Preliminary Exploration of the Application of Reinforcement Learning in Tetris

Richard Noh and Franke Tang

May 4, 2024

Abstract

This report details the development and evaluation of two reinforcement learning agents designed for playing Tetris: a Linear Sarsa(λ) agent and a Deep Q-Learning (DQN) agent. We attempted to compare these agents compared against a non-learning baseline agent whose weights were determined and fixed with cross-entropy optimization. Our methodology centered on employing different state and action space abstractions and features derived from prior research. The DQN agent demonstrated a capacity to develop a strategy surpassing that of random gameplay, proving the potential of complex model architectures in game environments that demand both strategic planning and reactive adaptation. However, our work is severely limited by the absence of extensive hyperparameter optimization and a lack of training and testing runs on different random seeds, which drastically hinders the statistical significance of the results.

YouTube Link: https://youtu.be/uGZMDKj8jJc

GitHub Repository: https://github.com/RichardNooooh/TetrisRL

1 Introduction

Tetris, a timeless puzzle game, presents a unique challenge that extends beyond entertainment into the realm of computational intelligence. The objective in Tetris is straightforward: players must align falling tetriminos to complete horizontal lines, which then disappear to free up space and score points. The complexity of the base game increases as the pieces fall faster, and the game ends when the pieces stack up to the top of the playing area. However, in our project, we remove this falling time dependence, and focus on the simpler case of place these pieces strategically to survive as many placements as possible. Despite the simplification, this dynamic environment makes Tetris an excellent testbed for various artificial intelligence (AI) strategies, particularly reinforcement learning (RL), where agents learn to make decisions by interacting with an environment to maximize cumulative rewards.

The application of RL to Tetris involves training agents to understand and manipulate the game's intricacies, aiming to maximize line clears while minimizing the risk of topping out. Traditional approaches have utilized a variety of algorithms, ranging from basic heuristic methods to advanced deep learning models. Each technique offers different insights into the problem-solving capabilities of AI in a constrained and progressively challenging environment.

In this report, we attempt to explore the development and performance of two RL agents—a Linear Sarsa(λ) agent and a Deep Q-Network (DQN) agent—trained to play Tetris. These agents were benchmarked against a non-learning baseline agent designed using a well-established feature set from the Building Controllers for Tetris (BCTS) paper [18]. However due to computational limitations, we were unable to truly compare these methods with each other in a precise "apples-to-apples" way. Additionally, we describe the design of our custom Tetris environment, which was essential due to the limitations and challenges associated with some of the existing implementations. Despite facing computational and methodological limitations, this project aims to shed light on the potential and challenges of applying RL to the classic game of Tetris.

2 Related Work

Previous research on Tetris playing agents has explored various methodologies and techniques. Algorta and Şimşek [1] conducted a historical survey of machine learning approaches for Tetris up to 2019, discussing methods ranging from simplified models to genetic algorithms and policy iteration. Thiery and Scherrer [18] provided a comprehensive review of feature design for Tetris controllers known as BCTS. They also provide a comprehensive list of features used and discusses some common variations in the Tetris problem. Their features weights were optimized using a technique called cross entropy.

Simsek and Yu [23] presented a method for filtering potential actions in sequential decision problems like Tetris, highlighting the convergence of multiple neural networks under certain conditions, which enables agents to make

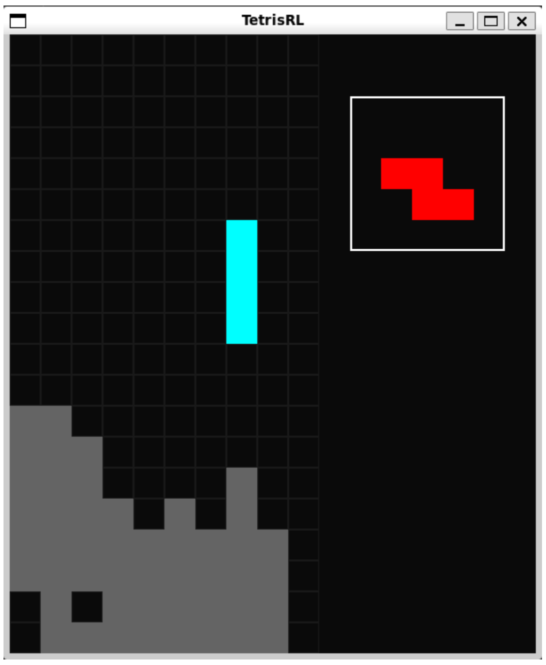


Figure 1: GUI of Tetris Implementation. The grey tiles are the filled in tiles from previous pieces, while the colored falling block is the "current piece". This state was obtained with manual playing. Note that while the tetrimino shown on the right-hand box is the "next piece", we did not end up utilizing that information in any of our experiments. The GUI was implemented with PyGame - independent of the internal Tetris model environment.

effective decision with simplified actions. They define three conditions which allow agents to make good choices without a detailed evaluation function: 1) simple dominance, 2) cumulative dominance, and 3) noncompensation.

Other algorithms have been tailored for Tetris Gameplay such as Classification-Based Modified Policy Iteration (CBMPI) and Surprise Minimizing Reinforcement Learning (SMiRL) [2, 3]. CBMPI is a type of rollout method on a linear function approximator for approximate dynamic programming. SMiRL is a general reinforcement learning algorithm that attempts to reduce the entropy of the states the agent visits.

