Playing Tetris Using NEAT, Comparing Raw Input and Feature Extraction

Daniel Jansson Martin Larsson Erik Rimskog

October 29, 2023

Abstract

The rise of machine learning and AI is noticeable in its ability to consistently outperform humans at video games. A game that has been thoroughly subjected to different machine learning methods is Tetris, a classic arcade game developed in 1989. This paper explores the use of the genetic algorithm NEAT (NeuroEvolution of Augmenting Topologies) for unsupervised learning of playing Tetris. A successful agent emerged that would outperform most humans and NEAT was found to be suitable for this problem, however probably not more suitable than other methods.

Contents

1 Introduction

The goal of this project was to explore how well the machine learning method Genetic Algorithms[6], specifically NEAT (NeuroEvolution of Augmenting Topologies)[9], can train and evolve a network to play the game Tetris, i.e. creating an AI (Artificial Intelligence).

Multiple different approaches were tested, from naive solutions to solutions with different kinds of feature extractions. The different features were evaluated based on how well they helped the network to play Tetris.

It was observed that the networks produced by NEAT all had a similar or identical structure so they were compared to networks trained with PSO (Particle Swarm Optimization)[7] to see how the evolution of the network structures contributed to the ability of the network to play Tetris.

2 Background

2.1 Tetris

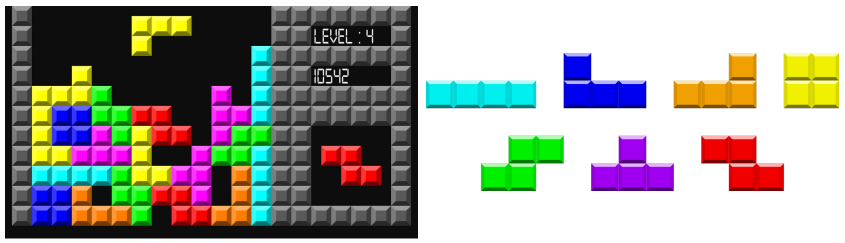

Figure 1: A Tetris board shown to the left[1] and the seven available pieces show to the right[2]

Tetris is a game first created by Alexey Pajitnov in 1989. The game consists of an initially empty grid (board) which is typically 20 rows by 10 columns big (as shown in figure 1). Each cell in the grid can be either filled or empty. One of seven different pieces (shown in figure 1), also called *tetrimino*, spawn at the top of the board and are then placed by the player. The goal of the game is to get as high score as possible by forming full horizontal lines across the board. When one of these lines are created, it disappears and gives the player some amount of score, this will be referred to as “clearing a line”. The game is played until a new piece can’t be spawned, which happens when it intersects with another already placed piece when it spawns.

The game has been shown to always eventually terminate, i.e. a game can not

[1]`https://farm4.staticflickr.com/3938/15538236401_a2eb533997_b.jpg`

[2]`https://upload.wikimedia.org/wikipedia/commons/thumb/3/39/Tetrominoes_IJLO_STZ_Worlds.svg/1280px-Tetrominoes_IJLO_STZ_Worlds.svg.png`

run forever[3]. There are sequences of pieces that make the game terminate even if placed perfectly by an perfect player.

Our implementation of the game gives the score of 100 for each formed line and 5 for each piece placed.

2.2 NEAT

NEAT (NeuroEvolution of Augmenting Topologies) is a genetic algorithm developed by Kenneth Stanley in 2002[9]. The method is centered around evolving the topology (i.e. structure) and weights of neural networks rather than just optimizing the weights. Considerable time and effort can be saved using this method since no design has to be put into the topology of the network before training[9]. Instead, NEAT will start with a simple topology and make it more complex over time. When the initial topologies in the population are sufficiently different they will separate into distinct species and continue evolving separately, ensuring that newly created topologies have time to mature and optimize and is not removed prematurely[8].

