Neuroevolution and Genetic Algorithms with Tetris

Philip Akkerman
dept. of Computer Science
Brock University
St Catharines, Canada
pa13xb@brocku.ca

David Hasler
dept. of Computer Science
Brock University
St Catharines, Canada
dh15pd@brocku.ca

***Abstract*—The game of Tetris is used as a video game test bed for neuroevolution and genetic algorithm models. Both of these models are defined in this paper and the approach used in building them is explained. Experiments were run to analyze how well each of the models learn play Tetris using various training parameters. The results found that the neuroevolution model performs sub-par and is not able to successfully play Tetris, despite various alterations to the model's learning structure. However, a genetic algorithm model using a heuristics evaluation function and a search tree algorithm to determine moves accomplishes a result comparable to human play and successfully learns to play Tetris. This paper explores the reasons why the neuroevolution model failed and how it relates to the random nature of video games like Tetris.**

I. Introduction and Problem Definition

A. Introduction

In this research paper, we will look into the background and explain how neuroevolution and genetic algorithms are used in the world of artificial intelligence. Furthermore, effectiveness of using both neuroevolution and genetic algorithms to learn to play Tetris will be examined and compared along with studies that use similar models to play Tetris.

B. Problem Definition

In this day of age artificial intelligence is becoming more important. It can be used in classification problems, optimization problems and many other types of problems. In the case of video game development, creating a game controller using machine learning can save substantial amounts of time and money which would ordinarily be spent hard-coding game AI. When using artificial intelligence, there is a looming question of how efficient different types of AI are at solving different classes of problems. The research done in this paper shows the effectiveness of the results when applying two different machine learning models to the game of Tetris. The first was a neuroevolution model and the second was a genetic algorithm, and both were examined while learning to play Tetris.

II. Background

A. Tetris

Tetris is a classic video game made originally by Alexey Pajitnov in 1984. It is played on a grid of squares (a gameboard) with a width of 10 and a height of 20. Shapes called Tetriminos fall down the gameboard one square at a time, and it is up to the player to stack them with the goal of filling in

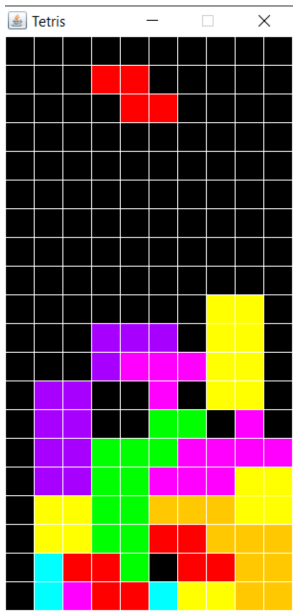

Figure 1. Tetris

complete horizontal lines (See figure 1). Each line filled grants a number of points, and higher number of lines cleared grant higher amounts of points. When a line is filled, it is deleted, bringing all above lines down one space. The game ends when the gameboard has been filled to the top and a new Tetrimino cannot be created without causing a collision. Clearing lines quickly allows the game to continue and the score to increase.

There are seven unique Tetrimino shapes, as shown in figure 2. The game starts with one of the seven being randomly generated at the top of the gameboard. The player can rotate the Tetrimino and move it left, right, or down. Once it lands on the bottom of the gameboard or collides with another Tetrimino, it is locked in place and a new Tetrimino is generated for the player to control.

Because Tetris has so much randomness built into the game, it is a challenging optimization problem to solve using AI. Regardless of how well a player controls the pieces, there are some sequences of Tetriminos which will always lead to a game over state [1]. Results like these can make it challenging for AI algorithms to learn, since an improvement over the current game may not lead to improvements in future games.

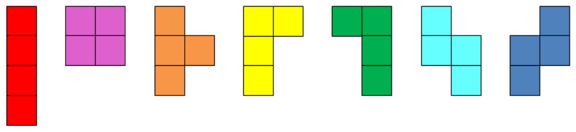

Figure 2. Tetriminos

a) Differences in Game Implementation: The falling speed of the Tetrimino typically increases over time, making it more difficult for a human player to plan its placement. Since this implementation of Tetris was to be used mostly with an AI which can calculate moves nearly instantaneously, the speed increasing feature was not implemented. In many implementations, the next Tetrimino is also made known to the player. This feature was also not implemented, partially to save on compute time when using search tree algorithms. The only other difference in our implementation of the game is that if a rotation of a Tetrimino would result in colliding with the edges of the gameboard, then the rotation is not allowed. In most official versions of the game, the Tetrimino is pushed towards the inside of the gameboard instead to make room for the rotation. Our implementation uses the official point scoring scheme where one line cleared grants 40 points, two lines grant 100 points, three lines grant 300 points, and four lines (known as a Tetris) grants 1200 points.

