Deep Reinforcement Learning Applied to Tetris

Isaac Wolf

2009257

Dissertation

Swansea University
Prifysgol Abertawe

Department of Computer Science

Adran Gyfrifidureg

26th April 2024

Abstract

This paper demonstrates the results of training an agent to competitively play the modern editions of Tetris using deep reinforcement learning (DRL). We achieve this by utilising a deep Q-learning approach in combination with a linear fully connected network architecture that approximates which action the agent should take given the current observation space. By parsing a state space into the neural network with a reduced feature set we can significantly decrease the amount of redundant states that the agent would otherwise explore, thereby increasing the efficiency of the agents learning. Additionally, instead of allowing the agent to choose any action from the available action space, we only allow it to choose from a pre-configured list of the most effective combinations of actions for placing pieces onto the game board. This study hypothesises that utilising a strategy that scores points more efficiently through multiple line clears, could produce a stronger AI model than one that focuses solely on survival with single line clears. We explore this question by training and evaluating models within a custom Tetris environment. This environment introduces mechanics such as piece holding and predictable piece queue randomisation algorithms, which can increase the viability of standard Tetris strategies that would otherwise be too risky to use in classic editions.

Contents

List of Figures

List of Tables

1 Introduction

The reinforcement learning (RL) process of evolving through exploration and exploitation has applications in a large number of fields, including finance, robotics and healthcare, with RL algorithms constantly evolving to help tackle real world problems. It's this idea in combination with the objective of further improving the performance of Tetris AI models that formed as the motivation for this project. Puzzle games, such as Tetris, can offer a unique perspective on problem solving and has been the focus of many research papers. These games can help can researchers to further investigate neural network solutions and alternative feature representation for control issues [5], which can then be applied to fields such as computer vision, robotics, and even autonomous vehicles.

While the majority of this paper will be focused on the modern Tetris editions, it is also important to briefly cover the history surrounding the classic versions of Tetris that came before. Tetris was originally created by the Russian computer scientist Alexey Pajitnov in 1984 for the Electronika 60 where it saw massive popularity and sold millions of copies over the following 5 years. Then in 1989 its rights were sold to Nintendo who went on to release the game for the NES console where it saw global popularity and is still considered the gold standard for Tetris games today. The NES version's rule set is the one that most academic research focuses on when producing Tetris AI models as it contains the largest amount of pre-existing data to compare to when reviewing the success of an agents performance.

Tetris is primarily played on a grid with 10 columns and 20 rows where the player is provided with random pieces, as seen in Figure 1, to be placed onto the grid, where the current assortment of pieces that have been placed onto the grid is called the *stack*. The pieces then move down the board at an ever-increasing speed, forcing the player to quickly decide where best to place their current piece. The main goal is to fill out entire lines which in turn removes those pieces, belonging to the completed line, from the board. However, if any piece reaches the top of the board, the game is immediately ended, resetting the players score and wiping the board. This game loop provides the constant challenge of having to stack pieces in such a way that helps maximize their score while at the same time placing pieces in a way that sets them up for easier piece placements in the future. When attempting to score as highly as possible, the player should be aiming to clear multiple lines at once which nets a far greater number of points than clearing single lines at a time, this concept is covered in greater detail in Section 3.3.1.

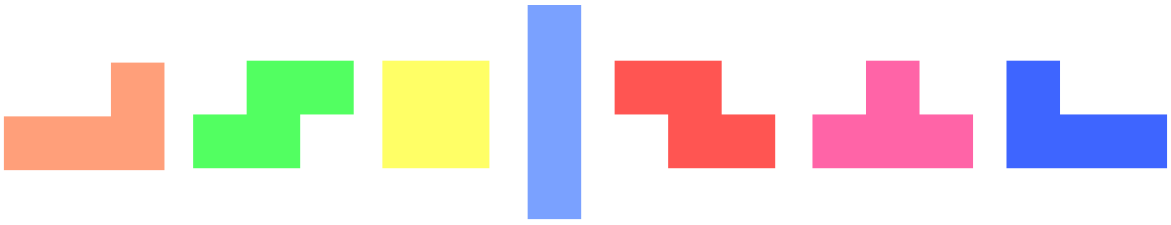

Figure 1: The seven pieces that make up Tetris. L, S, O, I, Z, T, J (Read from left to right).

As the main hypothesis, this paper aims to find out if multiple line clears could be a stronger strategy than one that is focused solely on survival with single line clears. These concepts are

explained in detail within Section 3.3.1. We explore this goal by applying DRL techniques to a modern Tetris environment and observing which strategies an agent would naturally develop when provided with guidance through a tailored reward system and high-level action space. The pytorch library and its detailed documentation [10] were primarily used for training the AI to explore and learn within a custom environment that implements the mechanics found within a modern Tetris game which are later covered in Section 3.2.

2 Related Work

Several studies have explored the use of DRL in the context of playing Tetris with one of the first papers being [4] which pioneered the application of RL in Tetris by demonstrating the effectiveness of Q-learning in conjunction with neural networks. Later, [3] extended this work by further discussing how neural networks could be utilised as a function approximators, potentially stabilizing training and thereby increasing performance. Building upon this, [7] investigated the role of different exploration strategies by using the points system within Tetris to construct a reward function that can remove the need for any supervised learning.

Together these papers form the backbone of research into Tetris related AI and have supported many papers in furthering their understanding of RL techniques and neural network models. One such paper that made use of this prior research is [12] which also used deep q-learning to develop an agent capable of playing Tetris at a competitive level by utilising convolutional neural networks as an alternative approach for their network architecture. In addition to this, [6] provides insight into the use of supervised back propagation networks for playing Tetris, in their research they demonstrate that these networks tend to over fit to specific game states, thereby hindering their ability to generalize and adapt to new situations.

3 Tetris

3.1 Custom Environment

The project implements a custom Tetris environment that was built from the ground up. All of the environment's visual components were added and handled using the pygame library. The implementation utilises an object orientated approach allowing the modular addition and removal of features depending on the type of model being trained. Example classes from the project's source code include:

- *Piece*. This is the parent class that all the different piece types inherits from. The class contains positional data and shape data about the piece, it also provides default implementations for piece movements and rotation.
- *Board*. This class handles data concerning the Tetris game board which is stored as a 2D array.
- *Window*. This class handles the drawing of all the game components to the screen.
- *Game_analysis*. This class stores all the functions that are used for analysing game states and piece information. E.g. checking for gaps in the board, checking differences in column heights and checking if the current stack is in a tetris ready state. These terms, and the usefulness of these functions, are further discussed in Section 4.5.

Having direct access to the underlying source code allowed for a wider range of reward calculation possibilities when fine tuning the agent between model training sessions, rather than being limited to design decisions of already existing Tetris titles or emulators from alternative sources.

3.2 Modern Features

There has already been a large amount of research around the classic versions of Tetris, and many attempts have achieved highly capable models that can both survive for long periods of time and accumulate large scores. To cover a less researched area of Tetris, this project focuses on the modern versions of Tetris such as Tetr.io [1] and Jstris [2] that introduce new rules and mechanics, allowing a greater freedom when deciding which strategy to follow. The main three features that modern Tetris introduces, and which have implemented into the custom environment, are:

- *Piece holding*. Rather than having to move a piece and decide where to place it, the player has the option of saving that piece in a single reserve space which can be accessed later in play if that piece is later required. This can be seen being used in Figure 2 on the left side of the environment holding the 'S' piece. However, the hold action cannot be performed in immediate succession with a previous hold action and will have no effect on the game state if attempted by the player. The way this nuance is mitigated is discussed in Section 4.5.2.