2.3 Particle Swarm Optimization

PSO (Particle Swarm Optimization) is an iterative optimization method developed by James Kennedy and Russel Eberhart [7]. The method uses a swarm of n particles to explore a n-dimensional search-space and iteratively proposes a solution. Each particle is a candidate solution and is associated with a position and a velocity. The movement of each particle is determined by its local best known position (the most optimal position the particle has visited) and the global best known position (the best position any particle in the swarm has visited). These two influences will move the swarm to the global minimum of the optimization function. Each particle is by some means interpreted as a solution, for example as a neural network of fixed structure by interpreting the position as weights.

3 Related Work

There are a plethora of examples of algorithms being used to play Tetris as Simón Algorta et al. summarizes in a paper[1]. One of which is a solution that Pierre Dellacherie[4] made which extracts features from the Tetris board and weighs them against each other. This approach is explained in more detail in section 4.4 and is the solution we had the most success with. The big difference is that Pierre hand-tuned all weights while we used machine learning to do it.

Jacob Schrum et al.[5] compared a raw representation (similar to section 4.1) to hand-designed features (similar to section 4.4) with NEAT and a variant of it called HyperNEAT to play Tetris. Their choice of features were different from ours, they had 21 features based on Bertsekas[2] while we had 6 other features based on Pierre[4].

4 Approaches

This section will explain the different approaches that was tried, what happened with each and what the reason for it working or not was.

4.1 Naive Approach

The initial use of NEAT to play Tetris was naive. Each cell on the board was used as an input to the network, meaning there was 200 input nodes where a 1 represented a filled cell and a 0 an empty one. This input space was large, around $2^{10 \cdot 20}$ different states, although many of which were invalid game states. There were six output nodes, representing six different actions: move left, move right, drop (move the piece all the way down), rotate left, rotate right and do nothing.

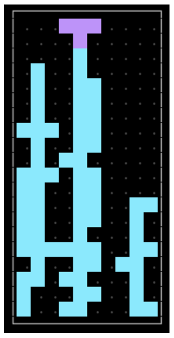
Figure 2

The game was played just like normal, no abstractions or simplifications. The flow was as follows:

1. Feed the network with the board state.
2. Perform the action whose output node has the greatest numerical value.
3. Move the piece one step down.
4. Remove rows and/or spawn a new piece.
5. Repeat from 1 until Game Over[1].

This approach did not produce satisfactory result. The network learned to survive as long as possible by filling the board with three distinct towers (see figure 2), it did not manage to clear any rows.

There are two big issues regarding this approach that makes learning more difficult, namely that the network has to find the current piece by itself and that there are too many steps before a piece lands. The current piece is almost impossible to determine by just looking at the board. The network doesn't know how all different pieces look like or that they can be rotated, it has to somehow learn that by itself, which is a waste because that information can easily be provided as input to the network. It is also difficult for other reasons, for example, if the current piece is right next to a filled square, where does the current piece start and stop? That is impossible to tell and is probably why the network could not manage to build rows.

[1]Popular term for stating that the game has finished

Having too many steps is also bad, because the network probably has a target location it wants to place the piece at right when it spawns and pushes it in the right direction. But it will be difficult to move it there because every step on the way also has to learn where the piece should land. It would be a lot better if the piece got placed in it is desired location immediately.

Another approach that does not have these problems is described in the following section.

4.2 Naive Instant Place Approach

To address the two problems from the previous section another approach was tested. Instead of playing the game with regular controls (move left, rotate right etc.) the game is played by dropping the piece on its desired location immediately. The network is given the current board state, the current piece to place and it outputs the location of where the piece should land and its orientation.

This approach uses a network with 207 inputs and 14 outputs. 200 of the inputs are the same as in the previous section, i.e. each tile of the board. The seven new inputs tells the network which of the seven different pieces is the next piece to place. Ten of the outputs specifies which of the ten different columns the piece should be placed at and the last four specifies which orientation it should land with.

