A Deepdive into Tetris AI

Student Name: T.M.A. Hui

Supervisor Name: S. Dantchev

Submitted as part of the degree of [MEng Computer Science] to the

Board of Examiners in the Department of Computer Sciences, Durham University

***Abstract* —**

Context/Background

Deep learning and artificial intelligence for playing games is a widely researched field, with applications in tabletop games such as Chess, and video-games such as Tetris. For Tetris, superhuman endurance has been achieved, but not superhuman efficiency.

Aims

The aim is to compare various AI techniques, applied in an official version of Tetris, NES Tetris. Different heuristics will be tested, and the best agents are compared against human players.

Method

An emulator is used to create the Tetris environment. An interface serves as the intermediary between the game and the agent. Different agents, based on the Genetic Algorithm and Q-Learning, are implemented and their performance is measured.

Results

Results show that a Genetic Algorithm agent outperforms Deep Q-Learning agents, although it also suggests that the performance of the Genetic Algorithm agent is close to saturation. In comparison to top-level human players, neither agents are competitive.

Conclusions

This paper successfully implements, optimizes, and train multiple Tetris players using the Genetic Algorithm and Deep Q-Learning in NES Tetris. Players were able to play beyond random selection, however they were not competitive with high-level human players.

***Keywords* —** Tetris, Tetromino, Game Playing, Genetic Algorithm, Q-Learning, Reinforcement Learning, Emulator, Deep Learning, Convolutional Neural Networks, Artificial Intelligence

I INTRODUCTION

Artificial intelligence is a field focused on simulating human behaviour using a computer. Applying AI to video-games, computers can be trained to play various video-games, from classic ATARI games (Mnih et al. 2013), to modern competitive games such as DotA 2 (OpenAI 2018). This paper will focus on Tetris, a classic puzzle game.

Tetris is a game first developed by Alexey Pajitnov in 1984. It is a puzzle game based on stacking tetrominos: shapes built from 4 connected squares. By filling a row completely, the row/line is 'cleared', emptying the row and shifting all filled spaces above itself down. The score

for a line clear increases exponentially given the number of lines cleared with a single piece, hence clearing 4 lines at once is highly valued. This is called a Tetris. In singleplayer Tetris, when a piece is placed above the play-space, the game ends. This is often referred to as 'topping out'.

Since 1984, the game has undergone many changes, including the move to 7-bag piece generation (a method to reduce randomness of the piece generation, explained in Related Works Section C), and expansion of the spinning mechanics of tetrominos. Despite all this, NES Tetris, released in 1989, still remains a widely played version of the game, with a growing competitive scene and audience. This version of the game has two modes of play: A-type, which is a score-attack mode where players aim for a high score before topping out, and B-type, where a player tries to clear 25 lines with various amounts of 'garbage' generated on the play-space. Garbage refers to tiles that are placed in an inconvenient way, making rows difficult to fill.

Tetris has been the target of many game-learning applications, beginning with Colin Fahey's Tetris article (Fahey n.d.), containing his own implementation of a Tetris player. Since then, many implementations have been explored, from genetic algorithms based on a small group of heuristics, to the more recent Deep Q-Learning approaches. (Fahey n.d.) focused on extending the length of the game, training the player to survive. In contrast, this paper will focus on maximising the game score within the limits of human play, allowing for it's performance to be contextualized with regards to human performance.

A Background

In order to play Tetris well, a player must analyse the play-space and consider the most optimal placement for a given tetromino. This problem becomes deeper when a player is given information on the next piece, as is the case in NES Tetris. Ultimately, the game can be modelled as an optimization problem, where all configurations of the play space are given a score, and the player moves from play-space to play-space whilst maximising this score. Hence, the problem becomes a situation of how the play-space is scored. In this paper, two approaches are considered: the Genetic Algorithm, and Q-Learning.

The Genetic Algorithm is a stochastic population-based optimization algorithm, based on human evolution. This method has proven to be successful with regards to maximising line-clears, and has been used in many other applications in sectors such as the motor sector, medicine, and physics. In order to make use of the genetic algorithm, there must be some parameters to optimize. In this case, heuristics may be derived from the state of the play-space, and weights can be attributed to these heuristics. A look into such heuristics have been made, but this paper also proposes a lesser-known heuristic, known as parity, on top of those previously used.

Q-Learning is a reinforcement learning approach, where rewards are given to a player, depending on the results of the move it makes. It has been used in many game-based applications, most notably retro Atari games, and typically takes the entire frame of a game as the input, processing every frame. This is sensible in games where the entire screen space may constantly be changing, and any actions at any point may influence a player's score. On the other hand, Tetris contains more idle time, in the form of waiting for a piece to fall, during which the best input is often no input. Hence, this paper takes an approach of processing a single frame at the moment a new piece is generated. At that point, this paper hopes to extract the relevant information from the game state, filtering out a large amount of redundant data, such as the background. Approaches using the play-space structure, or a set of features derived from heuristics, have been used as the

input in prior works. The paper will compare the two options.

Another aspect that must be considered is the version of the game used in training. Most prior works choose to implement their own versions of Tetris, and these are often simplified to make interaction with the game less complex. A common feature of these Tetris implementations are that pieces do not have a fixed spawn position and orientation. This increases the range of possible placements a player can make, by ignoring any 'spires' - thin and tall stacks of blocks extruding from the rest of the surface of the game space. These spires typically prevent a player from moving the tetromino to certain positions. By using an emulation of an official Tetris game, this paper is forced to play with a fixed spawn position and orientation.

In addition, many of these prior works ignore certain game mechanics. These include 'spins', which are when tetrominos are spun in a certain position in order to fill a space that is otherwise unreachable, and 'tucks', which are when tetrominos are moved horizontally into an empty gap. This paper will try to make human-like use of these mechanics.

B Objectives

This paper aims to compare the performance of different approaches to the game of Tetris, and contextualize them with regards to human performance. Hence, the following objectives were proposed, sorted into three levels of difficulty.

The minimum objectives are to simply implement a hand-tuned heuristic player, capable of playing NES Tetris through an emulator. Hence, an interface for which the player interacts with the emulator is required, and an investigation into different heuristics is made. The game should be easily viewable as well, for demonstration purposes.

The intermediate objectives focus on optimizing the heuristic player, using the genetic algorithm. Following that, a deep Q-Learning (DQN) approach will be implemented and trained. These players will be compared to human-players.

The advanced objectives involve optimizing the DQN to maximise performance. This will include the optimization of hyperparameters, a comparison of different input formats for the network, and modifications to the training process. Furthermore, a deeper look into the different heuristics will be made, comparing their effect on the performance of the Tetris player.

Throughout all these objectives, this paper aims to maximise the game score achieved prior to level 29, where the drop speed of tetrominos is made too fast for the majority of human players. In turn, the number of clearable lines is limited, preventing the player from playing infinitely, forcing it to improve it's scoring efficiency.