[1] https://tetris.wiki/TETR.IO
[2] https://tetris.wiki/Jstris

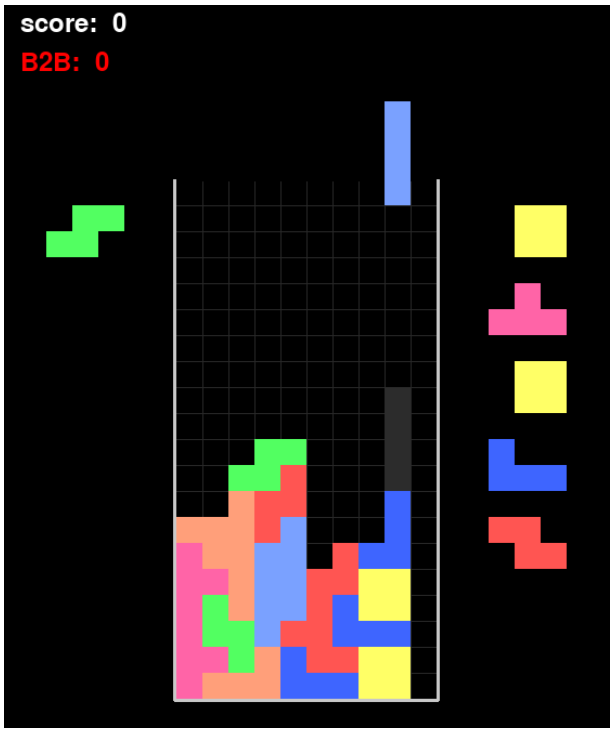

Figure 2: Example of a custom Tetris environment, with the green 'S' piece in the hold position, the current piece being placed being the blue 'I' piece and the next piece in the queue being the yellow 'O' piece.

- *5 piece queue*. This allows the player to view the next 5 upcoming pieces at all times, as seen in Figure 2 on the right side of the environment. This is vastly different from classic Tetris where the player would only have information about the single next piece they will be receiving.
- *7-bag system*. The way the order of the 5 piece queue is decided is through the 7-bag system [3]. This takes all 7 Tetris pieces and shuffles them, providing the same group of 7 pieces to the player each time except in a different order, guaranteeing that the longest you can go without a particular piece is 13 pieces.

3.3 Strategies

This section covers the main strategies that the agent utilised when learning to play competitively by scoring highly and managing the risk that comes along with such strategies to help create a balanced form of play. Some of these strategies were discovered solely by the agent through exploration and others were encouraged through the rewards system and high-level action space which are discussed in Sections 4.7 and 4.6 respectively.

[3] https://jstris.jezevec10.com/guide#randomizer

3.3.1 Tetris Line Clears

The *Tetris line clear* occurs when the 'I' piece is placed into the empty space inside or on either side of the stack, commonly called the *well* (an example of this can be seen on the right hand side of the stack in Figure 2). This clears 4 lines simultaneously and is the greatest number of line clears that can occur in a single piece placement. Clearing lines this way offers a far greater score than be achieved by clearing single lines at a time [7], as seen in Table 1.

Table 1: Tetris points system

Lines Cleared	Points
Single	100
Double	300
Triple	500
Tetris Line Clear	800
B2B Tetris Line Clear	1,200

3.3.2 Back-to-Back Tetris Line Clears

In modern Tetris games, achieving Tetris line clears repeatedly is called a *back-to-back (B2B)* and nets a massive score increase compared to spacing Tetris line clears apart. This score increase can be seen in Table 1. B2B combos are the basic strategy for most players when playing competitively in both single player and multiplayer environments. This strategy can only be performed if the player is able to not create any gaps in their stack, as a gap will forcibly break their B2B streak by making it impossible to clear exactly 4 lines without an incomplete line being left behind. It's this behaviour that this project focused on achieving and introduced the main problem of maintaining the balance between scoring highly while not stacking in such a risky way that the agent is then exhausted of available safe moves and is forced to create gaps on the stack, an example of this scenario can be found in Appendix B. One of the ways the agent manages to tackle this problem is discussed in following section.

3.3.3 Piece Burning

This strategy is described as dropping a non 'I' piece into the empty well that the player has created within the stack, this benefits the player by potentially expending a piece that might not be suited to being placed directly onto the stack. However, the player then must concede any B2B combo that they had built up as explained in the previous section. The option of burning a piece allows the player to follow a riskier strategy by having the option of briefly escaping from a potentially difficult situation. An example of such as situation is displayed both Figure 3 and Figure 4 below.

For this strategy, we can hypothesise that given an agent that is capable of stacking optimally, piece burning will always be a less efficient strategy than purely stacking for B2B combos. This is because piece burning forcibly breaks the players combo, reducing their maximum potential score. We can come to this conclusion when we consider that, if all the risk of making mistakes is removed by an agent that stacks optimally, then there should be no reason to utilise a strategy

which is designed to mitigate stacking risks. The validity of this hypothesis is later explored in Section 6.2.2.

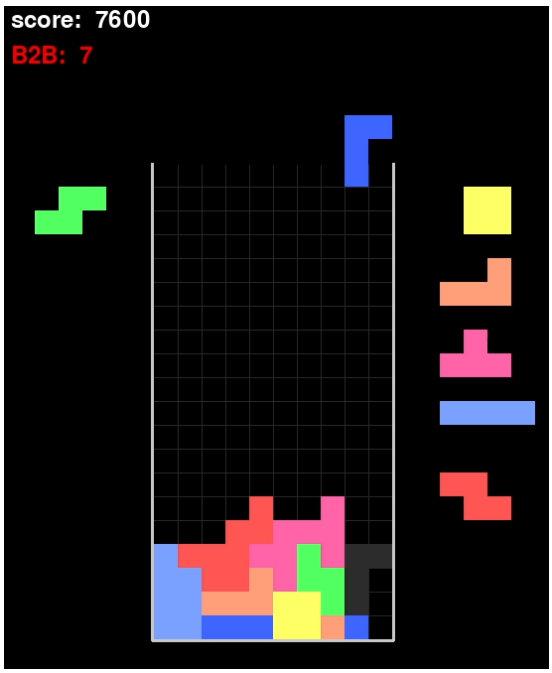

Figure 3: Example of the 'J' piece being dropped into the well to burn the piece, breaking the B2B combo but keeping a stack without any gaps.

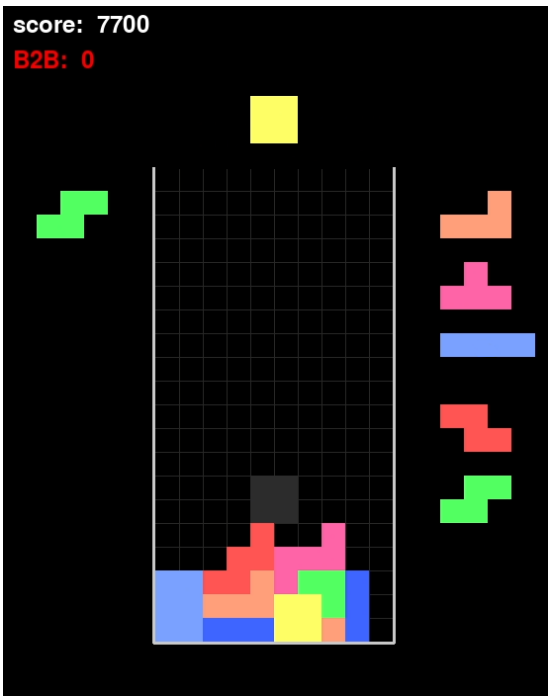

Figure 4: The stack after the piece from Figure 3 has been dropped, showing that a stack without any gaps has been maintained, however the B2B combo has reset to zero.