Immediately dropping a piece is not as flexible as using the regular controls. It is for example not possible to place a piece in a hole in a wall by pressing either left or right faster than the piece falls. We believe that there is a good possibility of training a good network without this rather special move.

A problem that still existed with this approach was the large input space, which mean that the exact same situation was not likely to happen more than once, hence it became hard to train the network. The network also never learned to fill up rows, but only to fill up the board just as the previous approach (figure 2).

4.2.1 More Complex Fitness Function

To help the network understand how to eliminate rows were different fitness functions were derived. The fitness functions that were tried were to give more points the further down the piece lands, both linearly and exponentially, and give more points for each filled cell of the row that the piece lands on, also both linearly and exponentially. These fitness functions did not result in any improvements, the network still did not understand how to eliminates rows.

4.3 Column Heights Approach

One recurrent problem with the approaches mentioned in the previous sections was that the input space was to large. To address this issue the inputs to the network was redesigned, instead of taking each cell of the board as an input, only the heights of each column was given as input. The height of a column is defined as the distance between the bottom of the board and the cell closest to the top that is filled. This network has 17 inputs (10 for the columns and 7 for which piece) instead of previous 207 inputs.

This approach has the main drawback of losing a lot of information. The network basically only sees how jagged the top of the board is, it does not see how many holes there are nor how filled each row is. The performance of this approach was not good either and demonstrated the same behaviour as the previous approaches.

The input space can not really be reduced any further, so an overhaul of the general strategy was in order.

4.4 Board Features Approach

The main idea of this approach is that the network evaluate how good an action is rather than providing the action itself. This means that all possible actions should be considered by performing the action and then evaluating how good it was. The action with the highest score is the one actually executed. An action in this case is a position for a piece to be dropped at and a rotation for that piece. An outline of this new approach is as follows:

1. Take a rotation and position.
2. Drop the piece and remove any complete rows.
3. Extract *features* from the board and feed them to a neural network that will output a score.
4. Reset the board.
5. Repeat from 1 until all possible rotations and positions have been tested.
6. Drop the piece a final time at the position and rotation with the highest score.

A *feature* is a relevant characteristic used to evaluate a board state. These features were inspired by Pierre Dellacherie[1] and are defined as follows:

Rows: the number of eliminated rows the dropped piece eliminated, a number between zero and four.

Landing height: the height at which the dropped piece will land at after all rows have been eliminated, the height is calculated from the piece's top.

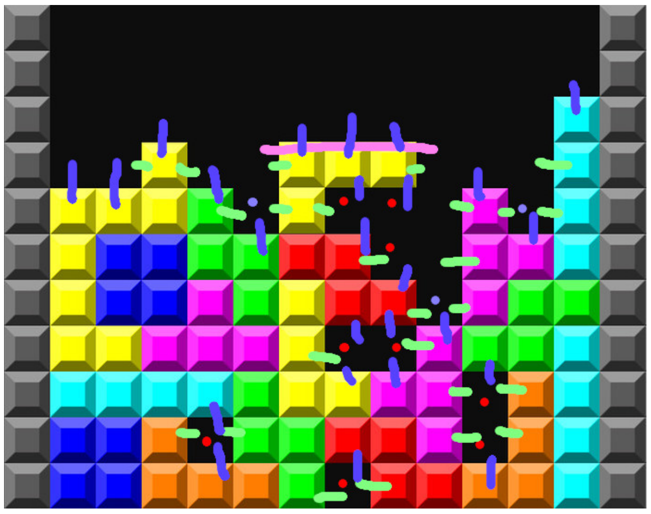

Figure 3: Annotated board state from dropping the yellow floating piece from figure 1 showing example of all features from section 4.4. The red dots are the holes, the blue dots are the wells, the green lines are row transitions, the blue lines are column transitions and the purple line marks the landing height. There were no lines eliminated.