II RELATED WORK

A Reinforcement Learning

Reinforcement Learning (Sutton & Barto 2018) is a method of training a computer to execute a certain sequence of decisions in order to accomplish a scenario, basing itself on the way humans would interact with something in order to learn how to use it. And just as a human begins as a novice when pursuing a new skill, the computer is given only the rules of the game as a starting point. It is very commonly used in game playing, having most notably been applied to games such as Chess, Go, and Reversi. AlphaGo Zero (D. Silver 2017) successfully created an AI capable of superhuman play in Go, and building off of that is AlphaZero (Silver et al. 2017, Silver

et al. 2018), a general purpose algorithm capable of learning other games of similar format, namely Chess and Shogi. (Schrittwieser et al. 2020) further expands the range of learnable games, introducing MuZero, capable of learning both turn-based and video games.

A step in a different direction is shown in (Mnih et al. 2013), where reinforcement learning is applied to Atari games rather than board games. Such games use a gamepad for input, and outputs visual information. (Mnih et al. 2013) uses a combination of Convolutional Neural Networks with Q-Learning to achieve this, taking an image as an input, and returning an output for every valid action.

B Tetris AI

Tetris AI gained popularity with the introduction of Colin Fahey's Tetris article (Fahey n.d.). (Thiery & Scherrer 2009) and (Algorta & Simsek 2019) compares Fahey's AI, along with many other controllers, and (Thiery & Scherrer 2009) specifically categorises them into one-piece and two-piece controllers. One-piece controllers, such as Pierre Dellacherie's (Fahey n.d.), only take into account the current piece, whereas two-piece controllers, such as Fahey's, look at the next piece as well. (Fahey n.d.) also makes a corresponding one-piece controller, which shows poor performance in comparison to leading one-piece controllers and Fahey's own two-piece controller, showing the benefits of knowing the next piece. No concrete conclusion was found in (Thiery & Scherrer 2009), claiming that comparisons aren't reliable due to small differences having a significant effect on the game-score, these differences being the many different heuristics being observed by the separate controllers.

In addition, many papers use a simplified version of Tetris, which generates a tetromino in a specified column and orientation as in (Szita & Lörincz 2006, Bertsekas & Tsitsiklis 1996). (Fahey n.d.) produces an AI for a game closer to Classic Tetris, where a piece is generated in the middle column, and must be moved as the piece falls, adding another level of complexity.

In order to have the computer player learn Tetris, a method of allowing the program to interact with a Tetris game is required. A naive approach would be to feed unprocessed frames of the Tetris game to the network, as in (Stevens & Pradhan 2016). However, this project hopes to process each frame and extract information from the board, beyond visual information. (Carr 2005) showcases an example that uses the differences in heights across columns. In this project, the program must extract up to four different details from the play screen: the board structure, the next piece, the current score and the current level. In that regard, this project is similar to hand-coded controllers, in that some kind of abstraction of the Tetris board is needed. Using an emulator, such as FCEUX, for NES Tetris, the option of obtaining information from the RAM itself is possible, using a RAM map described in (Romhacking.net 2018). This technique is often used in Tool-Assisted Superplay/Speedruns (TAS).

Q-Learning has also been used to train Tetris AI (Stevens & Pradhan 2016), where heuristics are used to aid the learning process. (Szita & Lörincz 2006) goes deeper into this approach, claiming that Q-Learning alone results in early convergence of the network, and proposes a cross-entropy technique to avoid early convergence. (Lundgaard & Mckee 2007) looks into the suitability of reinforcement learning for Tetris, and compares the different agents (random, heuristic, Q-learning, neural network) and their respective performance in different categories, concluding that reinforcement learning, by itself, is suitable for Tetris, given a stronger agent to train against, yet does not necessarily outperform certain heuristic agents.

In this project, a heuristic agent will be created, with reference to Dellacherie's one-piece

controller, reverse-engineered in (Thiery & Scherrer 2009). A new heuristic will be tested as well, building on the theoretical concept of parity (Hard-Drop n.d.), which concludes that given good parity, there are more 'good' positions to place a tetromino. Weights will initially be manually tuned, similarly to Dellacherie's controller (Thiery & Scherrer 2009). Following the basic heuristic agent, a learning aspect will be applied to it, to observe the effects of reinforcement learning on a supervised agent.

For a heuristic agent, a naive method such as (Bergmark 2015) would work; converted to this project's scenario, a breadth first search would be done on two levels, one for each tetromino, and the highest scoring arrangement would be selected. This would not be suitable if the agent were to attempt to predict more moves ahead, or perhaps if the next piece was unknown.

Our contribution is the investigation of current AI techniques in score maximization, in opposition to prior work where survival is prioritized. This means to train an agent that plays with a high efficiency. It should prefer to build the play-space such that 4-line clears are available, and avoid 3-line clears or less. Prior works such as (Fahey n.d., Lundgaard & Mckee 2007, Stevens & Pradhan 2016, Szita & Lörincz 2006) measure the number of lines cleared as the score instead, hence multiple-line clears are not weighted more heavily in training.

C Tetris Mechanics

From the introduction of Tetris, the Tetris Company have continually updated the specifications of what can be deemed a Tetris game, known as the Tetris Guidelines, as well as introducing new mechanics, such as garbage in multiplayer modes, and complex spin mechanics. This project is targeted at Classic NES Tetris, hence many newer features are yet to be introduced. On the other hand, features that were phased out of the game remain. Most notably, the lock-in time for a dropping piece is significantly shorter than in modern Tetris, resulting in much less time for a player to maneuver their piece, and reduced room for error with missdrops.

A further important introduction to modern Tetris is the use of 7-bag randomisation. This means all 7 tetrominos are randomised in a bag before being placed in the piece queue. As a result, the game becomes less difficult, as 'droughts' do not occur (where a certain piece does not get generated for an extended sequence).

Specific to NES Tetris, the game operates at 60 frames per second, and the drop speed of a piece is determined by the number of frames required for a piece to fall one tile. In competitive play, players start at level 18, where pieces drop every 3 frames. From level 19-28, pieces will drop every 2 frames. At level 29 and beyond, pieces will drop one tile every frame. For an in-depth breakdown of NES Tetris, see (Meatfighter 2014).

Two playstyles have emerged in NES Tetris (Laroche n.d.). These playstyles dictate the speed at which a player can move tetrominos across the board. DAS, shorthand for Delayed Auto-Shift, moves tetrominos at most one tile every 6 frames, or 10Hz. On the other hand, hypertapping relies on the speed at which a player can tap a direction button. For top players, this can be between 15-20Hz. At the maximum drop speed (level 29), DAS is ineffective as it becomes impossible to fill the sides of the board, even on a clear board, hence hypertapping becomes the only option. However, if the AI were to be allowed to hypertap unrestricted, the consistency of a computer player may result in infinitely long games, and the AI can simply clear 1 line at a time to slowly reach a greater score than the opponent, ignoring higher scoring tetrises. To circumvent this issue, this project limits the maximum level such that the game ends after level 28, thereby limiting the number of lines available to be cleared.