Steven [15] investigated the use of convolutional neural networks (CNN) for Tetris gameplay, implementing a Q-learning algorithm with an epsilon-greedy policy. This study considered both simplified and raw button press action spaces. Furthermore, research by various authors explored alternative approaches such as graph neural networks [4] and systemic n-tuple networks [6]. The use of Non-Euclidean space with GNN allow for a quite robust model though hard to replicate. The use of n-tuple networks was used on a modified version of Tetris that only contained S and Z tetriminos. This heavily reduced the number of states 2^n.

Additionally, experiments involving real-world applications, such as a Tetris playing robot arm [21], have been conducted, showcasing the versatility of Tetris as a testbed for reinforcement learning. The authors used a modified game of Tetris so that robot can play with 35 real-life tetriminos. The paper used a Deep Q Network with Proximal Policy Optimization.

3 Problem Formulation

3.1 Tetris Library

For this project, we initially wanted to use the `gym-tetris` Python package, which provided an OpenAI gym interface with an NES Tetris emulator [7]. In order to compare our agents with some of the work with the literature, we spent time implementing an additional interface which would allow us to incorporate these "group actions" (see Section 3.3). While we initially thought that this interface would be trivial to interact with, the emulator and Gym interface were much more difficult to interact with. Even asking the Gym interface to move the agent left or right two spaces was difficult due to how the emulator treats actions as hardware button presses and Tetris features such as "auto delay shift" [9]. Unfortunately, we discovered these difficult complications weeks into the project, and abandoned this Gym library.

Many of the other available Tetris implementations available on GitHub and cited by some of the papers [20] seemed to have suffered from poorly documented code, heavy dependence on timing events, and deeply coupled GUI systems. Rather than spending significant time dealing with these libraries or learning about their hidden features, we decided that it was more efficient for us to create our own Tetris environment, heavily utilizing our existing interface from the previous library. A screenshot of the GUI is shown in Figure 1.

Due to our this technical challenge, we unfortunately did not have enough time to obtain more trials of our agents to obtain good data, training on different seeds nor able to obtain good hyperparameters.

3.2 State and Action Spaces

In Tetris, a state is composed of the following components: the play field (or grid) and the current tetrimino. The grid is an array of tiles 20 units high and 10 units wide, containing both empty and filled tiles. The current tetrimino is defined by 3 variables: center position, type, and orientation. Each piece is located on the grid, which is initially centered at the top, middle tile. They can be of 7 different types: the "T", "O", "J", "L", "S", "Z", and infamous "I". Depending on the type, of remaining three "piece tiles" are positioned on the board. Additionally, these pieces are defined by up to 4 possible orientations, which each place the 3 piece tiles relative to the center tile, differently. We used the NES Tetris orientation scheme, described in [10].

$$\mathcal{S} = \{\texttt{grid\_configuration}, \texttt{piece\_position}, \texttt{piece\_type}, \texttt{piece\_orientation}\} \tag{1}$$

The action space is the following 5 controls for the current tetrimino, shown in equation 2. If the action is possible, the piece will deterministically move from its initial state to the next state according to that action. Note that we do not have an external clock system that forces the piece to fall down over time, nor do we have a rotation "wall kick" feature that is common in modern Tetris implementations [17].

$$\mathcal{A} = \{\texttt{move\_left}, \texttt{move\_right}, \texttt{rotate\_clockwise}, \texttt{rotate\_counter\_clockwise}, \texttt{soft\_drop}\} \tag{2}$$

Tiles on the grid are filled when a piece "lands" on the bottom of the grid or on top of another piece. Specifically, the grid is modified when an agent attempts a `soft_drop` action and fails, which means that the falling piece has officially "landed". This allows pieces to perform "sliding" or "T-Spin"-like actions that would otherwise be impossible. The environment fills in the corresponding tiles on the board, and prevents other pieces from occupying those tiles. If a row, or multiple rows (up to 4), on the board are filled when the piece lands, then those filled lines are cleared from the board and all tiles above them fall by those number of lines. Then the game randomly generates a new piece at the top of the board. The episode is over when the game cannot place a new tetrimino at the top of the board.

With the random generation of the new piece, Tetris is a stochastic Markov Decision Problem where the goal is to clear as many rows of tiles (lines) as possible. Mathematically, Tetris is essentially a episodic task, as "almost all Tetris games must end" [19]. However, but with sufficiently skilled agents, Tetris are essentially continuing tasks with an indefinite horizon.

3.3 Action Space Abstraction: Grouped Actions

In most of the literature that we could find, the action space, $\mathcal{A}$, is not often used. Most of the difficulty in Tetris, particularly for human players, is the placement of the falling tetrimino and not how to move it. For our project, we utilized a wider action set that [15] referred to as "grouped actions", $\mathcal{A}_g$.

Although we do not present it in this paper, we had significant difficulty getting our agents to converge to a good policy with $\mathcal{A}$ with our current computational resources. The control complexity from $\mathcal{A}$ requires that it not only learn how to maneuver the falling tetrimino, but also how to strategically place them to clear lines. As such, the grouped action set is defined by considering all possible unique landing positions of the current tetrimino. These landing positions are determined by the breadth-first search algorithm, traversing through the entire grid space with $\mathcal{A}$. We define the path of actions from the starting position to a landing position as the grouped action, and the collection of grouped actions to each unique landing position as $\mathcal{A}_g$.

With this new $\mathcal{A}_g$ action set, we can redefine the notion of a state. From the perspective of the agent, the state can be viewed as the current configuration of the grid and the current type of the tetrimino. While the orientation and position of the tetrimino is important, this information is not directly handled by the agent. Rather, from a given state, $s \in \{\texttt{grid\_configuration}, \texttt{piece\_type}\}$, the agent sees a set of possible actions, $a \in \mathcal{A}_g(s)$ that would lead it to a new grid configuration, and randomly, a new piece type: s'. We will refer to this new state space as $\mathcal{S}_g$. This redefined state space significantly helps our understanding of Section 4.3.