B. Neuroevolution

Neuroevolution is the combination of a neural network and genetic algorithm. The learning model used in this project was a simple feed-forward network which used an evolutionary algorithm to update the weights and biases. This neural network used an architecture of 231, 160, 120, and 40 nodes per layer, where the input layer was 231 and the output layer was 40. In the input layer it was decided that the inputs should be the rotations of a single Tetrimino, the type of Tetrimino, and the input of the gameboard, represented as an array of ones and zeros. For the last layer it was used differently dependent on the scenario. One way was to let the AI decide out of four options, which were left, right, rotate or down. These controls would be the same as a human player's controls. Another variation of the outputs had 40 options, which would determine what column the Tetrimino would fall and what rotation the Tetrimino would have, meaning outputs equal to 10x4. The layers in between changed depending on the number of outputs, so for an output layer of 40, the hidden layers were 160 and 120, but for the output layer of 4, the hidden layers used were 150 and 30. Once the network was built the input was fed forward and the next layer was found by getting the sum of the weights multiplied by the previous layer's activation values. Lastly the bias was added to the sum and then the total value was put through a sigmoid activation function. This feed-forward continued for each neuron in each following layer until the output neuron was reached. After building the neural network for the neuroevolution model, the evolutionary algorithm was added to make the model learn.

The evolutionary algorithm first finds the neural network in the population which returns the best evaluation. This network becomes the parent network for the next generation of networks. Then the parent neural network is mutated a number of time where each mutation applies a random number between -1 and 1 to a random layer node connection in the network. The mutated networks then go on to the next generation of neural networks along with the parent neural network and a few randomly generated networks. This parent network is kept in the next generation to keep the current best network in reproduction, and the new random networks are generated to provide new options for exploration when optimizing the networks' weights and biases.

A few variations of neuroevolution were implemented while trying to find the best implementation for playing Tetris. One variation used allowed the to AI play a number of games for each network in the population, then whichever network had the best score was kept. Another variation changed the measure from the average points scored by clearing Tetriminos, to the average time the network survives. A few more methods implemented were allowing the AI to play the game with control arrows versus playing with specific move placements. Using the control arrows was imitating human-like controls because they could move left, right, rotate, or down, just as a human controlled player would. In the specific move rendition the AI player could place Tetriminos using the Tetrimino rotation and column placement. A last big implementation change was using a fitness function to allow a form of supervised learning instead of unsupervised learning. In the supervised learning variant the learning algorithm would evaluate based on the move it made compared to the optimized fitness evaluation function given by Bohm et al[2].

C. Genetic Algorithms

Genetic algorithms are versatile optimization functions. They are inspired by the theory of evolution, in that they mutate values slightly over time, using survival of the fittest to find the best solution to a problem. In our case, a genetic algorithm was used to optimize a Tetris gameboard evaluation function. A basic version of this evaluation function was originally used to provide supervised learning to the neuroevolution network, but its performance on its own was so superior to the neural network's that we eventually focused on optimizing the evaluation function instead.

The evaluation function measures various attributes of a Tetris gameboard. Using a search tree algorithm, each possible move that can be made with the current Tetrimino can be evaluated, and the optimal move can be found. There are a maximum of 40 different moves available with any Tetrimino, 40 being a combination of 10 columns and 4 possible rotations. The attributes we used in the evaluation function were first developed and laid out in research done in a 2005 study: An Evolutionary Approach to Tetris [2]. The twelve attributes used are as follows:

1) Pile Height: The row of the highest occupied cell in the board.
2) Holes: The number of all unoccupied cells that have at least one occupied above them.
3) Connected Holes: Same as Holes above, however vertically connected unoccupied cells only count as one hole.
4) Removed Lines: The number of lines that were cleared in the last step to get to the current board.
5) Altitude Difference: The difference between the highest occupied and lowest free cell that are directly reachable from the top.
6) Maximum Well Depth: The depth of the deepest well (with a width of one) on the board.
7) Sum of all Wells: Sum of all wells on the board
8) Landing Height: The height at which the last tetrimino has been placed.
9) Blocks: Number of occupied cells on the board.
10) Weighted Blocks: Same as Blocks above, but blocks in row n count n-times as much as blocks in row 1 (counting from bottom to top).
11) Row Transitions: Sum of all horizontal occupied/unoccupied-transitions on the board. The outside to the left and right counts as occupied.
12) Column Transitions: As Row Transitions above, but counts vertical transitions.The outside below the gameboard is considered occupied.

