ gene} = \frac{f(g_1)}{f(g_1) + f(g_2)} \times g_1 + \frac{f(g_2)}{f(g_1) + f(g_2)} \times g_2$$

This type of crossover get's an average between the genes while favoring the better gene slightly more.

Mutation The method I used for mutation was the same as the other genetic algorithms. Every gene has a 5% chance of mutation. If a gene is mutated, a random value between -0.05 and 0.05 is added to one or more feature weight. This step is used to prevent a loss of diversity in the gene pool.

Once selection, crossover, and mutation occur, the next generation plays Tetris, starting the process over again. It continues this until the generation scores converge.

Analysis

For my analysis, I will be looking into how the different genes and the number of genes affect the evolutionary algorithm. I ran eight trials, each of them running for seven generations. The first four trials used all six features to approximate how good a move was. The last four had the max column height and relative height features removed reducing the number of genes down to four features: cumulative height, roughness, number of holes, and rows cleared. This is meant to test the effects on the evolutionary process of shortening a gene and removing possibly unneeded features.

In the first four trials using all six features, max column height was consistently positively weighted. This could be helping, however, it is counter-intuitive to reward the player for building the structure to be tall. Because of this, the max column height feature was removed. The relative height column was also removed simply to reduce the overall number of features the gene had.

From these trials, I am able to measure the effect of the different genes using the average Tetris score in each generation and how much this changes.

These are the results I got from the above trials:

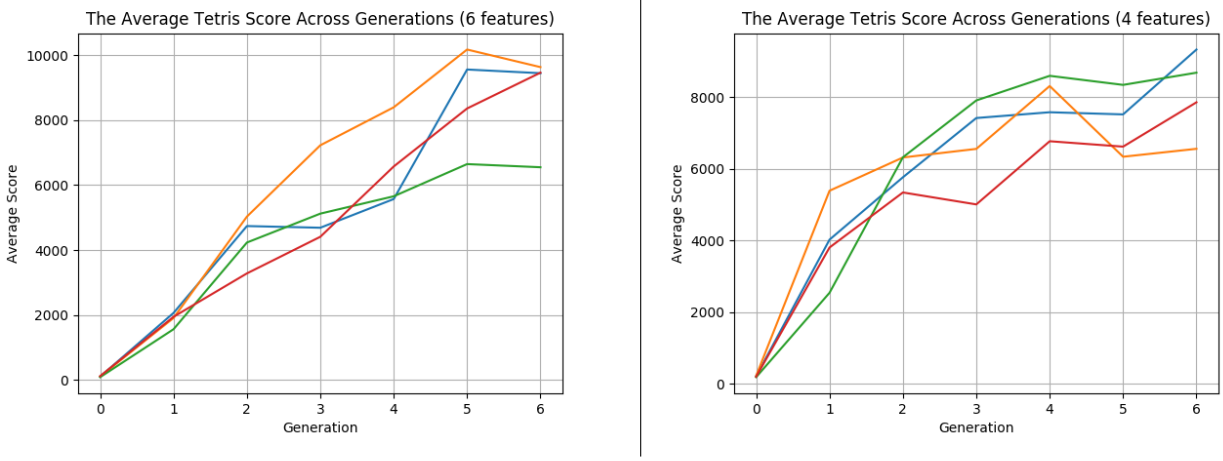


Figure 1

The tables above display the average score in Tetris a trial gets over its generations. The genes which had all six features grew at a relatively constant rate with a bit noise. The genes with only four features followed a logarithmic trend. Starting off with a quick increase in average score and plateauing at around 8000. From this, there are two things to take note of.

The genes with only four features had much more success with initial training. This could have been caused by either of the adjustments performed. Since we removed the max column height weight, this could have encouraged the genes to dislike taller structures, indicating it was causing issues in the six feature trials. Additionally, the increase in growth could have been a result of having fewer features in each gene. With less diversity, there will be fewer unique gene combinations to pick from, increasing the likely hood of obtaining fit genes in the initial population.

Although the genes with less features seem to do better in the start, they begin to converge early on.