3.4 State Space Abstraction: Features

When considering the size of $\mathcal{S}$ or $\mathcal{S}_g$, the number of grid configurations, alone, is on the order of 2^{200}. The sheer size of the state space of Tetris fundamentally necessitates the use of some abstraction and function approximation. The main concepts used for this state abstraction are *holes* and *wells*. Holes are defined as empty cells within a column that has one or more filled cells above it. Wells are defined as empty cells that have filled neighbors in the immediate right and left but has no filled cells above. These two features are generally undesirable, because holes prevents line clearing and wells often result in suboptimal placements of pieces.

We utilized the features from [18], which are listed below. We will refer to the vector of these features as $\mathbf{x}(s, a)$ for all $s \in \mathcal{S}_g$, $a \in \mathcal{A}_g$.

- Landing height: The height of the current piece when it would be placed. We consider the positions of the lowest and highest tiles of the current landed piece and average them.
- Row transitions: The sum of transitions of each row. Similar to the column transitions, except we assume both left and right walls to be filled.
- Column transitions: The sum of transitions of each column. We counts the transitions from a filled cell to an empty cells or the reverse. We assume that the top of the grid is considered empty and the bottom of the grid is considered filled.
- Number of holes: The total number of holes in the grid
- Cumulative wells: The sum of each cumulative wells. For each well, we consider the depth and for each successive well cell we add cumulatively. For example if the depth of a well is 4, the cumulative well would count towards 1 + 2 + 3 + 4 = 10.
- Hole depth: The sum of all filled cells above the holes in the same column
- Rows with holes: The number of rows that contain at least one hole
- Eroded piece cells: The number of lines cleared multiplied by the number of cells the current piece occupied when cleared. For example, a 4 line clear with an "I" piece would be 4 x 4 = 16.

3.5 Reward Function

As stated earlier, the goal of a Tetris agent is to clear as many lines as possible. A related goal is to also survive the game as long as possible. Without strategic placement of the falling tetriminos and line clearing, the grid will quickly fill up and end the episode. However, with a random agent, the likelihood that even a single line gets cleared is approximately one in a thousand games [15]. Any reward signal from line clearing would be incredibly sparse. While there are methods such as prioritized sweeping [16, p. 169] to encourage better and more useful exploration, we decided on the following reward function:

- $+10 \times$ `num_lines_cleared` to encourage line clearing
- -10000 for ending the episode
- $+0.01$ for each action

Since we are utilizing $\mathcal{A}_g$, each action results in a placed tetrimino. To encourage the agent to survive longer, we hoped that this small reward signal leads to a better policy that survives longer. Policies that result in more placed tetriminos per episode will have a higher return. This will also discourage placements that quickly raise the maximum height of the board, since that will naturally result in shorter episodes. With this small signal, we hoped to see that the agents will learn how to clear lines.

4 Methodology

4.1 BCTS Agent

For our experiments, we wanted to first observe how the gold standard of non-learning agents perform in our custom Tetris environment. We chose to implement the feature vector, $\mathbf{x}(s, a)$ for the "Building Controllers for Tetris Systems" (BCTS) agent [18], as described in section 3.4. The evaluation score for this agent is a linear combination of weights and this feature vector: $Q(s, a) = \mathbf{w}^\top \mathbf{x}(s, a)$. We listed the exact weight vector values in the comparison figure, Table 2.

It is important to note that this agent does not perform a learning update. Starting from a state, $s \in \mathcal{S}_g$, the agent evaluates the actions of $a \in \mathcal{A}_g(s)$ and unilaterally chooses the action that maximizes $Q(s, a)$. In other words, for each step in the episode, t, the agent takes the following action:

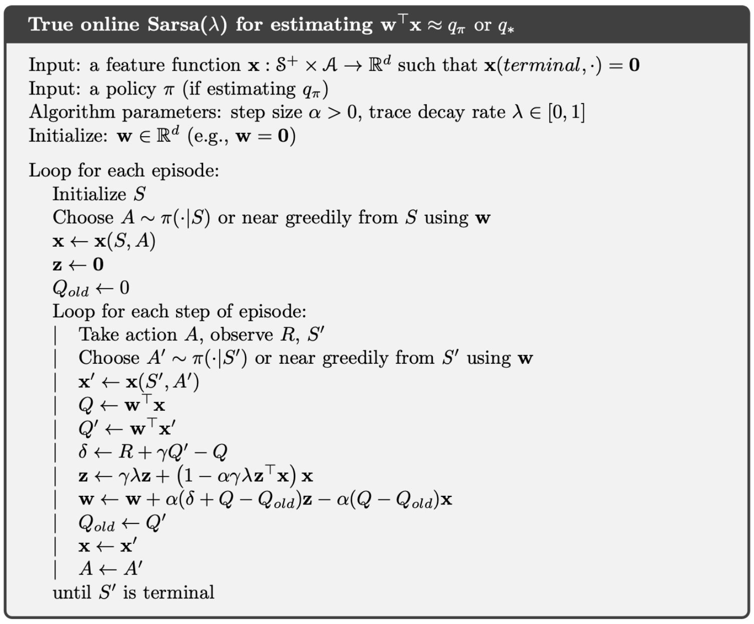
$$a_{t+1} = \arg \max_{a \in \mathcal{A}_g(s_t)} Q(s_t, a) \tag{3}$$

One difference in our implementation compared to the BCTS agent in the literature is that we perform a one-step update - only considering the Q values of the current falling tetrimino. Most of the results report the scores of this agent that additionally considers the next falling tetrimino (as shown in Figure 1. However, we decided to not consider this “next piece” for our experiments and analysis.