4 Methods

Since Tetris has been determined to be an NP-complete problem [2] we can confidently make use of RL knowing that it excels at solving problems that require approximation over precise search algorithms [3]. RL does this by using a trial and error approach, where an agent explores the environment and receives feedback in the form of rewards and penalties. This allows them to adjust their strategy in real time to achieve their objectives. This behaviour is achieved by forming a policy that creates a mapping between states and actions within the environment with the end goal being to optimise that policy and maximise the agent's rewards [1].

4.1 Deep Q-Network Algorithm

Q-learning is a RL algorithm which is used to describe a function that can find the optimal policy $\pi^*(s)$, where the optimal policy is the strategy that the agent should follow to maximise its rewards. This function works by taking a state-action pair (s, a) that maximises the predicted future reward $Q^*(s,a)$, also known as the Q-value.

$$\pi^*(s) = argmax_a \, Q^*(s,a) \tag{1}$$

To model this function, we want to be able to approximate these Q-values so the agent can take the highest rewarding action available from any state. In standard Q-learning we build a table that keeps track of these Q-value approximations for all the state-action pairs that the agent has explored, which would then expand linearly with every new pair that is discovered. This Q-table approach works for smaller environments with simpler state spaces but struggles to produce optimal learning policies in environments with more complex state and actions spaces. This limitation is further investigated in [9] which explains how Q-learning needs extensive exploration to achieve the same results of other similar RL methods and often results in slower convergence during training.

To overcome this limitation, we can use the success of [4] and how they utilize deep Q-networks (DQN) to help approximate these Q-values without the need for the inefficient Q-table. The Q-function that the DQN is attempting to model will follow the principles of the Bellman equation which takes into account any immediate rewards and any potential future rewards which are then discounted by the value γ due to their uncertainty [13].

$$Q^{\pi}(s,a) = r + \gamma Q^{\pi}(s', \pi(s')) \tag{2}$$

The temporal difference error is described as the difference between both sides of this equation. To reduce this error produced by the Bellman equation we implement the Huber Loss function which is effective at dealing with Q-values that are noisy and contain large outliers. It does this by acting as a quadratic function when the temporal difference is smaller but linear when the Q-value increases, making the temporal difference larger.

4.2 Epsilon Greedy

The epsilon-greedy policy is a fundamental strategy used in RL to balance exploration and exploitation. At every iteration of training, the agent must choose between exploring a new action or exploiting the currently estimated best action. This decision is selected by the parameter ε, which is a randomly generated number between 0 and 1. If that number is less than or equal to epsilon, the agent explores by selecting a random action, otherwise it exploits by selecting the action with the highest estimated value. This ensures that the agent occasionally explores new possibilities to discover potentially better actions while still primarily exploiting actions with higher expected rewards, thereby gradually improving its policy over time.

The ε parameter is applied to the training process by defining the following values:

- ε_{start}. This is the initial value for ε at the start of training. This value is usually close to or exactly 1, meaning the agent will almost always choose to explore when selecting an action.
- ε_{end}. This is the final value for ε at the end of training. This value is usually close to or exactly 0, causing the agent to almost always exploit when selecting an action.
- ε_{decay}. This value dictates the exponential rate of decay of epsilon where the lower the value is, the slower the rate of decay will be, i.e. the agent will explore the environment for a longer period of time.

4.3 Experience Replay Memory

To stabilize the training process of the neural network and make the gathering and analysis of data more efficient, we implement a common RL technique in the form of experience replay memory (ERM). This involves sampling from a buffer of state transitions, this helps the agent to adjust its learning based on the most recently discovered states while not completely forgetting previously encountered states that may potentially be important scenarios for the agent to understand. Two parameters must be defined when implementing ERM:

- *Buffer size*. The number of state transitions that can be stored in the experience replay memory at any one time.
- *Batch size*. The number of state transitions the agent should sample from the replay buffer when selecting an action.

The use of replay memory comes with its own set of problems that must be dealt with which is discussed extensively by [14]. This technique is a computationally expensive method when using increased buffer and batch sizes. However, for this project, experience replay sufficed as an appropriate way to stabilize the training process with the only tangible drawback being the increased time for the model to converge.

4.4 Network Architecture

Convolutional neural networks (CNNs) are a type of feed forward neural network that are made up of convolutional and pooling layers. The convolutional layers apply filters across the input

data which detects any local patterns while the pooling layers extract the most important features from the data. The more these layers are repeated in a network, the more complex the features the network will be able to detect. For Tetris, a CNN would appear to be best suited network architecture for solving Tetris related problems as the Tetris board can be simply treated as a 20×10 matrix with each pixel either being empty or occupied by a square, allowing its output to be analysed through pattern detection. However, after some experimentation with this architecture, it became clear that the agent struggled to perform multiple line clears and instead primarily focused on clearing single lines one at a time. This failed to fulfil the original goal of the project which aimed to make use of strategies that involve accumulating large scores. In addition to this, we can look at [12] who struggled to effectively implement a CNN architecture and saw unsatisfactory results with from their models, further supporting the use of alternative architectures.

We instead utilise a linear fully connected network architecture which is generally defined as having a single or multiple hidden layers in between the input and output layers. Similarly to CNNs, the more hidden layers that this network has, the more complex the connections the network will be able to make. As shown in Figure 5, this project's network architecture has 18 input nodes (defined in Section 4.5), 4 hidden layers each containing 2048 nodes which finally relay to the 33 output nodes, these map to the 33 high level actions that the agent has access to (defined in Section 4.6). The input layer and all hidden layers have rectified linear unit activation functions which add non-linearity to the network, allowing it to learn more complex patterns from the environment observations. The necessity for a large neural network comes from the increased state space size that stems from observing many additional features which are covered in Section 4.5.

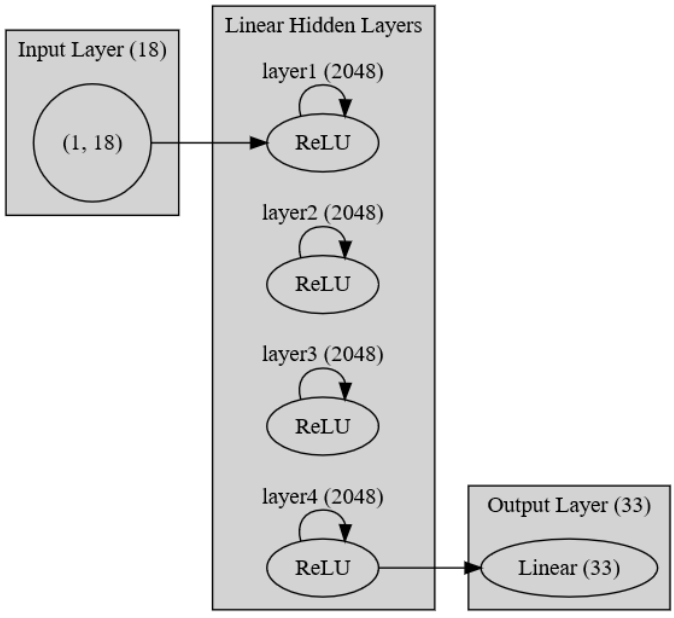

Figure 5: A visualisation of the linear fully connected architecture used for Q-value approximation.

4.5 State Space

4.5.1 Maximum Column Heights

Projects such as [7] provide high level state values for the agent to analyse as opposed to allowing full observation of the game board. This approach massively reduces the state-space and time it takes for the agent to make initial observations when exploring.