III SOLUTION

A Overview

To create a Tetris AI for NES Tetris, an NES emulation environment is used to play a ROM for Tetris. An interface is designed such that an agent can easily retrieve game-state information, including the board structure, next piece and possible moves. The agent should then be able to select one of the possible moves and pass it to the interface, which emulates the button presses required to complete the move. The interface should notify the agent when a move can be selected, and the agent should be constantly waiting to make a move.

In order to make the agent play the game well, two algorithms are tested:

- Genetic Algorithm: A genetic algorithm is applied to a set of heuristics that describe the board's state. These heuristics may be calculated from the game-state information.
- Q-Learning: A deep neural network is trained to select the most rewarding move, based on a scoring function that is applied to every move.

In both algorithms, the agent is provided a set of possible next-states, generated with regards to the current state (play-field, next-piece), and the agent learns to select the best next-state to 'move' to. This is in contrast to more classical Q-Learning approaches where the agent learns to select the best controller-action based on the current state.

B Game Interface

B.1 Programming Environment

In order to teach a computer to play Tetris, there must be some Tetris game-environment for the computer to interact with. Specifically with NES Tetris, an interface that can interact with NES games is also required. Two options were considered: using the FCEUX emulator, which has built-in Lua scripting support, or using the nes-py module in Python. Using a dedicated emulator provides the advantage of having a simple interface to monitor game progress, as well as realtime modification to certain emulation parameters, such as emulation speed. However, support for deep learning is limited in the FCEUX environment, hence Python was chosen as the programming environment. Furthermore, Python allows for multiprocessing, allowing for multiple games to be played simultaneously, reducing the training time.

B.2 Game-State Parsing

The nes-py module provides a simple way for direct memory access to the emulated console. The following are the main areas of focus:

- 16-bit Seed: found in locations $0017-$0018, this seed is typically randomly generated, but can be manually set. The seed is used in the RNG-based tetromino generator, hence by fixing the seed, the same sequence of tetrominoes can be played repeatedly.
- Current Piece: the current piece occupies 3 memory locations, from $0040 to $0042. ($0040, $0041) represents the (x,y) positions of the piece, whilst $0042 represents the piece ID (see Piece IDs paragraph)

- Next Piece: the next piece's ID is held in location $0019.
- Game Phase: found in location $0048, the game phase is used to indicate playable state of the game. A value of 1 indicates that the tetrimino can be moved, whilst all other values (2-8) indicate that the tetrimino is being placed and can't be moved.
- Level: found in location $0044, the current level is used to determine the speed at which tetrimonoes are falling.
- Lines and Score: found in location $0050-$0052 and $0053-$0054 respectively, in the form of a Binary-Coded Decimal (BCD) in little endian format, this denotes the current number of lines cleared and the score achieved in the game.
- Board Configuration: found in locations $0400-$04C7, every 10 locations represent a row of the Tetris board, from left to right, top to bottom. An empty tile is denoted by a 0 value.

Piece IDs In Tetris, there are 7 different tetrominos, each with up to 4 different orientations, pictured in 1. In the order they appear (left to right, top to bottom), they are named the T, J, Z, O, S, L and I pieces. NES Tetris treats every orientation as separate pieces, and assigns an ID to each piece as such. Each piece has a centre-point denoted by a green dot, for which their (X, Y) coordinates are based on. Furthermore, rotations also occur with this centre-point as the pivot (excluding the O piece).

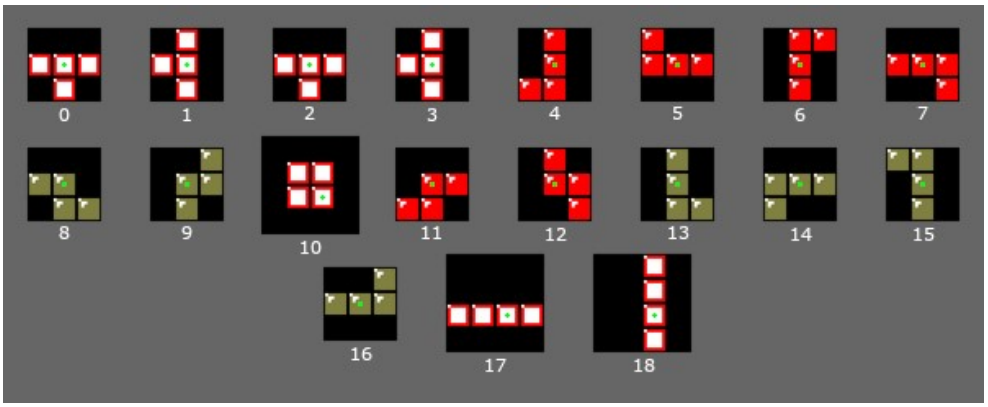


Figure 1: All piece orientations with their piece IDs

B.3 Interaction Layer

The interaction layer serves as the intermediary between the game and the agent. To further simplify what the agent has to do, this interaction layer also generates every possible move for the current piece, and optionally the next piece, storing the board configurations and the sequence of inputs required to reach that board configuration in a dictionary. The dictionary is indexed in the format $(X, Y, O(, X_{\text{next}}, Y_{\text{next}}, O_{\text{next}}))$, where X and Y are the placement coordinates, and O is the orientation of the piece, from 0 to 3, depending on the piece. The agent then accesses this dictionary the moment the game phase returns to 1. While the agent is selecting a move, the game is frozen.

Move Generation In order to generate moves given a board configuration and a piece, the piece is placed in every possible position on the board, and checked to see if that position is reachable from the spawn position. Firstly, every column, and some neighbouring columns, are considered iteratively. Neighbouring columns are selected depending on the width of the piece being placed, and it's orientation. The highest filled tile is found, and 2 is added to the Y-coordinate of that tile to find the highest possible placement position, Y_{max}. Then, the piece is placed iteratively in the middle column, from the bottom row up to Y_{max}. A placement is valid if the piece does not overlap with an already-filled tile. This process is repeated for every orientation of the piece.

To find whether a placement is reachable, the following restrictions are considered:

- Drop Speed: measured in frames per tile, the drop speed determines how many frames a piece can be moved for.
- Horizontal Speed: measured in frames per tile, this is fixed at 6 frames per tile, matching the auto-repeat rate of DAS.
- One input/action per frame: In real play, it is possible to spin the piece and move it horizontally in a single frame. However, this was inconsistent in the emulation environment, hence it is restricted to one action per frame.
- Spawn-Point: pieces have a fixed spawn-point in the 6th column from the left, in the top row.