One thing of note with the $\mathbf{x}(s, a)$ features is that the implementation details is not standardized in the literature, nor are there precise definitions of a *well* or *row transition*. Running this agent allows us to see if this agent still performs as expected with our particular algorithms to compute each component of $\mathbf{x}$.

4.2 Linear Function Approximation - Sarsa(λ) Agent

For our initial, baseline reinforcement learning agent, we are using Sarsa(λ) with the same linear function approximation as the BCTS agent. Specifically, we wanted to see if our reward function with the Sarsa(λ) algorithm will result in a learned policy that performs comparatively similar to the BCTS agent. We follow the structure and implementation of the True-Online Sarsa(λ) we did for a programming assignment. For reference, we attached a screenshot of the algorithm in Figure 2.



```
True online Sarsa($\lambda$) for estimating $\mathbf{w}^\top\mathbf{x} \approx q_\pi$ or $q_*$

Input: a feature function $\mathbf{x} : \mathcal{S}^+ \times \mathcal{A} \to \mathbb{R}^d$ such that $\mathbf{x}(terminal, \cdot) = \mathbf{0}$
Input: a policy $\pi$ (if estimating $q_\pi$)
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0,1]$
Initialize: $\mathbf{w} \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
    Initialize $S$
    Choose $A \sim \pi(\cdot|S)$ or near greedily from $S$ using $\mathbf{w}$
    $\mathbf{x} \leftarrow \mathbf{x}(S, A)$
    $\mathbf{z} \leftarrow \mathbf{0}$
    $Q_{old} \leftarrow 0$
    Loop for each step of episode:
    |   Take action $A$, observe $R$, $S'$
    |   Choose $A' \sim \pi(\cdot|S')$ or near greedily from $S'$ using $\mathbf{w}$
    |   $\mathbf{x}' \leftarrow \mathbf{x}(S', A')$
    |   $Q \leftarrow \mathbf{w}^\top\mathbf{x}$
    |   $Q' \leftarrow \mathbf{w}^\top\mathbf{x}'$
    |   $\delta \leftarrow R + \gamma Q' - Q$
    |   $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \left(1 - \alpha\gamma\lambda\mathbf{z}^\top\mathbf{x}\right)\mathbf{x}$
    |   $\mathbf{w} \leftarrow \mathbf{w} + \alpha(\delta + Q - Q_{old})\mathbf{z} - \alpha(Q - Q_{old})\mathbf{x}$
    |   $Q_{old} \leftarrow Q'$
    |   $\mathbf{x} \leftarrow \mathbf{x}'$
    |   $A \leftarrow A'$
    until $S'$ is terminal
```

Figure 2: Pseudo-code of Sarsa(λ) we used for eligibility trace implementation, obtained from [16, p. 307]

In our implementation, we had an issue using the $\mathbf{x}$ feature vector. Applying these features into the Sarsa(λ) algorithm causes the evaluation of $\mathbf{z}^\top\mathbf{x}$ to grow exponentially and diverge. To stop this behaviour, we attempted to restrict the values of these features to be between 0 and 1 by normalizing each of the 8 distinct features with their worst case degenerate values. For example, the worst case for the “Number of Holes” feature is to have the top row be completely filled and all rows beneath it be defined as holes, resulting about 190 for the worst. A real game would never almost never observe a feature vector like this, but this is to ensure our algorithm behaves correctly.

4.3 Deep Q-Learning Agent

To compare with our simpler feature-based agent, we also attempted to implement a deep Q-learning (DQN) agent. Unlike our other agents, which trained on the features described in Section 3.4, ϕ, this agent uses a convolutional neural network with the state directly from $\mathcal{S}_g$ and $\mathcal{A}_g$. We heavily referenced the work by Google DeepMind authors [11, 12] in the development of our agent. Additionally, to run our DQN agent, we utilized a personal RTX 2070 Super GPU with PyTorch’s CUDA interface.

4.3.1 Network Architecture

Our network is a convolutional network with six hidden layers, with four convolutional and two fully connected layers, as shown in Figure 3. The convolutional layers each use a 3×3 kernel with a 25% channel dropout to prevent overfitting issues. Additionally, the edge of the image is padded with 1s on all sides, for every layer. The padding allows the network to treat the sides of the input grid as filled tiles, which is how the edges of the grid behave. In

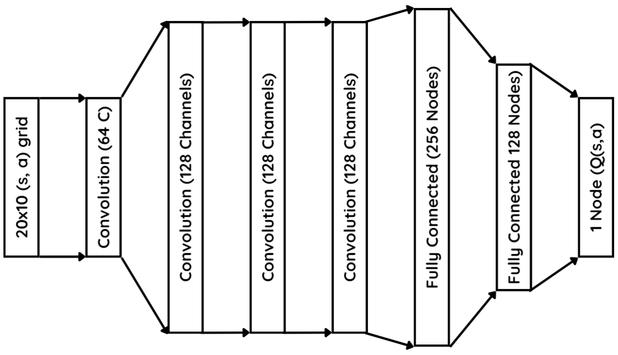

20x10 (s, a) grid
Convolution (64 C)
Convolution (128 Channels)
Convolution (128 Channels)
Convolution (128 Channels)
Fully Connected (256 Nodes)
Fully Connected 128 Nodes)
1 Node (Q(s,a))

Figure 3: Diagram of DQN Architecture. This network has approximately 6.96 million learnable parameters.

hindsight, it may have been better to have to top of the grid input be padded with 0s, instead, since the top is essentially empty.