As discussed in [3], a full observation of the board would produce too large of a state-space for any linear model to have to consider, with the maximum number of states on a 20 × 10 Tetris grid being:

$$2^{200} \approx 1.61 \times 10^{60} \tag{3}$$

However, this observation can be greatly simplified if we assume that the agent playing is producing no gaps in the stack. In this case, we can transform the observation into an array of the maximum heights of each column on the grid as shown in Figure 6. Additionally, the right most column, where the agent will be dropping the 'I' piece down to perform tetris line clears, can be removed from the observation, as the agent should always view this column as being empty when stacking in the other 9 columns on the left of the board. This feature reduction brings to the total number of states to:

$$20^9 \approx 5.12 \times 10^{11} \tag{4}$$

This value can be further reduced by constraining the agent to only stack at a maximum of 5 pieces above the minimum piece height as shown in Figure 7. This encourages a strategy that stacks pieces flatter and lower down and further decreases the number of possible board states to consider to:

$$5^9 \approx 1.95 \times 10^6 \tag{5}$$

4.5.2 Additional Observations

In addition to observing the maximum height of each column on the board, the agent also observes the following data about the current game state:

- *Tetris ready*. This is a value set to either 0 or 1 that indicates to the agent if it currently has a board setup for scoring a Tetris line clear.
- *Holds performed*. This value tracks how many times the agent has performed the hold action in a row, allowing it to learn that performing the hold action multiple times in succession has no effect on the observable space, as mentioned in Section 3.2.
- *Held piece*. This value holds the piece identifier for whichever piece is currently being stored in the hold position.

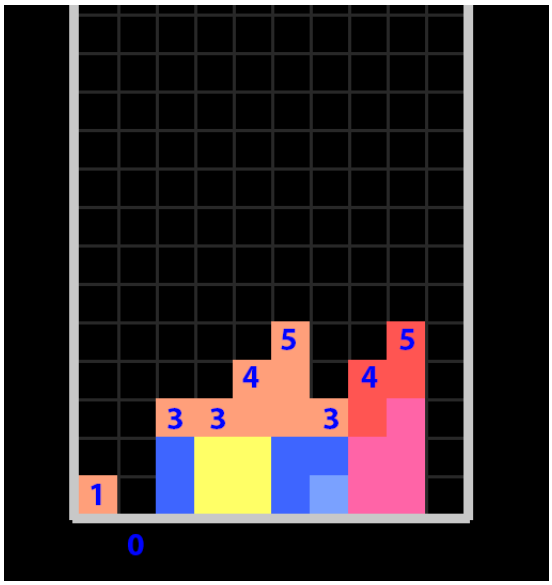

Figure 6: A labelled representation of how the heights in each column are calculated when the lowest point of any piece is at the bottom of the board, where the maximum variation in height is 5. Observation heights for each column in this example would be [1, 0, 3, 3, 4, 5, 3, 4, 5].

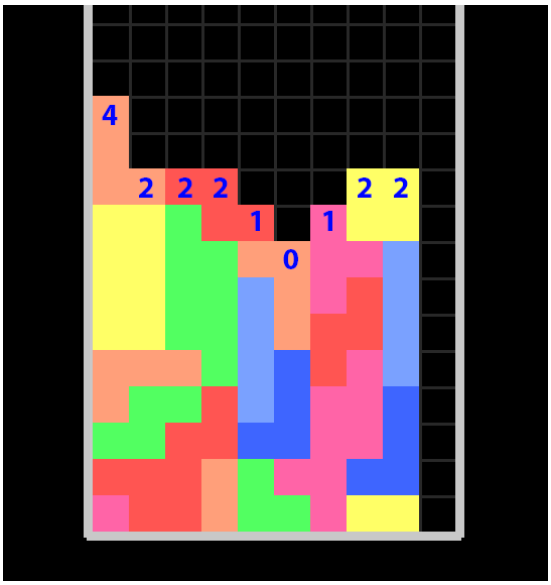

Figure 7: A labelled representation of how the heights in each column are calculated when the lowest point in the stack is above the bottom of the board. Observation heights for each column in this example would be [4, 2, 2, 2, 1, 0, 1, 2, 2].

- *Current piece*. This value holds the piece identifier for the current piece being moved and placed onto the board.
- *Piece queue*. This is the array of piece identifiers relating to the next 5 pieces in the queue. This observation is essential for giving the agent enough foresight to stack safely and efficiently.

While this increases the number of states, deep Q-learning utilises its neural network to relate undiscovered states to already explored ones. This allows it to effectively reduce the quantity of states that need to be initially considered when the agent is exploring. Thus, in its most basic form, the observation space can be seen as the maximum height of each column and the current piece. This brings the number of states that the agent must consider as entirely unique states to the following:

$$5^9 \times 7 \approx 1.37 \times 10^7 \tag{6}$$

4.6 Action Space

Typically, there are two main ways of defining the action space for an agent. The first is to allow the agent full access to all available actions, which for the game of Tetris, would be: horizontal movement (left and right), piece rotation (clockwise and anti-clockwise), piece holding

and piece placing (soft drop and hard drop). After experimenting with allowing the agent complete freedom to choose its next action, it quickly became apparent that because only the hard drop action, out of the 7 possible actions that could be taken, were placing the piece onto the board, the rate at which the agent received rewards was too sparse and therefore never made any significant progress when learning which sets of actions were desirable. This led the project to explore the other primary method of specifying the agents action space which was to pre-define unique combinations of actions that were most likely to result in a favourable state and instead allow it to select its next action from those.

4.7 Rewards System

The rewards system encourages the agent to keep the maximum and minimum height of the stack as close together as possible when placing pieces by rewarding it more significantly for placing pieces closer to the bottom of the stack. This is defined as the $Height$ value in the equation below.

The agent is rewarded if the bumpiness of the stack was equal to or less than the bumpiness of the stack in the previous game state. The bumpiness value is calculated by checking the adjacent heights on both sides of every column and summing them together. This is defined as the *Bumpiness* value in the equation below.

As this project was focused on the agent being able to stack with the purpose of performing tetris line clears, the agent is only rewarded when it clears exactly 4 lines, as not to encourage it to repeatedly clear single lines at a time for its short-term reward benefits. This is defined as the *Lines* value in the equation below.

$$Reward = Height + Lines + Bumpiness \tag{7}$$

Where:

$$Height = 20 - (maxheight - minheight) \tag{8}$$

$$Lines = \begin{cases} 100 & \text{if } cleared == 4 \\ 0 & otherwise \end{cases} \tag{9}$$

$$Bumpiness = \begin{cases} 10 & \text{if } bumpiness_{t-1} >= bumpiness_t \\ -10 & otherwise \end{cases} \tag{10}$$

5 Experiments

5.1 Resources

All the models that were produced throughout the duration of the project were trained on a Nvidia RTX 3070ti in combination with 32GB of 2133MHz DDR4 RAM which was required for storing the experience replay buffer.

The nature of DQN models is that training can take a long time to converge due to being a highly sample inefficient algorithm [8]. When training DQN models, convergence can take up to 100 hours. This makes it difficult for the project to make a complete comparison across all hyper parameters and network architectures.

Models were regularly saved throughout training to allow for sufficient progress monitoring. This resulted in 125GB of model files being produced. The large volume of models was a result of hyper parameter experimentation and reward system fine tuning, where each model was approximately 200MB in size due to the complex associated neural network.

5.2 Hyper Parameter Optimisation

This process was performed manually as using optimisation algorithms was unfeasible due to the previously mentioned model training times in Section 5.1 and the overall time constraints of the project.

5.2.1 Learning Rate

Hyper parameter experimentation started with comparing learning rates and analysing how they affected the agent's performance over a short training period. The learning rate is a fundamental parameter in any reinforcement learning project, as it defines how quickly the model updates its network weightings and therefore how much it's able to learn before converging. The initial hypothesis before testing was that a low learning rate would be a logical fit for DQN due to the model needing an adequate amount of time to converge on a suitable set of network weightings.