Once each attribute is calculated from the gameboard, they must be assigned weights to determine how much each attribute contributes to the gameboard evaluation (a higher evaluation indicates a worse position). These weights are the values that the genetic algorithm optimizes. The resulting evaluation of a move is therefore the weighted sum of weights times attributes.

The genetic algorithm is initialized with a random set of weights, which are mutated a preset number of times to form a new population of weights. Each member of the population (a set of weights) is used to play a preset number of Tetris games, and the average score of the games is used to rate that member. Whichever member obtains the highest rating is used to ’reproduce’, or to make the next generation of weights through mutation. This cycle of mutation and evaluation is repeated for a preset number of epochs.

The mutations used in our implementation were randomly chosen from six different mutations whenever the mutation function was called. The possible mutations were as follows:

1) A random value between -1 and 1
2) A random value between -2 and 2
3) The original weight times a random value between 0 and 1
4) The original weight divided by a random value between 0 and 1
5) The original weight times 2, divided by a random value between 0 and 1
6) The original weight less a random value between 0 and 1

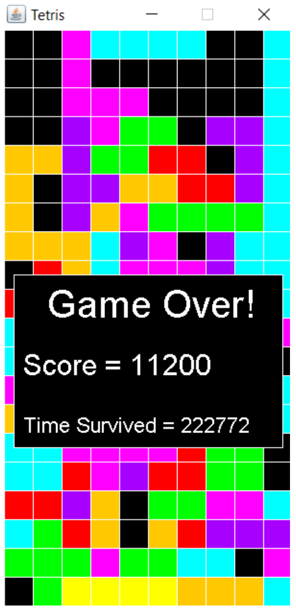

Figure 3. A Tetris Game Played with the Optimal Weights Using Genetic Algorithms

Genetic algorithms have the potential for many different options, from mutations and crossovers to the way that the parent of the next generation is selected. We implemented some of these extra options in our genetic algorithm. Firstly, a non-mutated copy of the parent (the highest rated member which is chosen for reproduction) was included in the next generation. Secondly, the three highest members (known as elites) were also included in the next generation. Since Tetris is a highly random game, we thought it wise to give the highest rated member two chances to remain in the next generation to prevent loss of quality. Lastly, to avoid the issue of the algorithm getting stuck in a local maximum, a preset number of new random members were added into each new generation. This mimics adding bio-diversity in the evolutionary analogy.

III. Experiment Setup

A. Neuroevolution

To begin, the neuroevolution model was trained and examined during 16 experiments. The desired result for these experiments was to explore neuroevolution parameters and the effects which it had on the AI learning to play Tetris. This model had many different designs or goals as mentioned in the above background section on neuroevolution, and the parameters are shown in the below list.

1) Learning model: Supervised/Unsupervised
2) Max epochs per experiment: 40
3) Moves per epoch for supervised: 30, 65, 100
4) Games per epoch for unsupervised: 30, 65, 100
5) Number of new networks for epoch: 10, 20, 30
6) Number of mutations for network: 30, 165, 300
7) Number of random networks per mutation: 1, 3, 5
8) Score evaluation: Points/Time

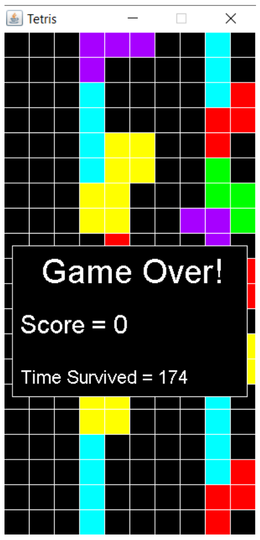

Figure 4. AI Attempting Longer Survival Using Neuroevolution

9) AI control: Controls using keys/Specified moves

In the third and fourth points is moves per epoch and games per epoch; each is for a different learning method. The reason behind this is that the moves per epoch ends the game from training after so many moves, where as the games per epoch for unsupervised will finish after training the number of games. This is because the unsupervised model got feedback from the game score after every game, and the supervised model got feedback after every move from the fitness function. The fifth point in the above list declares how many networks will be created for each epoch in beginning. If there are 10 new networks, then the 10 networks will compete to reproduce for the next generation which is where the number of mutations is key. Number of mutations declares how many mutations will happen to the network that is reproducing. Score evaluation was implemented to toggle between evaluating how well the network was doing based on the Tetris points score or how long the network stayed alive. Lastly, the AI control parameter allowed the AI to play using human-like controls (arrow keys), or specified moves via column and rotation to place the Tetrimino.