Holes: the number of inaccessible cells, which is defined as the number of empty cells that has at least one filled cell above it.

Wells: the number of cells that has the cells immediately to the left and right of it filled and cells above it is not filled.

Column transitions: the amount of transitions within all columns, where a transition is a cell that has a neighbour of the other type (filled or not filled).

Row transitions: the amount of transitions within all rows, where a transition is defined the same as with column transitions.

The network is trained to evaluate all these features against each other and provides a score based on these features. The network has then six inputs and one output. This approach turned out to be the most successful of them all, managing to get scores in the millions, i.e. hundreds of thousands of lines cleared. A picture of the best network produced (both overall and with this approach) is shown in figure 7.

The importance and role of each feature is explored in the next section.

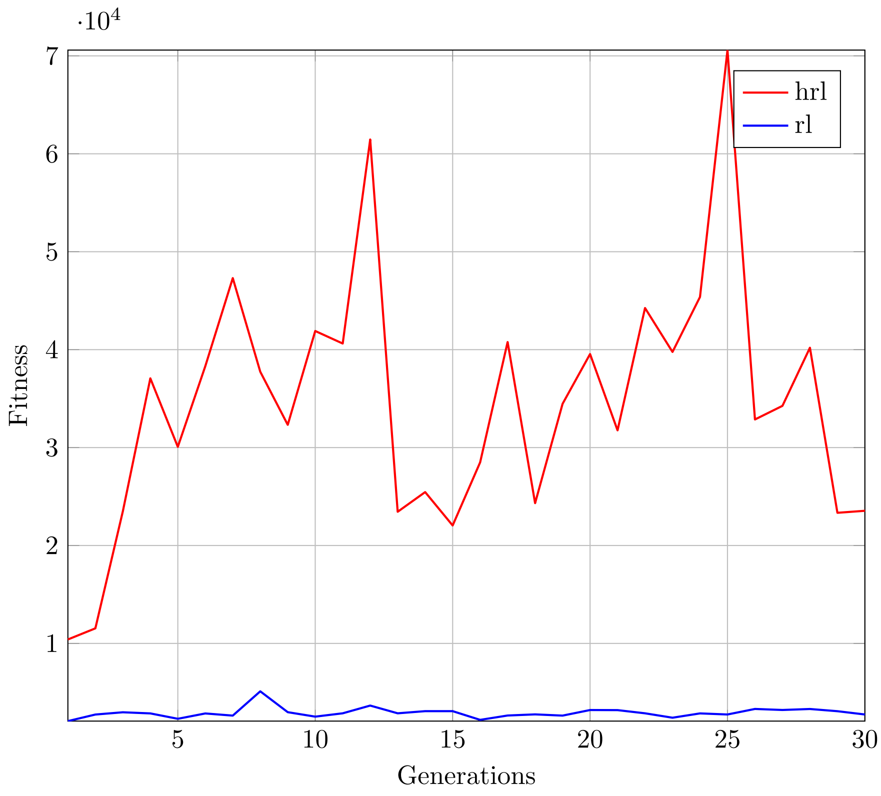

Figure 4: The maximum fitness when training over 30 generations. Plot —— uses **rows** and **landing height** while plot —— uses **rows**, **landing height** and **holes**.

4.4.1 Feature Analysis

The general goal of Tetris is to eliminate as many rows as possible (that is the main way to get score) and to not fill the board to the top, because then the game is over and the ability to get more score is gone. This naturally justifies the existence of the **rows** and **landing height** features. The network should prefer actions that eliminates many rows while trying to place the pieces as close to the bottom as possible. Plot —— in figure 4 shows that this is not enough, there is an aspect missing, namely the number of holes. Minimizing the number of holes is important because a hole is just dead space, it can not be reached by a piece and hence can not form a full row. Plot —— in figure 4 shows that the introduction of the **holes** feature dramatically improves things.