We can confirm the validity of this hypothesis when looking at Figure 8, where we can see that a low learning rate of 5e-4 yielded the longest game duration over a short training period. Interestingly, we can see a massive drop off in mean duration when decreasing the learning rate lower to a value of 1e-5. This drop off could be explained by considering that if the learning rate is too low, the agent is never able to make any connections between its chosen actions and their rewards.

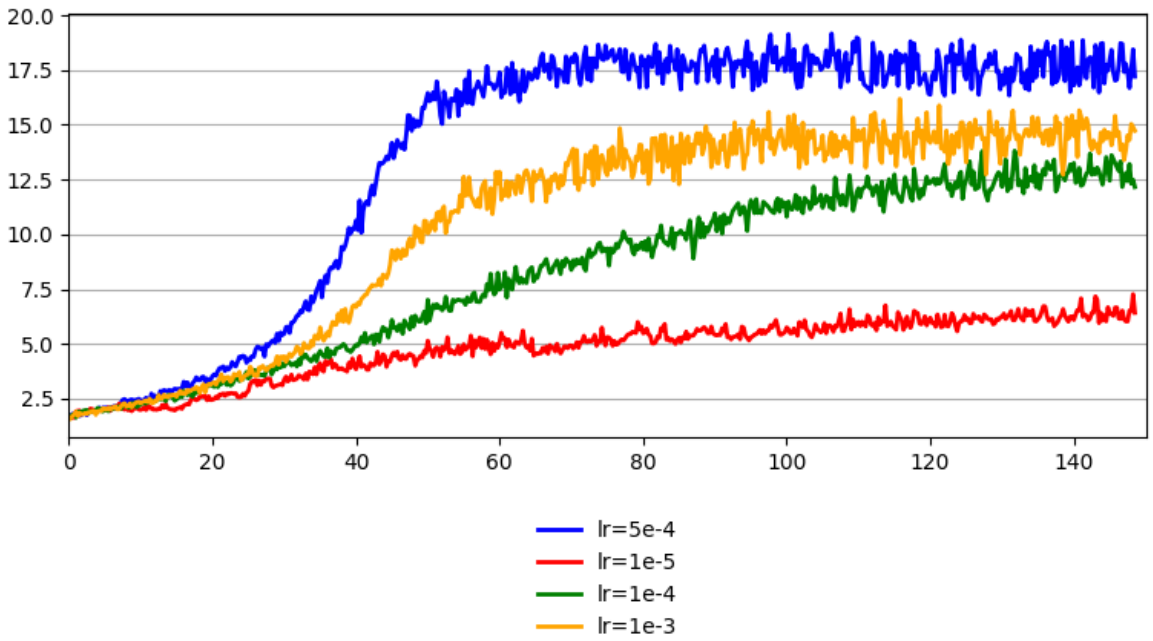

Figure 8: Comparing the mean game duration of different learning rates over 140 epochs of training.

5.2.2 Epsilon Decay Rate

Testing started with a low epsilon decay rate of 4e-6 which, as explained in Section 4.2, will result in a longer exploration period. We can see in Figure 9 that by continuing to drop this the decay rate, the exploration time increased exponentially, which greatly increased the agent's performance.

The lowest decay rate tested was 2e-6. In theory, decreasing the decay rate should keep increasing the agent's performance. However, a decay rate of 2e-6 already equates to approximately 300 epochs of mostly exploration. Therefore, further decreasing this value would result in an unreasonable length of training time for a project that already had significant time constraints.

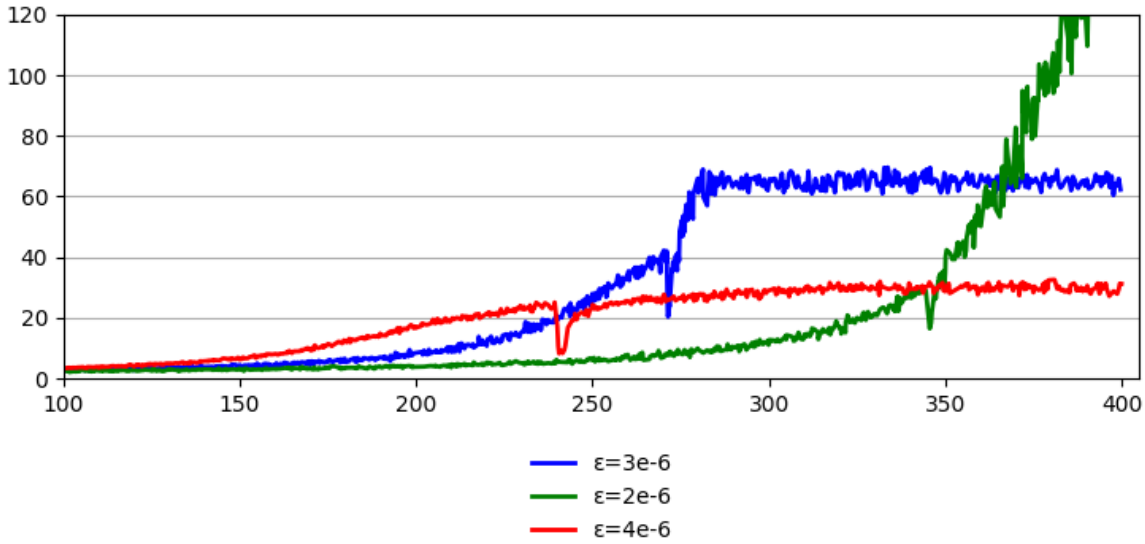

Figure 9: Comparing the mean game duration of different decay rates over 400 epochs of training.

5.2.3 Replay Memory Buffer

The point on each line in Figure 10 where the mean reward starts increasing is the moment when the agent learns how to perform tetris line clears. We can see that with an insufficiently sized replay buffer, such as 1e5, the agent fails to discover how to tetris line clear and is never able to produce any significant reward during training. Conversely, we can see that when the agent's replay buffer is increased to 1e6, it learns to effectively make use of tetris line clears to help increase its mean reward. This is again exaggerated when we increase the replay memory to 5e6, where we can see the agent using tetris line clears more frequently and efficiently, allowing it to survive for longer and therefore increase the mean reward.

Similarly to Section 5.2.2, Figure 10 shows that increasing the replay buffer would continue to increase the agent's performance. This behaviour could be explained by considering that because the state space doesn't allow the agent to observe any gaps in the stack, every state that the agent discovers can be considered a valuable state. Therefore, we can conclude that given this particular setup, the replay buffer should be large enough to cover all the discoverable transitions in the state-space. Hence, 5e6 was chosen as the final value going forward, as it's approximately

the number of states that the agent observes throughout its 400 epochs of training. Additionally, it then becomes trivial to conclude that it would be inconsequential for the replay buffer to be larger than the total number of discoverable transitions because the replay buffer will never be filled resulting in wasted storage space.