The testing was conducted by using 40 epochs per changed parameter and the average score and parameters were recorded.

B. Genetic Algorithms

Aside from initial tests, twelve experiments were run with different genetic algorithm parameter setups. The goal of these experiments was to find the optimal set of weights, such that the evaluation function would be able to play the game of Tetris to a level equal or greater than human players. The parameters that were modified for training were the number of games played per child per epoch, number of children, number of mutations per child, and number of additional random children, where a child refers to a set of weights in the population. Experiments were capped at 40 epochs each due to time constraints. This cap seemed acceptable since the algorithms generally stopped increasing in score by 25 epochs during initial testing.

Due to the random nature of genetic algorithms, which was compounded by the random nature of the Tetris game, we were unable to make any conclusions about cause and effect relationships between parameter setups and outcomes. However, some experiments did yield weights that generated higher scores than others.

Throughout testing, we recorded the average game score of the highest performing weight setup per epoch, as well as the highest score achieved in any one game, called the high score. Although this high score is not necessarily indicative of future performance, it was still valuable to record this to see what the evaluation function was capable of. By taking an average of at least 30 games, we were able to get more accurate feedback about which set of weights was actually better at playing Tetris. The final setup of weights was recorded for each experiment after all training epochs were finished.

Since there were only 12 weights to mutate, the number of mutations per child was generally kept low at one per child. For the sake of computation time, only one new random child was added each generation by default, on top of the ten children which were mutated from the parent, the parent that was kept to the next generation, and the three elite weight setups, for a total of 15 members in a population by default.

IV. Results and Discussion

A. Neuroevolution

After completing the experiments, the graphs of the neuroevolution model results are shown in figure 5 and figure 6. The results of figure 5 have a very similar trend that increases slowly, but do not show that this neuroevolution model is learning how to play Tetris well. Another observation of the graph is that when the AI used the control arrows it began with a much lower performance, but gradually learned to play almost as well as specified movement. From the graph in figure 5 when the neuroevolution model used supervised learning with number of moves per epoch 30, number new networks 10, number of mutations 30, and number of random networks 3; these parameters ended with the best time evaluation. The rest of the parameters tested yielded results which were not unique and followed the general trend of figure 5.

Next was the results of changing the evaluation method to using the score, fewer experiments were done on the score because it yielded results that were sub-optimal when compared to the genetic algorithm playing Tetris. Appearing in figure 6 is the results for the neuroevolution learning model using score as the evaluation. From these results immediately the parameters that used a score goal are shown performing worse than all other parameters. The other parameters used an error goal which was found to provide better results as shown in figure 6. Also seen in the results graph is that when varying the parameters they all begin at around the same score, which then lowers over time with small jumps up and down. By the

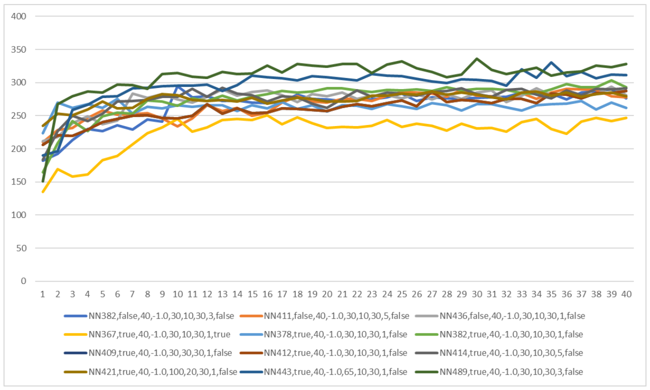

Figure 5. Neuroevolution Time Graph

end of the graph the parameter using 300 mutations slightly outperforms the other two competing error goal experiments.

Clearly the results found from the experiments show that the neuroevolution model performed sub-optimally, especially in comparison with other models in the article *The Game of Tetris in Machine Learning*, because this neuroevolution model is only able to reach 50 or less score from playing Tetris[1]. The model wants to perform well and does learn certain patterns; for example in figure 4 it shows the AI playing Tetris that had the goal of surviving longer and it appeared to attempt to survive longer, but it did not realize that it must clear lines in Tetris to survive longer.