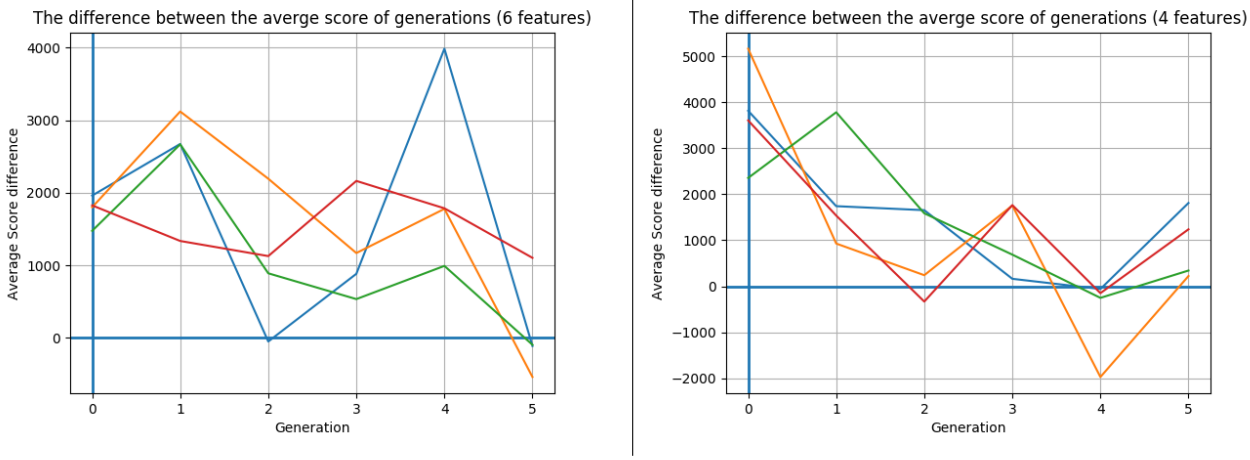


Figure 2. The Graphs above show how much the average scores differ from one generation to the next

Figure 1 shows the genes with four features converging around a score of 8000. Compare this to the six feature genes which continue on past that point with no signs of slowing down. Taking a closer look, Figure 2 displays the change in score for four feature genes begin to converge to 0. Again comparing this to the graph for six feature genes which is just noise with a slight negative trend. Although removing the max column height feature from the gene could have influenced early convergence, it is more likely that having fewer features is was caused it. Something to keep in mind when creating a genetic algorithm is to make sure each generation has diversity. Otherwise, the algorithm will converge too early at a local maximum. Since the four feature genes often having less population diversity, the early on convergence can be attributed to this. In this case, the six feature genes are advantageous if you want to get better more accurate results given a bit more time.

Conclusion

From the analysis, two properties can be observed from removing unnecessary features and shortening genes. First, when shortening a gene or removing a less helpful feature, genes are able to improve faster. Second, the more features included in a gene, the more generations it will be able to train for without convergence. With these two properties, the genes can be modified to fit the needs of an individual. For a project where time is a valuable resource, limiting the number of genes used would be preferable. This will evolve the genes faster with alright results early on. However, if time is less of a concern, Having more genes would be preferred. With more features the AI will be able to train for longer, allowing for better genes in the long run.

My goal for the Tetris AI was for it to be able to make educated moves by scoring well and setting itself up for success with the following pieces. Over generations of training, the AI was able to find a handful of genes that got it to score upwards of 40,000 points. To put this in perspective, my record is about 1000 in the same version of Tetris. Although I did

succeed in my basic goal, there are a few improvements that I would like to implement in the future.

The first improvement would be to focus on how long it takes to train the AI. It was able to get a relatively good score of 1000 - 3000 with about 20 minutes of training. However, in order for the genes to score above 20 thousand, it had to train for about 12 hours. There are a few things I can do to speed this up. First, is to get rid of anything that the AI does not need. This mostly refers o the GUI and simplifying the process of placing tetrominos. Another option is to parallelize the learning process by running the simulations in different process nodes. At the end of each generation, all the nodes compile the gens and perform selection on the joint set. This would speed up the learning by allowing simultaneous training with a larger selection of genes.

Besides speeding up training, there are a few other features that could improve the AI's ability to make optimal moves. If the version of Tetris being used allows the player to see the next piece, a look ahead algorithm could be used to make the best placement taking into account the next tetromino. If the next piece is unknown, it can take advantage of the 7 system random generator Tetris uses to generate the piece. The AI would then be able to perform expectimax with the probability certain pieces will come next. Finally, being able to use the Tetris bot with a real Tetris game as opposed to an unofficial version. This would involve having a screen reader to get data and faking keyboard input.

References

[1] "Tetris_Guideline". November 22, 2018. Accessed Devember 14, 2018 https://tetris.wiki/Tetris_Guideline

[2] Laria Silvasur. "tetris.py". May 20, 2017 Accessed December 13, 2018. https://gist.github.com/silvasur/565419

[3] Muhammad Z. Mousa. "tetris-ai". October 19, 2017. Accessed December 13, 2018. https://github.com/mzmousa/tetris-ai

[4] Yiyuan Lee. "Tetris AI - The (Near) Perfect Bot". April 14, 2013. Accessed December 13, 2018. https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/

[5] Vijini Mallawaarachchi. "Introduction to Genetic Algorithms". July 7, 2017 Accessed December 13, 2018. https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3