Figure 5 illustrates the importance of each feature by showing the results of training sessions with all features except for one, one session for each omitted feature. Plot —— and plot —— perform worse than the other ones, which

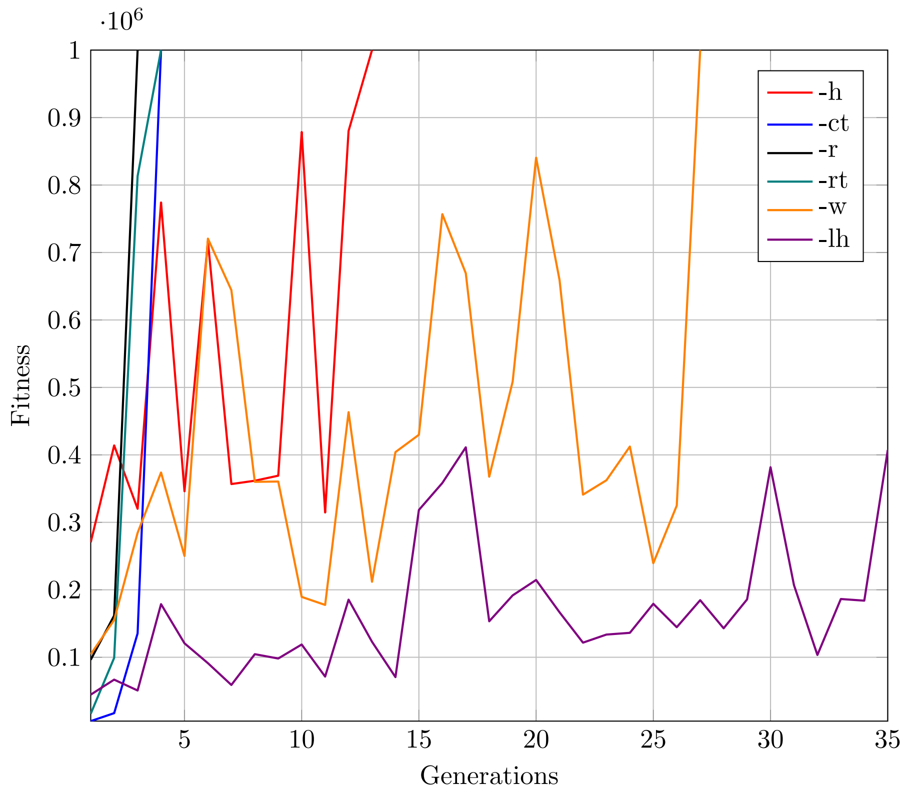

Figure 5: The maximum fitness when training with a score limit (fitness) of one million over 35 generations. Plot — has all features enabled except **holes**, plot — has all features except **column transitions**, plot — has all features except **rows**, plot — has all features except **row transitions**, plot — has all features except **wells** and finally plot — has all features except **landing height**.

makes sense because the landing height and the number of holes are important, as already discussed. This figure shows another feature that seems important, namely **wells** shown on plot —. Wells are not desirable because they can be hard to fill without leaving holes. A well with deep three or more can only be filled with a 1x4 piece, which is only one out of seven available pieces. The other plots in the same figure reached a score of one million quickly, indicating that **rows**, **row transitions** and **column transitions** are not that important. But maybe they are, they may just grow quickly in the beginning but are not able to reach a high score in the long run.