The two fully connected layers also utilize a 25% weight dropout. All layers utilize the same ReLU activation function. To initialize our network, we also applied He initialization, which allows the network to converge from a better starting point by taking into account the ReLU activation functions in our network [5].

The output layer is a single node, unlike what is described by [12]. Instead of having the deep neural network predict the Q values of all possible actions, instead predict the Q value of the given state-action pair. This is due to the fact that the size of our action set, $\mathcal{A}_g$, is heavily dependent on the current state, since a given state could have a different number of possible landing positions/orientations than another.

4.3.2 State-Action Representation

In the DQN literature, many papers [11,12,15] process the GUI screen and insert the resulting state-images into the neural network. For the sake of computational resources and time, we instead processed the state and action pair into a single 20×10 grid. For each cell in the grid, we represent the state of that cell with a 1 or 0, depending on whether or not that cell is occupied. But while representing the grid configuration fairly straight forward, it is not entirely trivial to encode the action from $a \in \mathcal{A}_g$ into the grid.

Consider the tetrimino that follows the group-action, $a \in \mathcal{A}_g$. This tetrimino will occupy 4 tiles on the grid in a landing position, but does not place itself on the grid until the environment updates the state when the agent acts with a. To differentiate these four tiles from the rest of the board, we decided to represent each of those tiles with the value 1/2. This provides an ordinal relationship between empty, filled, and "about to be filled" tiles, that we hoped would help the agent understand the environment dynamics. We have not encountered another paper that chose to represent the pair $(s, a) \in \mathcal{S}_g \times \mathcal{A}_g$ in this way before.

One alternative of this encoding is to instead use a hybrid mixed-input model that takes in image and vector data, before combining the network connections into a single hybrid network [14]. While there is one non-academic instance that we found where this sort of mixed-input technique was used for Tetris agents [8], we felt that our method of encoding would help directly correlate the tetrimino positions and the overall grid.

4.3.3 Experience Replay Buffer

To help our DQN agent converge to a good policy, we also implemented an experience replay buffer, $\mathcal{D}$. This buffer is implemented as a circular buffer of $N = 1,000,000$ transitions, although none of our experiments ended up reaching this limit. The agent stores each transition that it encounters, then uniformly samples `batch_size` $= 64$ transitions to update its weights.

Each transition is simply the tuple (s, a, r, s'), where r is the reward from taking the action a from state s. In the context of our state and action space abstractions, we are storing actually storing following information:

- (s, a): the grid of 0s, 1/2s, and 1s as previously described
- r: the reward of this transition
- $[(s', a')\, \forall a' \in \mathcal{A}_g(s')]$: the list of all possible next state-action pair grids
- `is_terminal`: the boolean of whether or not this transition terminates at s'.

While the first two bullet points follows fairly straight forward from the standard view of an MDP transition, we require the list of all possible next state-action pair grids to evaluate the off-policy term, $\max_{a'} Q(s', a')$, with our agent. We are caching these grids to reduce the number of breadth-first searches through the grid. The last variable that we store for each transition, is required for the deep Q-learning algorithm that we will describe in the following section.

4.3.4 Algorithm

We present the overview of the deep Q-learning algorithm directly from the Atari DQN papers [11, p. 5] and [12, p. 7], but with our notation of the state and action abstractions and other noteworthy details.

Algorithm 1: Tetris Deep Q-Learning With Experience Replay and Delayed Target Network Update

```
Initialize replay memory $\mathcal{D}$ with capacity $\mathcal{N}$.
Initialize model action value function, $Q$, with He initialization.
Initialize target action value function, $\hat{Q}$ as a copy of $Q$
total_action ← 0
for each episode do
    Initialize $s_1 \in \mathcal{S}_g$;                                          /* env.reset() */
    for $t = 1, T$ do
        total_action ← total_action + 1
        Compute $\mathcal{A}_g(s_t)$ with breadth-first search;
        With probability $\epsilon$ select random action, $a_t \in \mathcal{A}_g(s_t)$;   /* $\epsilon$ - Greedy Action Selection */
        otherwise select $a_t = \arg\max_a Q(s_t, a)$;
        Execute group action $a_t$ in the environment and observe $r_t$ and state $s_{t+1}$;   /* env.step($a_t$) */
        Compute $\mathcal{A}_g(s_{t+1})$ with breadth-first search;
        Store transition $((s_t, a_t), r_t, [(s_{t+1}, \mathcal{A}_g(s_{t+1}))]$, t+1 == T) in $\mathcal{D}$;
        Randomly sample transitions $((s_j, a_j), r_j, [(s_{j+1}, \mathcal{A}_g(s_{j+1}))]$, is_terminal) from $\mathcal{D}$;
        if is_terminal then
            $y_j = r_j$;
        else
            $y_j = r_j + \gamma \max_{a_{j+1} \in \mathcal{A}_g(s_{j+1})} \hat{Q}(s_{j+1}, a_{j+1})$;
        end
        Run gradient descent step on $\mathcal{L} = (y_j - Q(s_j, a_j))^2$;          /* MSELoss() [12] */
        if total_action % C == 0 then
            $\hat{Q}$ ← copy($Q$);
        end
    end
end
```

Algorithm 1 describes the overall process of our Tetris DQN agent. Note that C is the number of actions taken before we update the target network. According to our lecture notes [22], this late update slows down value overestimation with the trade off in convergence rates. While the literature utilized a target network update C of 10,000 frames, we utilized an update rate of C = 2000 actions.

Finally, note that the exploration parameter, ϵ, begins at 1.0, and is linearly decreased over $100,000$ actions until it reaches 0.001, at which point it is constant for the rest of the episodes. The large initial exploration parameter is used to build up the replay buffer with transitions that lead to poor results.