Theoretically the neuroevolution model may perform better with a better mutation function as in the genetic algorithms model which does not just use a random multiplier between -1 and 1. There also may be further exploration required with the network architecture or how the learning is done with the neural network. Another solution is to attempt different learning techniques to find different results, such as building a neural network model which uses back-propagation and examining if the network does better than the neuroevolution model. If it does better, then the neuroevolution model should be rebuilt because there may indeed be a configuration issue.

Another notion is that the game of Tetris is random therefore making it more difficult for the neuroevolution model to reflect changes made when learning. For example if a network performs very well in one game, the next game it could try to make the exact same moves and get a much worse score. To further investigate neuroevolution, the game tested could be a more consistent game which could provide better results.

B. Genetic Algorithms

The genetic algorithm method performed far better than the neuroevolution model. Experiments with the genetic algorithm experiments provided a wide range of results. On the top end of the results was a high score of 57,300 points, and on the low end was a high score of 11,080. Using the average of the last 20 epochs to exclude the initial training phase, the best scoring experiment had an average score of 7,965 points, and the lowest had 2,860.

Unlike with neural networks, these results were not dependent on the set of parameters used to train the network. As a control, the default parameter set was used in four of the 12 experiments. This default parameter set trained weight setups that achieved the two worst results, as well as the third and fifth best results. This wide gap in results with the same parameters demonstrates the random nature of genetic algorithms, and how time and luck play an important factor in finding optimal solutions using these algorithms. There likely exist many weight setups that would perform even better than the ones discovered in these experiments, and it would just take more time to discover them.

With the randomness of the Tetris however, one parameter that we expected to make a difference was the number of games per epoch. Since the same set of weights can play games with both scores of 40,000 and 1700 depending on the order of Tetris pieces it receives, as demonstrated in further tests, we hypothesized that playing more games per epoch would generate a more stable evaluation of the weight setups, allowing for continual improvement and reducing the risk of selecting a sub-optimal weight setup for reproduction. The experiment that performed the best, Test 1 in figure 7, was one where the number of games per epoch was increased from 30 to 65. Despite this however, it is clear that from epoch 22 to 23, the superior weight setup was not selected due to having multiple bad games in that epoch. The result was that the best weight setup was discarded, leading to the downward trend in the graph in subsequent epochs. Unfortunately, only the final weight setup of each experiment was recorded, so we could not test if the setup achieved at the peak would perform at that level consistently.

After the experiments were performed and the optimal weight setup was found, we had the computer play 10,000 games with that weight setup to see how it performed on average with a larger sample size. The highest score achieved was 45,020 and the average score was 7078. The average score of Test 1 at the last epoch was 7408, which was relatively close to the 7078 achieved in the large test. The fact that the high score was 12,280 points lower than the highest score achieved among the experiments further indicates that there was a

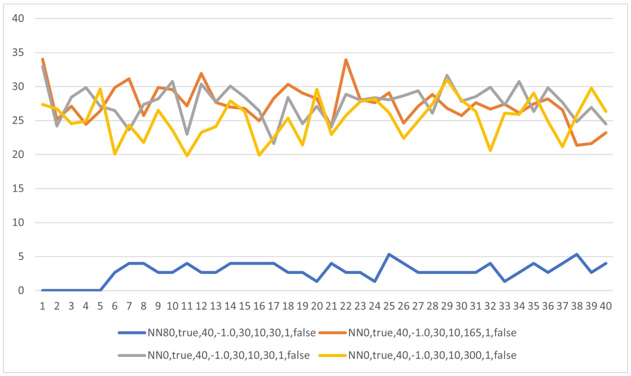

Figure 6. Neuroevolution Score Graph

weight setup achieved in earlier epochs that had the potential to outperform the final setup found. This further demonstrates the challenge of working with a continually changing testing set such as real Tetris games.

A way to avoid this might be to create a testing set of a large number, say 10,000, Tetris board positions, and to train the genetic algorithms or neural networks on this data set. Since the data set would not change, training could be more predictable, and a better result could be achieved. This method runs the risk of over-fitting, or memorization of solutions, but the benefits of consistent improvement could outweigh those risks. Tetris is estimated to have 7×2^{200} states [1], so even a very large testing set would be a minute subset of the whole Tetris game.