Figure 6 shows these three and some more up to a score of five million instead of one million. Having all features enabled (—) gives the best result out of everything, showing that all these six features are needed. The excluded **rows**

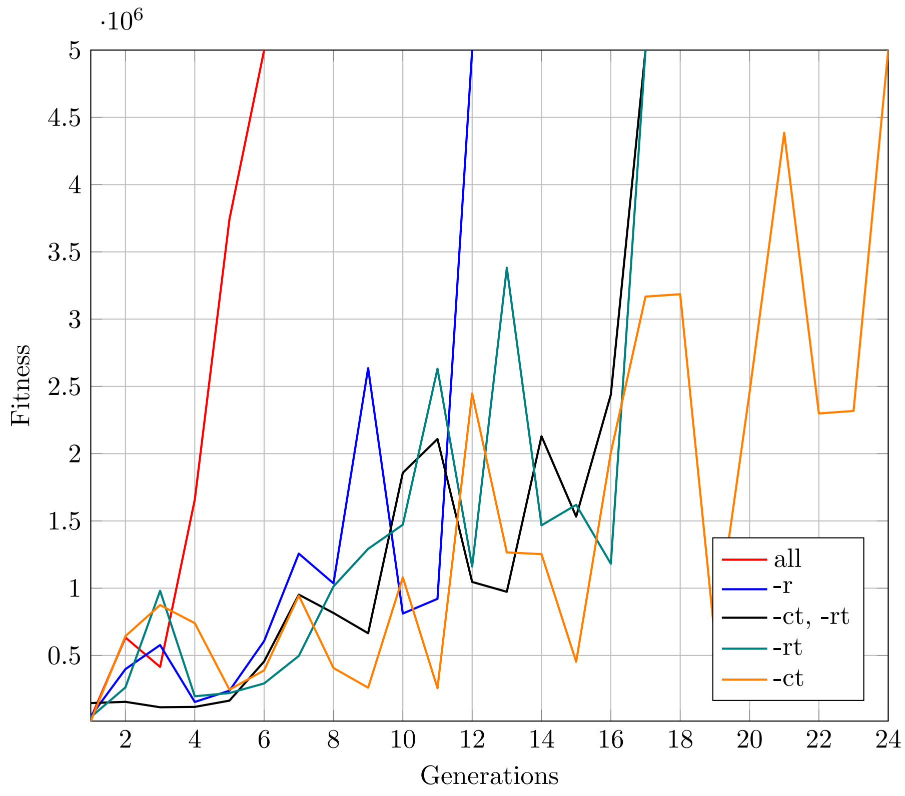

Figure 6: The maximum fitness when training with a score limit (fitness) of 5 million over 24 generations. Plot —— has all features enabled, plot —— has all features except **rows**, plot —— has all features except **column transitions** and **row transitions**, plot —— has all features except **row transitions** and finally plot —— has all features except **column transitions**.

runs from figure 5 did indeed not perform as well in the long run as seen in figure 4.

Both transition features plays the role of making sure that all pieces are placed compactly against each other, making sure that there are few gaps. Having pieces placed without gaps is good because it makes it easier for future pieces to be placed without creating holes. Figure 6 shows that training up to five million is a lot slower without any transition feature, so they are important.

5 Comparison to PSO

The successful approaches from section 4 all converged to the same network structure, an example of which can be seen in figure 7. The structure consists

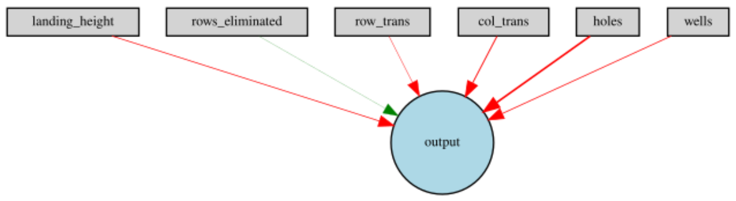

Figure 7: Illustration of a network that produced a fitness of 55 million using the approach described in section 4.4. The rectangles are the input nodes, the blue is the output node and the arrows are the weights where a red color is a negative weight and a green color is a positive weight. The thickness of each arrow illustrates how big the value of the weight is.

Wells	-2.2325745006784596
Holes	-5.468481414786788
Column Transitions	-2.9407012203149034
Row Transitions	-1.066752716475175
Rows Eliminated	0.21727304852256102
Landing Height	-2.078277000532176

of one output node with a linear activation function, no hidden nodes and all input nodes connected to the output node, meaning that the output of the network is a simple weighted sum of all inputs.