5 Results

In this section, we present our results on the three different agents. Note that due to time constraints and computational resources, we were not able to perform any hyperparameter search and only used a simple guess-and-check for our two learning agents' parameters. As a result, we are not able to truly perform an "apples-to-apples" comparison between each of these agents [13], with both learning agents tuned to their best parameters against the BCTS agent.

Additionally, we were only able to run our two learning agents once. It is known that the current state of reinforcement learning literature often neglects proper statistical and error analyses when presenting their results [13].

With these two deficiencies in mind, we do not present results that claim anything about the overall performance of our learning agents in our Tetris environment. We only provide some analyses on each agent with their particular set of hyperparameters and for their single run.

5.1 BCTS Agent

This simple agent with its predetermined weight vector, $\mathbf{w}$, performed extremely well. We ran this agent for three episodes, and it managed to clear up to 30000 lines in the first two trials, and well over 80000 lines on the third trial.

Episode #	# Cleared Lines	# Actions ($\mathcal{A}_g$) Survived
1	27040	67638
2	16495	41277
3	$>$ 80000	$>$ 198000

Table 1: Number of Lines and Actions Cleared for 3 Episodes with BCTS Agent

We were unable to run the last episode to completion due to its extremely long length. It appears that the BCTS agent only failed its first two runs due to some sequence of generated tetriminos that resulted in a game over. However, the original authors of the BCTS paper claimed that this agent is able to clear "$30,000,000 \pm 20\%$ lines with probability 0.95" [18].

In fact, the first two episodes achieve a line-clearing score 5 standard deviations under the authors' original claim. If their claim is to be believed and assuming a normal distribution of scores, it is clear that our implementation is significantly different from the BCTS paper. We attribute this difference primarily to two key differences. The first difference is due to our one-step evaluation as noted before with Equation 3 - the knowledge of the second tetrimino allows the agent to plan to place pieces, accordingly. The second difference, albeit comparatively minor, could likely be due to the difference in implementation of the feature vector, $\mathbf{x}(s, a)$, as we could not find the code that the authors used to evaluate their agent.

5.2 Linear Sarsa(λ) Agent

With our first learning agent, we were unable to find hyperparameters to guide it toward a good policy. We show the number of survived actions and line clears per episode in Figure 4. For each episode, the agent manages to place around 21.9 pieces, on average, before reaching the top of the board. Additionally, it rarely clears any lines. On average, the agent manages to clear 3 lines out of every 100 group actions.

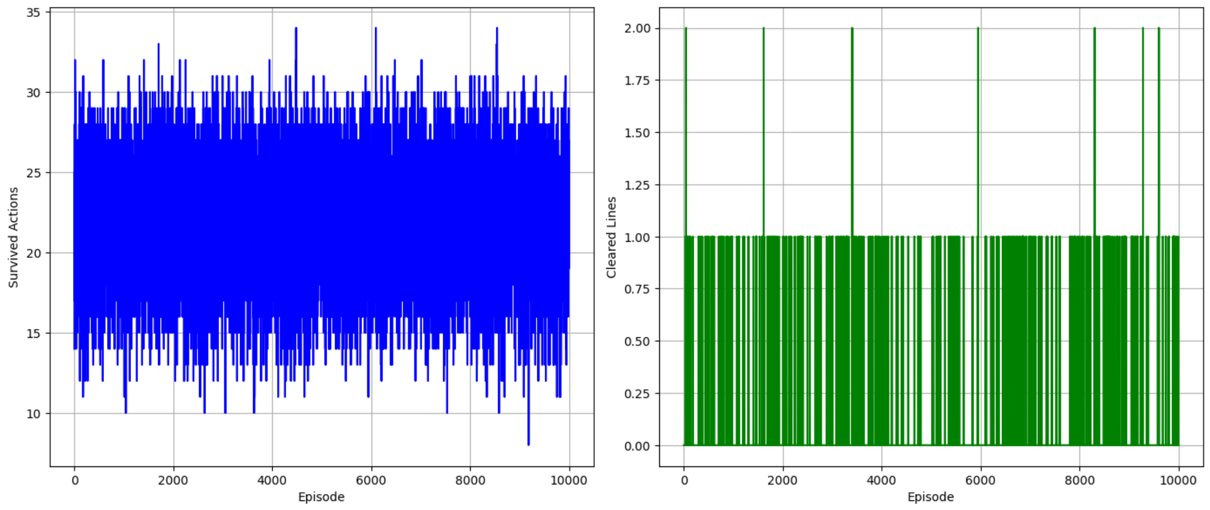


Figure 4: Sarsa(λ) Scores Over Episodes. Blue is for the number of survived actions while green is for the number of cleared lines.

While we weren't able to get interesting data for this agent, the weights can offer some insights to why it did not converge. We list the weights of the feature vector from the BCTS paper side-by-side with the weights obtained by our agent in Table 2.

The first seven weights for the Sarsa(λ) agent tells us that it does, in fact, recognize that those features in the $\mathcal{S}_g \times \mathcal{A}_g$ space can lead to the terminal state. However, for the "eroded piece cells" feature, it still has a weight of 0.0. Although part of the reason could be attributed to our chosen learning and exploration parameters, α and ϵ, this suggests that the agent did not experience enough line clearing transitions to recognize that this feature is essential for its success. Perhaps a higher ϵ value would have enabled it to explore more and be able to clear lines better.