Table I
FINAL WEIGHT SETUPS IN TEST 1 AND 2 (ROUNDED TO 2 DECIMAL POINTS)

Weights	**Test 1**	**Test 2**
Holes	-0.46	-0.49
Blocks	-0.02	0.00
Weighted Blocks	0.80	18.76
Row Transitions	0.56	2.41
Removed Lines	-1.24	-70.43
Connected Holes	0.58	1.49
Column Transitions	0.46	4.73
Altitude Difference	-1.63	-34.95
Maximum Well Depth	0.33	11.81
Sum Well Depths	0.24	0.04
Pile Height	0.73	-16.15
Landing Height	2.44	-0.09

Another interesting result from the experiments was how different the final weight setups were between the first and second best performing experiments (see table IV-B. Test 1 had a positive weight of 0.73 for Pile Height, where Test 2 had a negative weight of -16.15. The differences in Removed Lines, Altitude Difference, and Weighted Blocks were even greater. A natural assumption is that there exists an optimal set of weights, and different experiments would converge toward the same setup. These results disprove that assumption however, since vastly different weight setups can have similar results. When doing a test of 10,000 runs to find the average of Test 2's weight setup compared to Test 1's, Test 2 had a high score of 40,600 (45,020 for Test 1) and an average score of 6,405 (7,078 for Test 1). So although the scores for Test 2 were lower, they were still relatively high when compared to other experiments, despite how different the weight setups were from the optimal setup found in Test 1.

V. CONCLUSIONS

The most prominent conclusion that we found from this project was that neural networks are not always the most appropriate solution for an automation problem. In the case of Tetris, it was much more efficient to develop a heuristic evaluation function and use another form of machine learning, genetic algorithms, to refine that evaluation model and solve the problem of creating an AI controller to play Tetris.

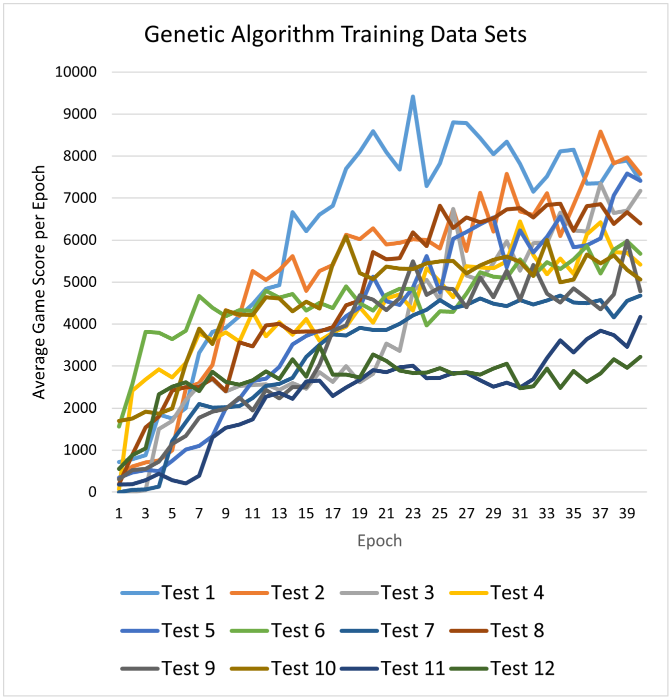

Figure 7. GA Data Graph

While it is possible that with more time and more variations to the neuroevolution model, we could have eventually solved the problem, the initial results of the neuroevolution network compared to the evaluation function showed that this effort was not the right direction. Even after modifying our neuroevolution model from using arrow key controls to specific moves, then from using an unsupervised approach to a supervised approach, the results were still sub-par. Using genetic algorithms to refine an evaluation function to decide moves provided results quickly and effectively.

Trying more consistent game patterns using a pre-set training set of gameboard positions would be adequate further exploration into the discipline of neuroevolution with Tetris. In the future, more complex games could be tested using the neuroevolution and genetic algorithm models. With more complex games, such as real time strategy games, a simple heuristic evaluation function which can effectively choose actions like the one used in Tetris might be more difficult or even impossible to create. In these cases, neuroevolution networks would be a more appropriate choice.

REFERENCES

[1] S. Algorta and Ö. Simsek, "The game of tetris in machine learning," *CoRR*, vol. abs/1905.01652, 2019. arXiv: 1905.01652. [Online]. Available: http://arxiv.org/abs/1905.01652.

[2] N. Böhm, G. Kókai, and S. Mandl, "An evolutionary approach to tetris," in *The Sixth Metaheuristics International Conference (MIC 2005)*, Vienna, Austria, Aug. 2005. [Online]. Available: http://www2.informatik.uni-erlangen.de/publication/download/mic.pdf.