One advantage of using NEAT is the natural exploration of different network structures. Therefore, if NEAT consistently produces the same structure then perhaps NEAT is not the most suitable method for this problem. PSO can be used to optimize a neural network with a fixed structure by searching a n-dimensional space where each dimension represents a weight or a bias. In this case a seven dimensional method was used: one weight for each of the six inputs and one bias. PSO performed well in just a few iterations, oftentimes reaching a fitness of around 5 million in the first few iterations.

6 Conclusions and Future Work

Here we present various conclusions and potential future work or explorations that could be interesting to find answers to.

6.1 Approach conclusions

We managed to get a working Tetris AI that scored in the millions with our rule set. The best score was 55 million, which is about 500000 cleared lines. Our AI can most likely score higher

We can conclude from our different approaches that it is beneficial to make the work of the network as simple as possible. Having the network see every cell as input, as explained in section 4.1, was not good at all. The input space to the network was too big and the network had to make a complex decision. It ended up filling the board as much as possible instead of actually building rows. The key for fixing this was to offload some work of the network by having the network evaluate moves and perform the best one instead of directly deciding the move to make. That is what we did with the feature extraction approach explained in section 4.4. Simplifying and abstracting helped for Tetris and is most likely a good approach for other machine learning related problems as well.

All the features we used in the best and final approach, i.e. feature extraction (section 4.4), each contributed to something as shown in section 4.4.1. The results in that section is not that reliable because most of the runs were only run once. Tetris is stochastic, sometimes when training the network it is not able to get high scores at all and sometimes it reaches very high scores on the second generation. But even though we can not draw reliable conclusions with absolute confidence our test runs are at least somewhat representative of their actual behaviour. During the training sessions there were also score limits to reduce computing time, which further reduces the reliability of the test results.

Jacob Schrum et al.[5], whom did a similar study, got the same results i.e. raw input from the board did not work well and feature extraction worked better.

6.2 Approach future work

All of the approaches presented in this paper only looked at one piece at a time, but to consider two or more pieces at a time should be able to give better results. This can easily be done in with the features approach (section 4.4) by placing two pieces after each other before extracting all features. This will lead to many more possibilities to evaluate, which will lead to the question of if it is worth it? The potential for high scores might not be that much higher than the single piece approach, not making the extra computations worth it.

6.3 Training algorithm conclusions

The PSO algorithm is likely better suited for this problem due to its faster learning speed. This is in large part because PSO does not explore different network structures, so it can start optimizing the weights straight away and can propose a good solution early. However, for the PSO algorithm to work the structure of the network needs to be defined beforehand, which was provided by NEAT. This makes it difficult to conclude that one method is strictly better than the other, but in situations where the structure of the network is provided, PSO would be preferred over NEAT.

References

[1] Simón Algorta and Özgür Şimşek. The game of tetris in machine learning. *arXiv preprint arXiv:1905.01652*, 2019.

[2] Dimitri P Bertsekas and John N Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996.

[3] Heidi Burgiel. How to lose at tetris. *The Mathematical Gazette*, 81(491):194–200, 1997.

[4] Colin Fahey. Tetris AI. `https://www.colinfahey.com/tetris/tetris.html`, June 2003. Taken 2020-05-26.

[5] Lauren E Gillespie, Gabriela R Gonzalez, and Jacob Schrum. Comparing direct and indirect encodings using both raw and hand-designed features in tetris. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 179–186, 2017.

[6] John H Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. University of Michigan Press, Ann Arbor, Michigan, 1975.

[7] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4, 1995.

[8] Kenneth O Stanley and Risto Miikkulainen. Efficient reinforcement learning through evolving neural network topologies. *Proceedings of the Genetic and Evolutionary Computation Conference*, 2002.

[9] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.