From the placement position, iterating frame-by-frame, the piece is moved towards the spawn-point, with regards to the restrictions. For example, if a piece is moved horizontally, the piece is not allowed to move horizontally for 6 frames. Similarly, after 6 frames, the piece is automatically moved up. This is repeated until the piece is in column 6, and there are no filled tiles above the piece. If this condition is not met, then the placement is considered impossible. Once the condition is met, the sequence of moves are reversed and converted into controller inputs.

Controller Interface The interaction layer receives a sequence (X, Y, O) indicating the placement position and orientation of the piece. The interaction layer then finds the corresponding sequence of inputs required to place the piece in the given position and orientation by parsing the move-dictionary, and applies it to the game. The emulation environment recognises an 8 bit binary number as controller inputs, where each bit corresponds to a button as follows:

$$(\text{up}, \text{down}, \text{left}, \text{right}, \text{A}, \text{B}, \text{start}, \text{select})$$

C Training an Agent

When training an agent, games are started at level 18, and end after reaching level 29, limiting the number of lines to 230.

C.1 Heuristics

In order to train the genetic algorithm, a set of heuristics are required. 11 heuristics were selected:

Holes A commonly used heuristic for Tetris players, a tile on the board is considered a hole if it is not filled, and isolated from the top of the board.

Overhangs Similar to holes, overhangs are tiles that have at least one filled tile above itself, but aren't completely isolated like a hole.

Hole Depth For every hole, it's depth is determined by the distance to the top of the stack, within the same column.

Placement Height The placement height of a piece is a key consideration for the player, as higher placements can increase the risk of losing.

Jaggedness To measure the jaggedness, the absolute difference in the maximum height between adjacent columns are summed.

Slope The slope of the board is similar to it's jaggedness. However, the absolute difference is not considered. Instead, the height of the right column is subtracted from the height of the left column, for every column applicable, and the sum of that result determines the slope. In this implementation, a positive number is considered good. This heuristic is derived from high-level NES Tetris strategy.

Wells Similar to jaggedness, the wells heuristic finds unevenness in the stack. However, it punishes deeper wells, defined by the minimum difference in height with adjacent columns.

Right-side Well Although wells can be indicative of a bad stack, it is also important to maintain a single well, in order to maximise Tetris rate. This heuristic punishes any filled tiles located in the far-right column. This heuristic is also derived from high-level NES Tetris strategy.

Parity A less commonly known heuristic, parity is a heuristic that many top NES Tetris players plan around, in order to maximise the number of 'good' piece placements. By assigning +1 and -1 to every tile on the board in a checkerboard fashion, the parity is calculated as in 2e. The difference in number of the black/white tiles is the parity value.

Cleared Lines When a piece is placed, there is a possibility of completing, hence clearing, a line. The reward increases exponentially for every extra line cleared with a single piece. Hence, to encourage 4-line clears, 4-line clears are rewarded greatly, whereas anything less is punished.

Next Piece The next piece can also be considered a heuristic. However, it is difficult to generate a reward function for, hence it is more useful in a Q-Learning setting. Instead, for the genetic algorithm, brute force is used to generate all possible placements for the current and next piece.

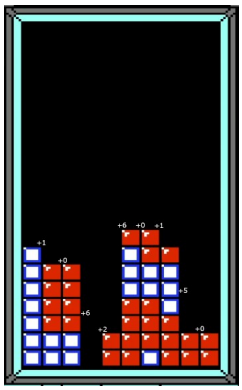
C.2 Genetic Algorithm

A genetic algorithm (GA) is trained using the above heuristics, with the goal of maximizing the following scoring function:

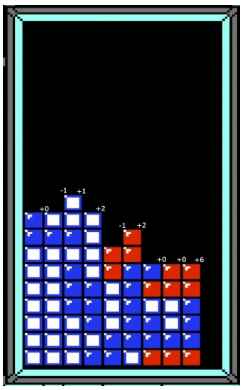
$$\begin{aligned} &- \text{Holes} - \text{Overhangs} - \text{Hole Depth} - \text{Placement Height} - \text{Jaggedness} + \text{Slope} - \text{Wells} \\ &- \text{Right Well} - \text{Parity} - \text{1-Line Clears} - \text{2-Line Clears} - \text{3-Line Clears} + \text{4-Line Clears} \end{aligned} \quad (1)$$

Each item of the function has a trainable weight, for which the GA will optimize in order to maximize the score. For training, 150 players are initialized with random weights. Each player is given an identical set of 5 random seeds, for which the game is played on, ensuring fairness when comparing performance. The average score of the 5 games are found for every player, and the top 20% of players move on to create the next generation of size 100. When producing new players, a mutation probability of 0.03 is used. The algorithm is trained for 5 generations.

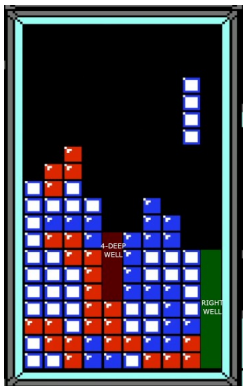
(a) A Tetris board with holes (red) and overhangs (yellow).



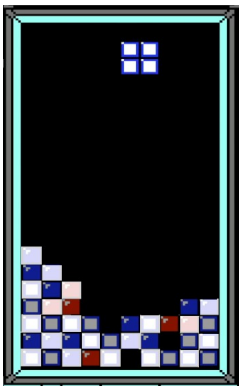
(b) A Tetris board depicting the jaggedness of a board.



(c) A Tetris board depicting the slope quality of a board.



(d) A Tetris board with a bad well (red) and a good right-well (green).



(e) A Tetris board with a checkerboard overlay on filled tiles.

Figure 2: Tetris boards visualizing some heuristics.

C.3 Deep Q-Learning

A Q-Learning agent is trained to play Tetris, using one or more of the following heuristics:

Index	0	1	2	3	4	5	6	7
Feature	Next Piece	Holes	Overhangs	Hole Depth	Jaggedness	Quality of Slope	Wells	Parity

Table 1: Heuristics and their corresponding index

All agents use the next-piece (0) heuristic as a minimum requirement for a two-piece controller. 3 input-types are also tested:

1. An extra row is appended to the playspace, containing the next piece.
2. The playspace is omitted, and a list of heuristics is used as the input, as in (Nguyen n.d.).
3. A combination is used: the playspace is passed through some convolutions, and the heuristic features are concatenated with the convoluted features.

The network architecture for the last input-type itself is depicted in 3. An important design decision is that the convolutions are of size 5, accomodating for the dimensions of the I-piece.