$\mathbf{x}(s,a)$ Names	BCTS Weights	Sarsa(λ) Weights
landing height	-12.63	-37.42
row transitions	-9.92	-25.10
column transitions	-19.77	-25.80
num holes	-13.08	-18.11
cumulative wells	-10.49	-1.03
hole depth	-1.61	-11.98
rows with holes	-24.04	-34.41
eroded piece cells	6.60	0.0

Table 2: Comparison of Weights for BCTS and Linear Sarsa(λ). BCTS weights obtained through "cross-entropy" optimization [18], while Sarsa(λ) weights obtained from training agent over 10,000 episodes with the hyperparameters in Table 3.

5.3 DQN Agent

Our last agent that we experimented with did, in fact, converge to a better-than-random policy. Figure 5 showcases the same evaluation score over episodes graph as the previous section. While the first 50,000 or so episodes remained largely random with poor line clearing ability, just like our Sarsa(λ) agent, our design of the agent along with the hyperparameters led it to begin clearing significantly more lines per episode.

As mentioned before, we scaled our exploration parameter, ϵ, from 1.0 to 0.001 over the course of 100,000 actions. Figure 6 shows the point at which we cross the 100,000 total actions taken, at which point the agent began to follow its policy very strictly. As the peaks in the graphs seem to suggest, the agent begins to slowly improve over the last 50,000 episodes. In hindsight, we might have obtained faster convergence by raising the ending ϵ from 0.001 to 0.01 or slowly lowering it over a longer period of time.

Unfortunately, we did not get a converged result, and we may have gotten an even better agent by letting it learn over many more episodes, perhaps on the order of millions.

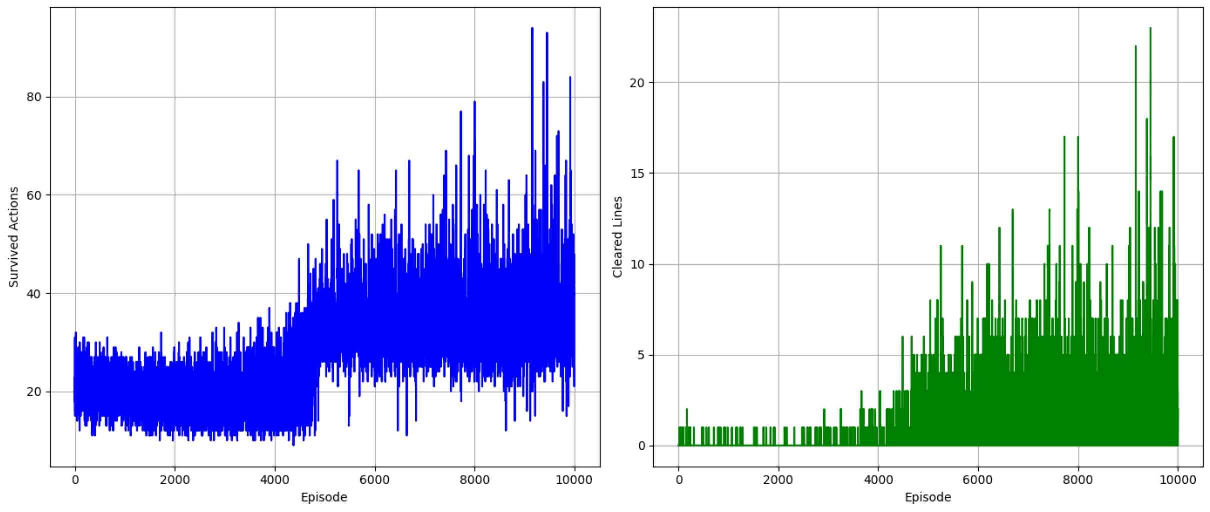


Figure 5: DQN Scores Over Episodes.

While we cannot comment and precisely compare our results between our linear Sarsa(λ) agent and our DQN, the one key defining feature of DQN may have led it to converge better than our simpler agent would be its replay buffer. Our particular choice of the constant ϵ in the Sarsa(λ) would not have enabled it to explore many insightful transitions. On the other hand, after its initial exploration phase, is able to observe these line-clearing transitions more frequently.

Again, for a more accurate comparison, we should have scaled the ϵ parameter in the same way for both agents, and we cannot claim that this replay buffer is the sole reason why the DQN agent converged to a better line-clearing policy.

6 Conclusion

The exploration into the development of Tetris-playing agents using reinforcement learning techniques has yielded insightful preliminary results. The DQN agent emerged as notably superior to the random agent, affirming the efficacy