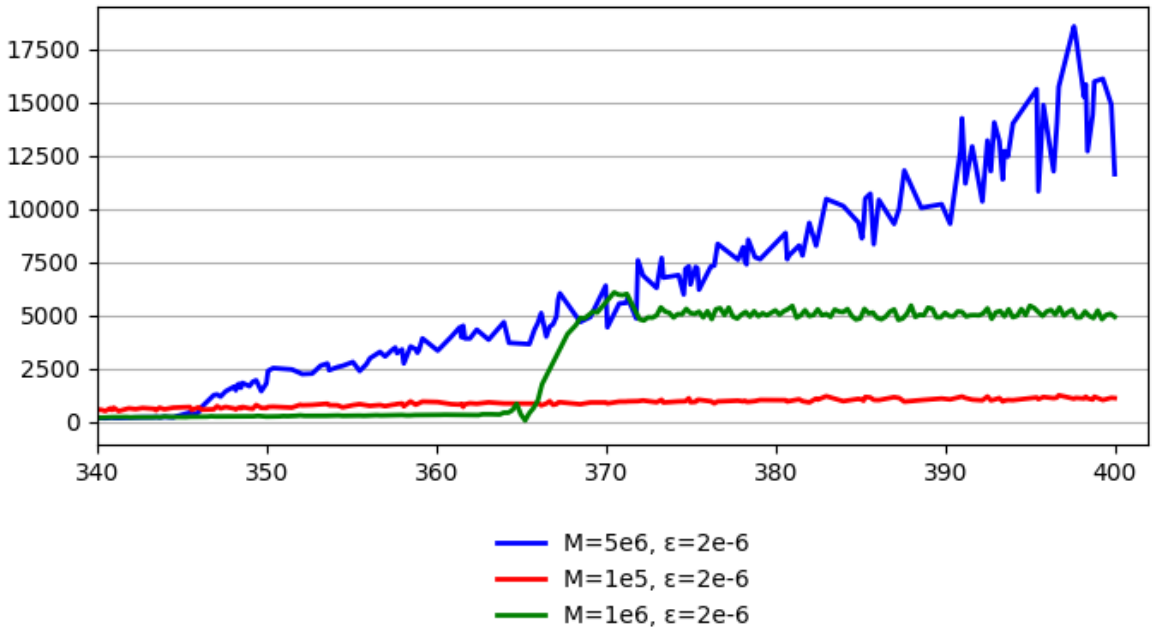

Figure 10: Comparing the mean reward of different replay buffer sizes over 400 epochs of training with an epsilon decay value of 2e-6.

6 Results

6.1 Assessment Criteria

To evaluate an agent's performance, we can compare the mean and best values for the following metrics: pieces dropped, total score, percentage of mistakes, lines cleared and the maximum B2B combo achieved. For this evaluation, a mistake is defined as an action that increases the number of gaps present on the board. The total score will be based on the standard Tetris guideline scoring system [4] as seen in Table 1.

6.2 Performance

6.2.1 Random Actions

To set a clear benchmark for the metrics the results will be tracking, we can look at the performance of an agent taking random actions, as seen in Table 2. This agent is only able to drop 14 pieces on average before the game ends and is unable to clear any lines over the 5,000 games played. While most models will be able to outperform these values, they help to give context to the results in upcoming sections.

Table 2: Random actions agent results

Metric	Value
Games played	5,000
Mean Pieces dropped	14
Mean Total score	0
Mean Mistakes (%)	76.47
Mean Lines cleared	0
Mean Maximum B2B combo	0
Best Pieces dropped	24
Best Total score	0
Best Lines cleared	0
Best Maximum B2B combo	0

6.2.2 B2B vs B2B without piece burning

We hypothesised that allowing the agent to burn pieces, as seen in Section 3.3.3, was an inefficient strategy that would only reduce the agent's total score. To test this hypothesis, we directly compare the *B2B* strategy with a *B2B without piece burning* strategy.

When looking at Table 3, we can see that the *B2B without piece burning* agent was able to achieve a best B2B combo of 89 compared to the *B2B* agent's best combo of 51. Yet, the *B2B* agent was able to achieve a mean score of 55,821, whereas the *B2B without piece burning* agent was only able to achieve a mean score of 33,280. Adding to this, the results show that the *B2B without*

[4] https://tetris.fandom.com/wiki/Scoring

piece burning agent made a mistake 3.11% of the time when selecting an action, whereas the *B2B* agent made a mistake just 1.61% of the time when selecting an action. Hence, the *B2B* agent made mistakes almost half as often as the *B2B without piece burning* agent, showing the higher level of consistency that the piece burning strategy provides.

In theory, given an agent that is capable of stacking optimally, the piece burning strategy is less efficient than a pure B2B strategy. However, our hypothesis failed to take into account that producing an agent with such capabilities is an exceedingly difficult task for reinforcement learning. Thus, piece burning appears to be a technique that will compliment most agents that both are susceptible to mistakes and need support in managing that associated risk.

Table 3: B2B vs B2B without piece burning results

Metric	Value
Games played	5,000
Mean Pieces dropped	721
Mean Total score	55,821
Mean Mistakes (%)	1.61
Mean Lines cleared	272
Mean Maximum B2B combo	11
Best Pieces dropped	6,235
Best Total score	567,800
Best Lines cleared	2,464
Best Maximum B2B combo	51

Metric	Value
Games played	5,000
Mean Pieces dropped	435
Mean Total score	33,280
Mean Mistakes (%)	3.11
Mean Lines cleared	173
Mean Maximum B2B combo	13
Best Pieces dropped	2,589
Best Total score	221,200
Best Lines cleared	1,027
Best Maximum B2B combo	89

Exact distributions of data across all 5,000 games, for the *B2B* agent, can be found in Appendix A.

6.2.3 Score Efficiency

The area where this project excelled the most was in the agent's scoring efficiency. Table 3 shows us that the *B2B* agent had a mean score per lines cleared of 205. We can compare this performance to [7] which utilised a high-level neural network to achieve a mean score per lines cleared of 160. The model produced in [7] is presented as being able to score efficiently in its own right, yet we can still see a clear gap in performance, exaggerating the proficiency of the *B2B* agent in this field. Another metric supporting this is the percentage of lines clears per pieces dropped. The highest percentage possible, using only Tetris line clears, is 40%. The *B2B* agent was able to achieve a mean percentage of 37.73%, again displaying the high scoring efficiency that the project was able to achieve.

It should be noted that the stacking efficiency that the *B2B* agent is able to achieve can be largely attributed to both the 5 piece queue and the 7 bag system, introduced in Section 3.2. The 5 piece queue massively reduces the risk of stacking for Tetris line clears. This is because, if the agent knows ahead of time how it's going to place future pieces, it will be far less likely to stack in such a way that leaves it without any optimal piece placements. In conjunction with this, the 7

bag system guarantees that no piece is ever more than 13 pieces away, as explained in Section 3.2. Hence, the agent will rarely ever be without an 'I' piece, making playing for B2B combos far more viable.

6.2.4 Total Line Clears

While the project was able to achieve more than satisfactory results in regard to score efficiency, it struggled to perform desirably when considering the total number of line clears across the duration of an entire game. The highest number of lines cleared for the *B2B* agent was 6,235. When we compare this with papers such as [11], which achieved approximately 35,000,000 line clears, we can see that the *B2B* agent massively under performs in this area.

6.3 Behaviours

This sections briefly covers the types of behaviours that we can observe when the *B2B* agent is playing.

6.3.1 Piece Holding

An interesting way that the *B2B* agent chose to utilise the piece holding mechanic was to frequently hold onto the 'I' piece, as seen in Figure 11. The most likely reason why the agent does this is because it wants to reduce the time between Tetris line clears, which it views as the most valuable action due to its large reward value. If the agent is in a Tetris ready state, then always having an 'I' piece available to use means the agent doesn't have to wait for one to appear in the queue before it can perform the Tetris line clear.