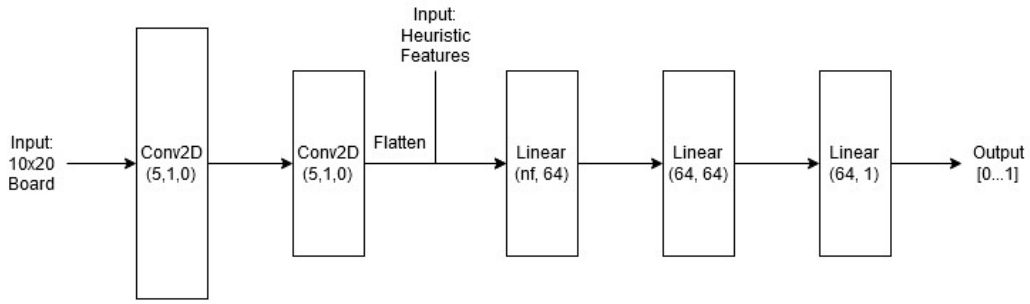


Figure 3: Network Architecture for Deep Q-Learning

During training, two separate networks are used: one is constantly trained with every game/epoch, and one is updated every 15 games, where the main network's weights are copied onto the secondary network. The Adam optimizer is used, and loss is calculated using Mean-Squared Error.

Given that Tetris only rewards a player after a line is cleared, a separate scoring function is required. The reward table 2 is used instead. Hence, every piece placed is rewarded, in order to maximise the number of pieces placed per game. As for the rewards per lines cleared, these are selected to be proportionally identical to the game-score rewards per lines cleared.

Lines Cleared	Reward	Score Reward @ Level 18
0	1	0
1	10	760
2	25	1,900
3	75	5,700
4	300	22,800

Table 2: Reward Table for Q-Learning

An epsilon-greedy decay is applied using the following formula:

$$\epsilon = \epsilon_{\text{start}} - \min\left(\frac{\text{epoch}_{\text{curr}}}{\text{epoch}_{\text{end}}}, 1\right) * (\epsilon_{\text{start}} - \epsilon_{\text{end}})$$

with ϵ being the probability a move is selected randomly, ϵ_{start} being the starting probablility, ϵ_{end} being the final probability, $\text{epoch}_{\text{curr}}$ being the current epoch/game, and $\text{epoch}_{\text{end}}$ being the total number of epochs to decay over. In our configuration, $\epsilon_{\text{start}} = 0.999$, $\epsilon_{\text{end}} = 0.001$ and $\text{epoch}_{\text{end}} = 0.5 * \text{epoch}_{\text{total}}$.

In order to ensure fairness between training cycles, a fixed set of 10000 seeds is generated, and the agent trains by iterating through each seed until completion.

In order to determine the best configuration for the Deep-Q-Learning algorithm, the following are tested:

- Learning Rate

- Configuration of additional heuristics
- Games played/number of epochs (within achievable training times)

To ensure fairness between training cycles when comparing different Q-Learning agents, a fixed set of 10000 seeds is generated, and agents train by iterating through each seed until completion. These agents are trained for 5000 games.

IV RESULTS

This section details the experimental process, and presents the results gathered. These results will portray the performance of the agents, calculated as an average score. It will also visualize the changing performance of the Q-Learning agent over time.

A Experimental Setup

In order to compare the Q-Learning agents, every configuration is trained 3 times, in order to increase the reliability of the scores. Then, every agent is made to play games with 10 fixed seeds, (0,1,2,3,4,5,6,7,8,9). For each configuration of the Q-Learning agents, the scores from the 3 separately trained models will be averaged, and the scores for the 10 separate seeds will be averaged again. The standard deviation of these scores will be used to measure consistency. The highest achieved score by any model of a certain configuration is also observed.

For the Q-Learning agents, their performance is tracked using an average of their most recent 100 games played in training. The statistics being tracked include game score, reward score and line-clear count. In contrast, the GA agents are tracked solely on highest game-score per each generation. To determine the best input-format for the Q-Learning, these tracked statistics are compared.

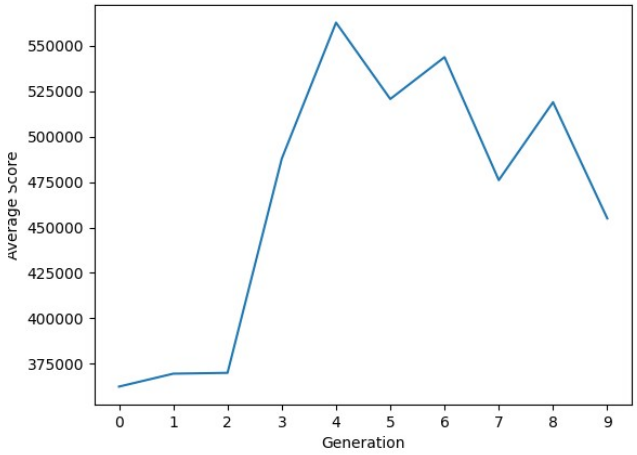
The best Q-Learning and GA agents are compared using the methods previously described.

B GA Performance

The GA shows good performance, with averages of over 500000 across 5 games after 5 generations. It's performance stagnates past the 5th generation, fluctuation between 450000 and 600000. With regards to the heuristic weights, they are as follows:

$$\begin{aligned}\text{Board Score} = &-0.97 * \text{Holes} - 0.47 * \text{Overhangs} - 0.41 * \text{Hole Depth} && (2)\\ &- 0.41 * \text{Placement Height} - 0.15 * \text{Jaggedness} + 0.22 * \text{Slope} && (3)\\ &- 0.46 * \text{Wells} - 0.33 * \text{Right Well} - 0.17 * \text{Parity} && (4)\\ &- 0.82 * \text{1-Line Clears} - 0.42 * \text{2-Line Clears} - 0.35 * \text{3-Line Clears} && (5)\\ &+ 0.82 * \text{4-Line Clears} && (6)\end{aligned}$$

As indicated, the GA has learned to prioritize holes the most, followed by 4-line clears. 1-line clears also have a high negative weight. The parity weight is the second smallest, with jaggedness having the smallest weight.



Generation	0	1	2	3	4
Highest Average Score	362,424	369,540	369,992	487,876	562,752
Generation	5	6	7	8	9
Highest Average Score	520,696	543,740	476,056	518,952	454,976

Figure 4: Highest average scores over 10 generations of the Genetic Algorithm

C Deep Q-Learning Performance

C.1 Input Format

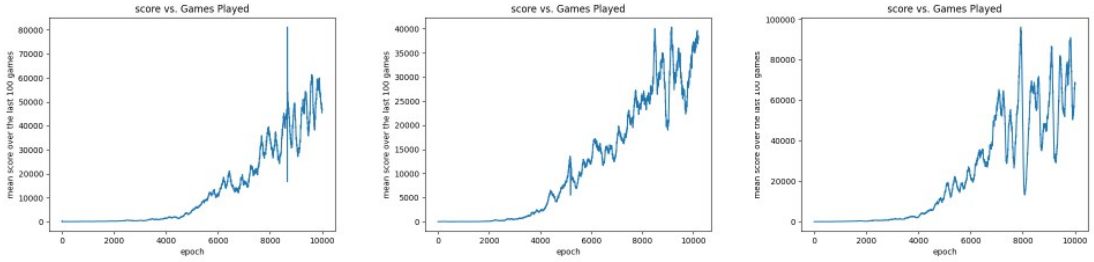
Method 1 outperforms method 2 by a factor of 2, whereas method 3 marginally outperforms method 2.