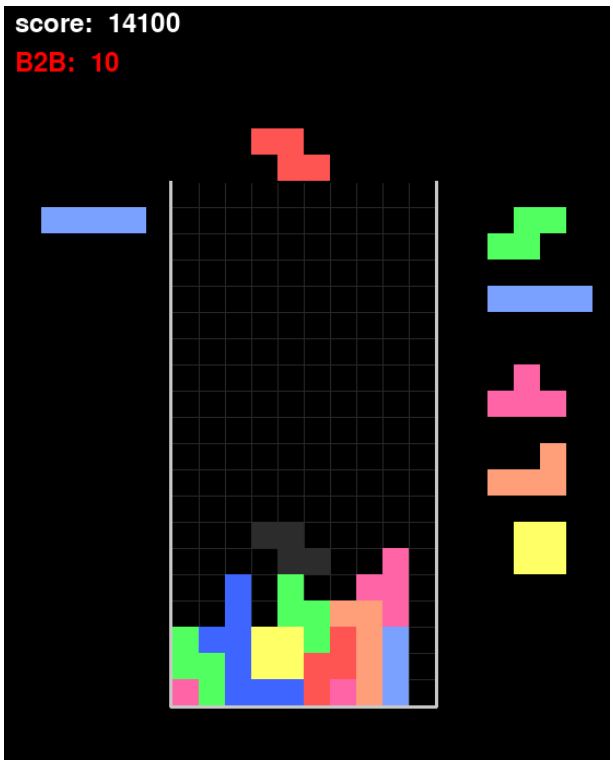

Figure 11: Example of the agent holding onto the 'I' piece as it's preparing for an upcoming Tetris line clear.

6.3.2 Stack Patterns

One of the stacking patterns that the agent regularly made use of is stacking groups of pieces together to form a repeating pattern within the stack. This is a technique commonly used among human players to help them keep their stack organised. Playing this way makes it easier to quickly decide where to place future pieces into the already defined pattern. The *B2B* agent can be seen utilising a simple stacking pattern, as described in Figure 12.

Figure 12: Figure of the agent placing linear pieces such as the 'I', 'L', 'J' and 'O' together on the left and right sides of the stack, and placing the non-linear pieces such as the 'T', 'S' and 'Z' pieces together in the centre of the stack.

7 Conclusion

This project was successful in producing an agent that is capable of advanced Tetris strategy through the utilisation of modern Tetris features. The results showed that the agent could play with great scoring efficiency and could, in some cases, even outclass models from other papers in terms of score per lines cleared. This supported the hypothesis that clearing multiple lines is a stronger strategy than clearing single lines at a time, as the agent was able to achieve greater scores during the beginning of a game. However, without a more consistent agent that makes less mistakes over the course of a game, the results show that single line clears are a far safer strategy and produce more capable models across longer games.

Throughout the project, adjustments were constantly made to balance the long term and short term rewards that the agent was aiming for. This combined with the uncertainty problems that Tetris presents, shows that machine learning Tetris projects could have potential real world applications to risk management problems such as prioritising cyber security threats and assessing risks in financial sectors. Puzzle games, like Tetris, also offer a unique perspective on problem solving and could be used as an educational tool to offer a more enjoyable alternative to understanding computer science concepts.

While the results from Section 6.2.4 were initially discouraging, the *B2B* agent still clearly has a lot of room for improvement. Future work on improving the agent's performance could involve further optimising the feature set of the state-space. This could be achieved by decreasing the size of the piece queue that the agent observes. Any future work on improving performance would benefit from frequent access to high end hardware and the ability to train the agents for a longer period of time than the project's original time constraints.

References

[1] Kai Arulkumaran et al. 'Deep Reinforcement Learning: A Brief Survey'. In: *IEEE Signal Processing Magazine* 34.6 (2017), pp. 26–38. DOI: 10.1109/MSP.2017.2743240.

[2] Ron Breukelaar et al. 'Tetris is hard, even to approximate'. In: *Int. J. Comput. Geom. Appl.* 14.1-2 (2004), pp. 41–68. DOI: 10.1142/S0218195904001354. URL: https://doi.org/10.1142/S0218195904001354.

[3] Donald Carr. 'Applying reinforcement learning to Tetris'. In: 2005.

[4] Kurt Driessens and Sašo Džeroski. 'Integrating guidance into relational reinforcement learning'. In: *Machine Learning* 57 (2004), pp. 271–304.

[5] Tanmoy Hazra and Kushal Anjaria. 'Applications of game theory in deep learning: a survey'. In: *Multimedia Tools and Applications* 81.6 (2022), pp. 8963–8994.

[6] Ian J. Lewis and Sebastian L. Beswick. 'Generalisation over Details: The Unsuitability of Supervised Backpropagation Networks for Tetris'. In: *Adv. Artif. Neural Syst.* 2015 (2015), 157983:1–157983:8. DOI: 10.1155/2015/157983. URL: https://doi.org/10.1155/2015/157983.

[7] Nicholas Lundgaard and Brian McKee. 'Reinforcement learning and neural networks for tetris'. In: *Technical report, Technical Report, University of Oklahoma* (2006).

[8] Volodymyr Mnih et al. 'Playing atari with deep reinforcement learning'. In: *arXiv preprint arXiv:1312.5602* (2013).

[9] Ashvin Nair et al. 'Overcoming Exploration in Reinforcement Learning with Demonstrations'. In: *2018 IEEE International Conference on Robotics and Automation, ICRA 2018, Brisbane, Australia, May 21-25, 2018*. IEEE, 2018, pp. 6292–6299. DOI: 10.1109/ICRA.2018.8463162. URL: https://doi.org/10.1109/ICRA.2018.8463162.

[10] Pytorch. *Reinforcement Q-Learning Example*. URL: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html?highlight=dqn.

[11] Bruno Scherrer et al. 'Approximate modified policy iteration and its application to the game of Tetris.' In: *J. Mach. Learn. Res.* 16.49 (2015), pp. 1629–1676.

[12] Matt Stevens and Sabeek Pradhan. 'Playing tetris with deep reinforcement learning'. In: *Convolutional Neural Networks for Visual Recognition CS23, Stanford Univ., Stanford, CA, USA, Tech. Rep* (2016).

[13] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[14] Shangtong Zhang and Richard S. Sutton. 'A Deeper Look at Experience Replay'. In: *CoRR* abs/1712.01275 (2017). arXiv: 1712.01275. URL: http://arxiv.org/abs/1712.01275.

Appendices

A Result Distributions

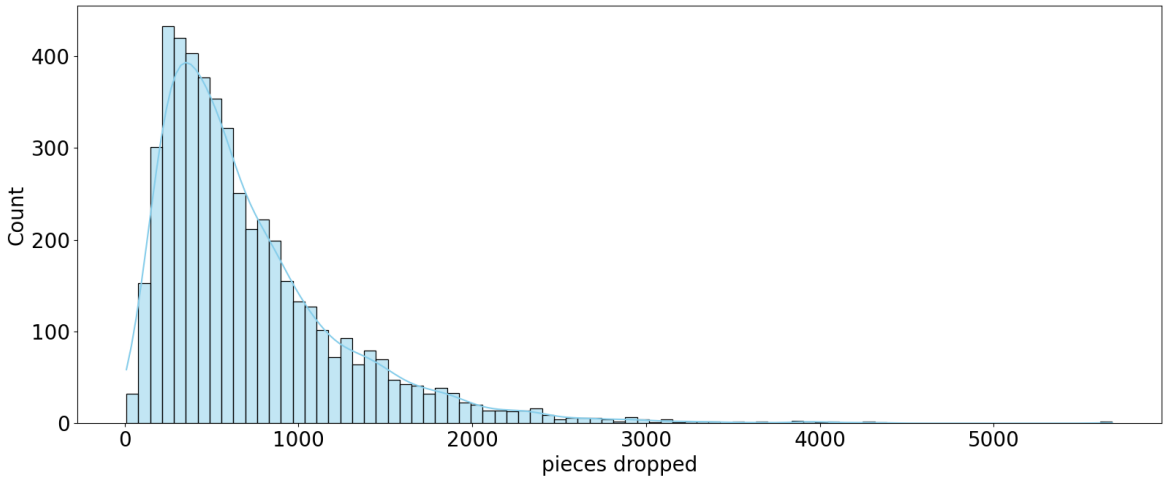

Figure 13: The distribution of the number of pieces dropped across the 5,000 games played when evaluating the *B2B* agents performance.

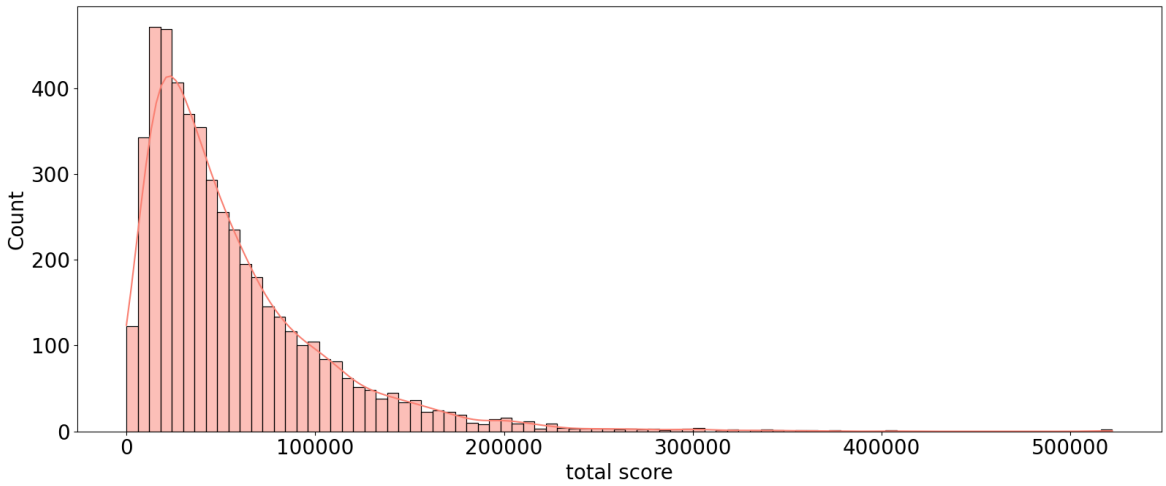

Figure 14: The distribution of the total score across the 5,000 games played when evaluating the *B2B* agents performance..

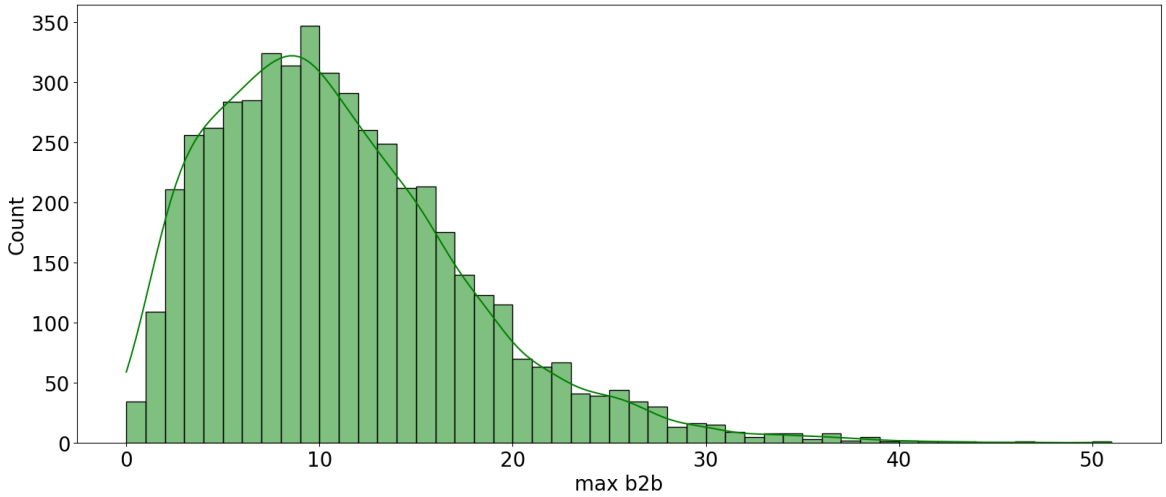

Figure 15: The distribution of the maximum B2B combo achieved across the 5,000 games played when evaluating the *B2B* agents performance..

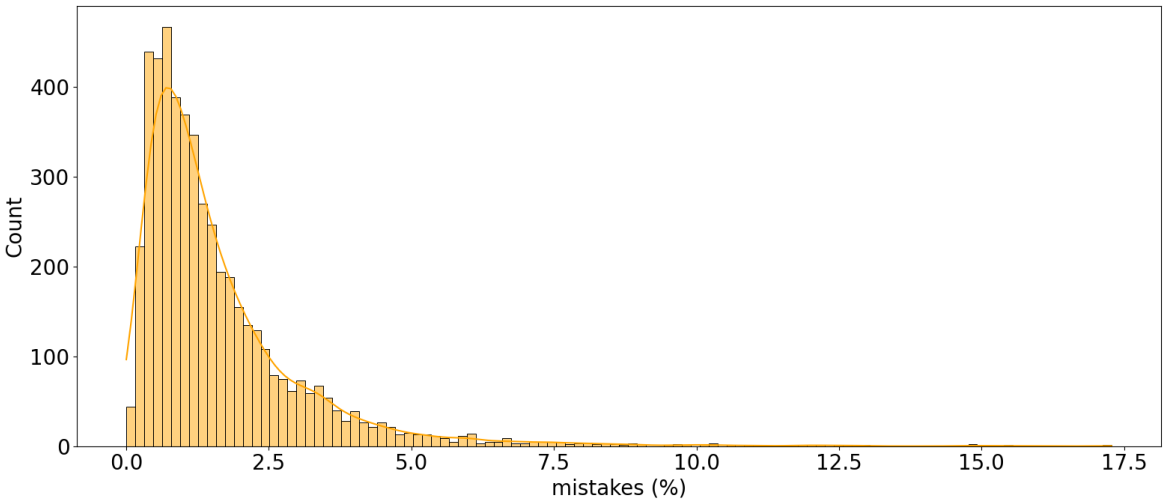

Figure 16: The distribution of the percentage of mistakes made across the 5,000 games played when evaluating the *B2B* agents performance..

B Stacks with Gaps Examples

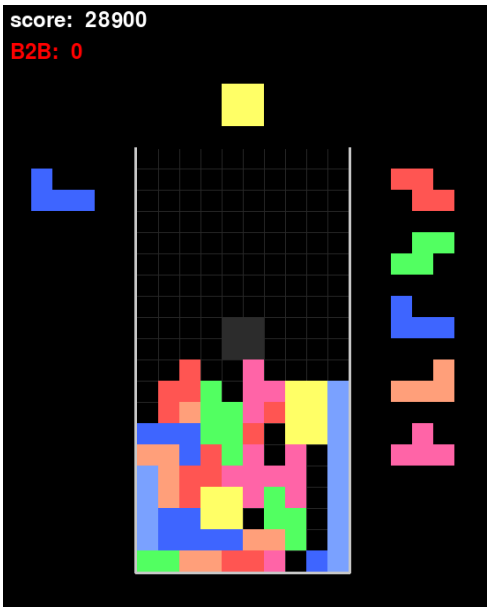

Figure 17: Example of stack where gaps have been created low down on the stack in a situation that is easily recoverable.

Figure 18: Example of stack where gaps have been created higher up the stack in a situation that is harder to recover from.