C.2 Learning Rate

A learning rate of 0.01 and 0.001 produce comparable results, as seen in 7, with the lower learning rate outperforming the higher learning rate in average score, highest score and consistency (standard deviation). A learning rate of 0.0001 is clearly too low, with a severe drop in performance in all metrics. At a learning rate of 0.01, the agent's standard deviation is greater than it's average score.

C.3 Heuristic Configuration

With regards to the selection of heuristics, 9 shows that using a combination of the next-piece (0) and jaggedness (4) produced the best average score. However, the highest achieved score was obtained using only the next-piece information. The novel feature, parity (7) produces comparatively good results as well, with the 2nd highest average score and 2nd greatest high-score. With regards to consistency, using the next-piece only results in the lowest standard deviation, implying a high consistency. Due to time constraints, larger subsets of heuristics could not be tested.

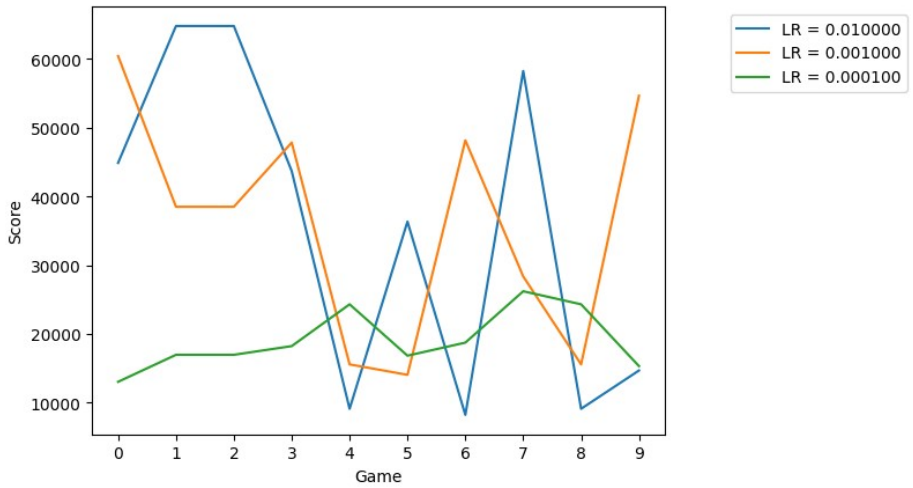


(a) Score over Time using Method 1 from C.3

(b) Score over Time using Method 2 from C.3

(c) Score over Time using Method 3 from C.3

Figure 5: Graphs visualizing the progress of an agent over 10000 games



Learning Rate	0.01	0.001	0.0001
Average Score	35,399	36,173	19,101
Standard Dev.	43,308	33,183	8,195
Highest Score	168,380	255,560	99,180

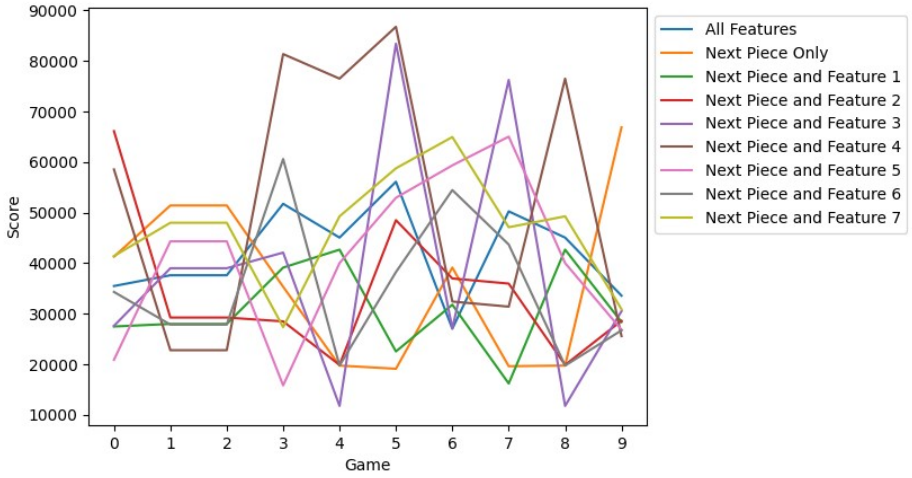
Figure 6: Average Scores across all 5 games

Figure 7: Scores for different learning rates, after 5000 games, averaged across 3 models

D Final Models

10 visualizes the improvement of an agent, trained on 30000 games. At the end of training, the agent averages a score of approximately 175000, clearing approximately 200 lines per game. Throughout training, there were many occasions where the agent would reach 230 lines.

11 shows a clear performance disparity between the best GA and DQN agents, with the GA agent being better on average, and reaching a higher score. However, the DQN shows much more consistency, as shown by the lower standard deviation. Neither are comparable to a top human player, however the GA shares a similar behaviour in that the standard deviation is very high.



Features	0	0, 1	0, 2	0, 3	0, 4
Average Score	36,379	30,691	34,301	38,861	51,472
Standard Dev.	21,017	23,965	27,347	47,515	39,097
Highest Score	255,560	148,580	206,160	197,960	223,200

Features	0, 5	0, 6	0, 7	ALL
Average Score	40,965	35,334	46,510	41,964
Standard Dev.	34,377	23,348	32,762	27,504
Highest Score	222,060	166,460	223,220	214,660

Figure 8: Average Scores across all 5 games

Figure 9: Scores for different feature configurations, after 5000 games, averaged across 3 models. The next-piece (0) heuristic is featured in all configurations, making the agent a 2-piece controller.

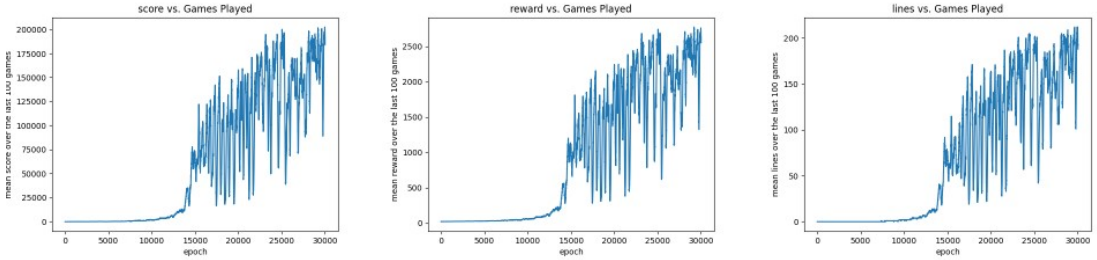
V EVALUATION AND DISCUSSION

In this section, the results are discussed, and the strengths and limitations of the tested solutions are explored and contextualized with regards to human performance.

A Comparison of Algorithms

The GA agent is capable of averaging a score of over 500000 after 10 generations. On the other hand, the Q-Learning agent is far less capable - after 5000 games, the model struggles to consistently reach a score of 100000, as seen in 2. Even after 20000 games, the Q-Learning agent achieves a maximum of only 334180. It is important to note, however, that the agent hasn't been trained to saturation, as indicated by 10, where the trend is still upwards. This implies that there is still room for the agent to improve/learn. In contrast, the GA agent shows stagnation after 6 generations.

Looking at the standard deviations, almost all configurations of the Q-Learning agent resulted in a standard deviation of at least 50% of the mean. This can be attributed to the randomness of Tetris, as well as the short training process of only 5000 games. For example, the final GA



(a) Mean score from the past 100 games (b) Mean reward from the past 100 games (c) Mean line-clears from the past 100 games

Figure 10: Graphs visualizing the progress of an agent over 20000 games

model achieves a standard deviation of 10% of it's average score, having been trained longer, and capable of scoring higher.

In game-play observations, it could be seen that the GA agent is very passive, often clearing 1/2/3 lines to maintain a lower and safer play-field, as opposed to letting the play-field build up and waiting for 4-line clears. Similar behaviour is displayed by the Q-Learning agent, but the play-field is generally much lower quality.

A.1 Heuristic Analysis

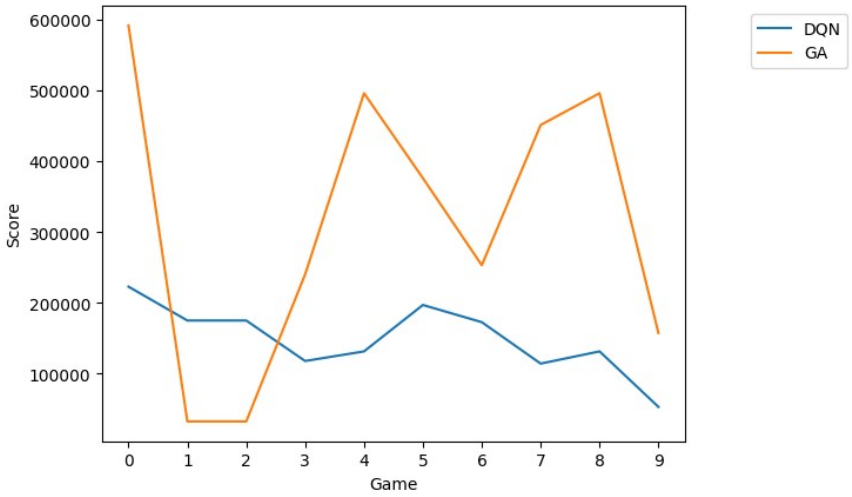
Interestingly, the graph in 9 shows no clear correlation between heuristics selection and performance; rather, certain configurations perform better than others in certain games. For example, the (0,2) and (0,4) agents perform very well in seeded games 0 and 1, but are clearly worse than other configurations in seeded games 2, 3, 4, and 5. Furthermore, no two networks perform similarly to each other. Overall, it is difficult to decisively determine which features were most beneficial to the training of the agent.

Interestingly, the configuration using the parity heuristic performs comparatively well with a Q-Learning agent, whereas the GA weight would imply poorer performance in comparison to other configurations.

B Comparison to Real Players

The GA agent demonstrates clearly above-average play in comparison to human play. In context of human performance, the GA agent's highest score places it in 85th place in (Tetris-Concept-Forums n.d.). However, this is far from superhuman performance, given that player Joseph Saelee (jdmfx_) achieved a score of 999999 in approximately 178 lines. As for the Q-Learning agent, it's score places it at approximately 150th in (Tetris-Concept-Forums n.d.)'s rankings.

Something that isn't shown in (Tetris-Concept-Forums n.d.) is the consistency of a player, which is especially important in competitive play. (Franklin n.d.) lists the qualifying scores for the most recent Classic Tetris World Champions (CTWC) event, which shows the number of 'maxouts', i.e. scores of 999999, a player obtains in a period of 2 hours. In particular, Joseph Saelee plays a total of 16 games, maxing out in 12. This is a testament to how much better a human player currently is at Tetris, compared to current solutions, in both consistency and ability.



Algorithm	GA	DQN	Top Human*
Average Score	312,648	148,908	857,753
Standard Dev.	189,365	46,599	299,295
Highest Score	737,930	334,180	999,999

Figure 11: Comparison of the best GA agent, best DQN agent, and top human player (Classic-Tetris-YouTube-Channel 2020)

However, Joseph Saelee is a hypertapper who manually moves the piece laterally with individual key-presses, as is many of the players in the CTWC Qualifiers. The agents in this paper are limited to moving the piece laterally at a rate of 10Hz, in line with the limitations of Delayed Auto-Shift, whereas hypertappers are known to move the piece at up to 20Hz. As a result, the number of available moves for the AI agent is less than that for a hypertapper.

C Solution Limitations

The results suggest that a GA or basic Deep Q-Learning approach is insufficient in reaching superhuman Tetris play. With regards to the GA, this may be explained by it's static heuristic functions/weights; Tetris is a very situational game, and a human player will often make a seemingly poor placement in order to prepare a future move. However, due to the greedy nature of the GA, only the best immediate move will be selected. The 2-piece controllers of this paper are designed to consider the current and next pieces when selecting an action, but are unable to strategize any deeper.

Similarly, the Deep Q-Learning agent in this paper is unable to strategize more than 2 moves ahead. This is a result of the network's inability to learn sequences of moves, and put moves into the context of the rest of the game. In particular, the network is trained on the reward from moving from a current-state to a next-state, for which the reward is entirely independent of any preceding/following moves within the same game. Similarly, this reward is unaffected by the final score achieved in the game. Hence, two moves in separate games may be rewarded equally, despite a possible difference in final scores.

With regards to the interface, the solution does not perfectly emulate Delayed Auto-Shift,

instead using a fixed 6-frame delay between D-pad inputs (similar to a hypertapper at 10Hz). Due to the lackluster performance of the agents, this does not greatly affect the validity of the comparisons to the performances of real players. Another slight concern is that the algorithm used to find all possible moves often makes mistakes when the maximum height of any column exceeds 18. At this height, newly generated pieces can overlap with previously filled tiles. The interface determines whether a position is valid by checking for overlapping tiles, hence if an overlap occurs at the spawn position, the interface is unable to generate any possible placements, returning only the spawn position. However, occasionally the piece is able to fall regardless, resulting in a mistake. This is only a minor issue, as at a competitive level, if any column reaches that height, it is very likely that the player is about to lose, due to the speed of the falling tetrominos.

A more important issue with the implementation is that currently, the game waits for the player to determine the best move. This idle time includes the generation of possible moves, pathfinding/input-sequence generation for moves, and the selection of moves. Though this may not be a large issue during training, it becomes particularly noticeable in demonstrations, and becomes an unrealistic representation of how the game is played by humans.

D Approach

The approach in this paper was to use an official version of the game as a platform for training Tetris agents. Ultimately, this proved to be a disadvantage, especially during training. Due to the emulator's requirement of generating every frame of the game, training times are greatly extended, with much of the time being idle time for the agent. Training speed can be increased in the case of the GA, where games can be played in parallel. However, this does not apply to a deep-learning agent.

Another flaw in the approach pertains to the experimental setup. As mentioned previously, it is difficult to determine the effects of each heuristic conclusively. Both the small number of training games and evaluation games likely contribute to the lack of separation between the results of different configurations/agents.

VI CONCLUSIONS

This project has explored the effectiveness of existing solutions for playing Tetris in a competitive manner, using an official version of Tetris as the training and testing environment. An effective interface has also been produced, allowing for interaction between an agent and the game. The average and highest scores achieved by these solutions have been detailed, and contextualized with regards to the performance of top-level human players.

The performance achieved in this paper shows that the Genetic Algorithm outperforms current Q-Learning approaches, however neither is able to compete with top-level human players. This paper was able to explore a subset of heuristics, but was unable to deeply analyse the effectiveness/importance of each one. The performance of the Q-Learning approach was able to be improved marginally through hyperparameter optimizations and changes to the input format, but it is still the worst-performing agent.

A *Further Work*

The implementations in this paper can be further improved. With regards to the interface, an update should be made such that the predictions of the agent can be made in parallel to the rendering of the game.

With regards to the agent's performance, it is likely that the GA implementation is close to saturation. Hence, focus should be directed towards deep-learning approaches. Adjustments to the network architecture should be tested. For example, using a Recurrent Neural Network, such as a Long Short-Term Memory model, would be an interesting idea, with their ability to capture contextual information and train on sequential data. Adjustments to the reward function should also be tested, such as weighting the reward of an action according to the final score achieved in the game. The final goal would be to achieve super-human performance.

In terms of the training process, a separate training environment should be made, mimicking the mechanics of NES Tetris as closely as possible, whilst eliminating the redundant in-between frames.

References

Algorta, S. & Simsek, Ö. (2019), 'The game of tetris in machine learning', *CoRR* **abs/1905.01652**.
URL: *http://arxiv.org/abs/1905.01652*

Bergmark, M. (2015), Tetris: A heuristic study : Using height-based weighing functions and breadth-first search heuristics for playing tetris. Bachelor's thesis, KTH, School of Engineering Sciences, Stockholm, Sweden.

Bertsekas, D. & Tsitsiklis, J. (1996), *Neuro-Dynamic Programming*, Athena Scientific, Belmont, MA.

Carr, D. (2005), Applying reinforcement learning to tetris.

Classic-Tetris-YouTube-Channel (2020), 'Tetris perfection??!! joseph saelee's historic 2020 ctwc qualifying run', https://www.youtube.com/watch?v=vVbeDEejKBQ.

D. Silver, J. Schrittwieser, K. S. e. a. (2017), 'Mastering the game of go without human knowledge', *Nature 550* pp. 354–349.

Fahey, C. (n.d.), 'Tetris', https://www.colinfahey.com/tetris/tetris.html.

Franklin, G. K. (n.d.), 'Ctwc 2020 qualifying scores', https://listfist.com/list-of-ctwc-2020-qualifying-scores. Accessed: 19/4/2021.

Hard-Drop (n.d.), 'Parity', https://harddrop.com/wiki/Parity. Accessed: 22/10/2020.

Laroche, S. (n.d.), 'What is das and hyper tapping in tetris', https://simon.lc/what-is-das-and-hyper-tapping-in-tetris. Accessed: 22/10/2020.

Lundgaard, N. & Mckee, B. (2007), Reinforcement learning and neural networks for tetris. Technical report, University of Oklahoma, Oklahoma.

Meatfighter (2014), 'Applying artificial intelligence to nintendo tetris', https://meatfighter.com/nintendotetrisai/?a=b#The_Mechanics_of_Nintendo_Tetris. Accessed: 22/10/2020.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. A. (2013), 'Playing atari with deep reinforcement learning', *CoRR* **abs/1312.5602**.
URL: *http://arxiv.org/abs/1312.5602*

Nguyen, V. (n.d.), 'Deep q-learning for playing tetris', https://github.com/uvipen/Tetris-deep-Q-learning-pytorch. Accessed: 23/10/2020.

OpenAI (2018), 'Openai five', https://blog.openai.com/openai-five/.

Romhacking.net (2018), 'Tetris ram map', https://datacrystal.romhacking.net/wiki/Tetris$_(NES)$: $RAM_map. Accessed : 23/10/2020$.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T. & et al. (2020), 'Mastering atari, go, chess and shogi by planning with a learned model', *Nature* **588**(7839), 604–609.
URL: *http://dx.doi.org/10.1038/s41586-020-03051-4*

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K. & Hassabis, D. (2017), 'Mastering chess and shogi by self-play with a general reinforcement learning algorithm', *CoRR* **abs/1712.01815**.
URL: *http://arxiv.org/abs/1712.01815*

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K. & Hassabis, D. (2018), 'A general reinforcement learning algorithm that masters chess, shogi, and go through self-play', *Science* **362**(6419), 1140–1144.
URL: *https://science.sciencemag.org/content/362/6419/1140*

Stevens, M. & Pradhan, S. (2016), Playing tetris with deep reinforcement learning.

Sutton, R. & Barto, A. (2018), *Reinforcement Learning: An Introduction (2nd Edition)*, MIT Press, Bradford Books, Cambridge, MA.

Szita, I. & Lörincz, A. (2006), 'Learning tetris using the noisy cross-entropy method', *Neural Computation* **18**(12), 2936–2941.

Tetris-Concept-Forums (n.d.), 'Nes tetris a-type leaderboard', https://tetrisconcept.net/threads/nes-ntsc-a-type.1918/. Accessed: 17/4/2021.

Thiery, C. & Scherrer, B. (2009), 'Building Controllers for Tetris', *International Computer Games Association Journal* **32**, 3–11.
URL: *https://hal.inria.fr/inria-00418954*