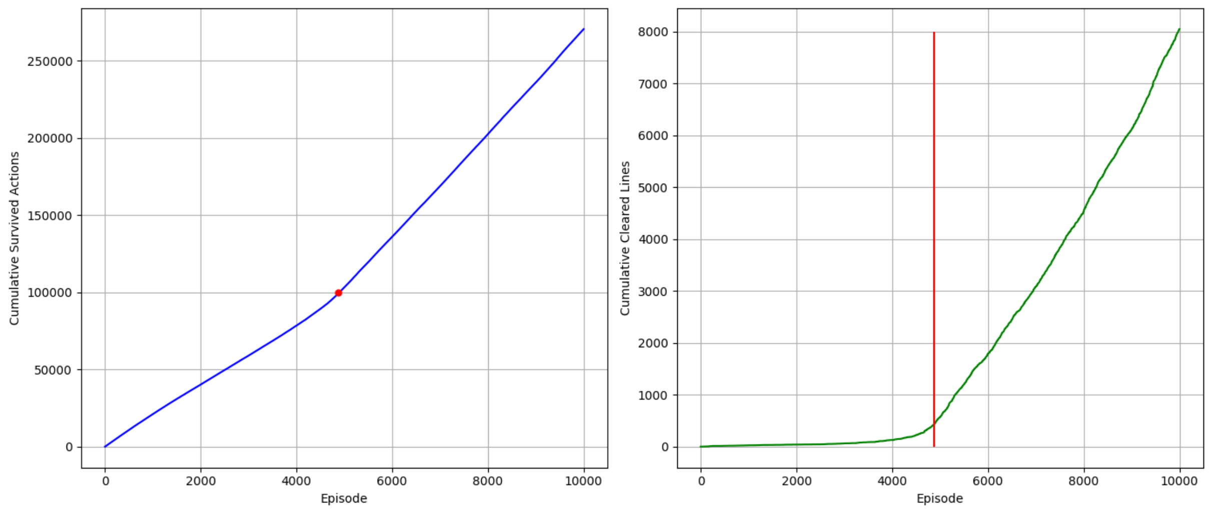


Figure 6: DQN Cumulative Scores Over Episodes. The red dot on the left graph designates the point at which the exploration parameter, ϵ, reached its minimum value at 0.001, as listed in Table 4. The red line is placed at episode 4876, which corresponds to the red dot. The lack of error bars is due to the fact that we only have a single run.

of using convolutional neural networks within the established state-action framework for this type of application. Additionally, the feature set employed demonstrated enough relevance and effectiveness to facilitate meaningful gameplay by the BCTS agent.

However, the study faced several critical limitations. The scope of experiments was significantly constrained by computational resources, leading to a limited number of runs which hinders the reliability and statistical significance of our results. This is further exacerabated by the stochastic nature of the Tetris environment.

Additionally, the hyperparameters were selected through a non-systematic approach, which might have compromised the line-clearing policy potential of these two agents. Furthermore, the inability to train and test the agents across a diverse set of random seeds means that our agents might not generalize well when running them on different random initializations.

If we had more time, we would address these deficiencies by employing systematic hyperparameter tuning, increasing the number of trials for more robust statistical validation, and testing the agents on varied initial random seeds to assess their adaptability and robustness. This would provide a deeper understanding of the strategic capacities and limitations of reinforcement learning models in complex, random environments like Tetris.

7 Appendix

7.1 Hyperparameters

Parameter	Value
Learning Rate α	0.001
Discount Factor γ	0.999
Exploration Rate ϵ	0.001
Eligibility Parameter λ	0.8

Table 3: Table of Sarsa(λ) Agent Parameters Used in This Paper

References

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

[2] Glen Berseth, Daniel Geng, Coline Devin, Nicholas Rhinehart, Chelsea Finn, Dinesh Jayaraman, and Sergey Levine. Smirl: Surprise minimizing reinforcement learning in unstable environments. In *International Conference on Learning Representations*, 2021. URL: `https://api.semanticscholar.org/CorpusID:231854994`.

Parameter	Value
Learning Rate α	0.001
Discount Factor γ	0.999
Initial Exploration Rate	1.0
Ending Exploration Rate	0.001
Number of Actions Until Ending Exp. Rate	100,000
Capacity of Replay Buffer $\mathcal{D}(\mathcal{N})$	1,000,000
Batch Size Sampled from $\mathcal{D}$	64
Number of Actions Before Target Update, `C`	2,000

Table 4: Table of DQN Agent Parameters Used in This Paper

[3] Victor Gabillon, Mohammad Ghavamzadeh, and Bruno Scherrer. Approximate dynamic programming finally performs well in the game of tetris. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013. URL: `https://proceedings.neurips.cc/paper_files/paper/2013/file/7504adad8bb96320eb3afdd4df6e1f60-Paper.pdf`.

[4] Weijie Guan, Zhufeng Li, and Hiroyuki Yamauchi. Graph structure exploration for reinforcement learning state embedding – train tetris agent with graph neural network. ICECC '23, page 42–48, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3592307.3592314`.

[5] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. 2015. URL: `https://arxiv.org/abs/1502.01852v1`, `doi:10.48550/arXiv.1502.01852`.

[6] Wojciech Jaśkowski, Marcin Szubert, Paweł Liskowski, and Krzysztof Krawiec. High-dimensional function approximation for knowledge-free reinforcement learning: a case study in sz-tetris. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, page 567–573, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2739480.2754783`.

[7] kautenja. gym-tetris. URL: `https://pypi.org/project/gym-tetris/`.

[8] Rex L. Reinforcement learning on tetris. URL: `https://rex-l.medium.com/reinforcement-learning-on-tetris-707f75716c37`.

[9] Tom Lokovic. NES Tetris DAS: Tools to understand it and improve your skill. URL: `https://www.youtube.com/watch?v=JeccfAI_ujo`.

[10] MeatFighter. Applying artificial intelligence to nintendo tetris. URL: `https://meatfighter.com/nintendotetrisai/`.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. Playing atari with deep reinforcement learning. *DeepMind Technologies*, 2013.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

[13] Joelle Pineau. Reproducible, reusable, and robust reinforcement learning. URL: `https://media.neurips.cc/Conferences/NIPS2018/Slides/jpineau-NeurIPS-dec18-fb.pdf`.

[14] V. Sinitsin, O. Ibryaeva, V. Sakovskaya, and V. Eremeeva. Intelligent bearing fault diagnosis method combining mixed input and hybrid cnn-mlp model. *Mechanical Systems and Signal Processing*, 180:109454, 2022. URL: `https://www.sciencedirect.com/science/article/pii/S0888327022005714`, `doi:10.1016/j.ymssp.2022.109454`.

[15] Matt Stevens. Playing tetris with deep reinforcement learning. 2016. URL: `https://api.semanticscholar.org/CorpusID:41925763`.

[16] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction.* The MIT Press, 2018. URL: `http://incompleteideas.net/sutton/book/RLbook2018.pdf`.

[17] TetrisWiki. Wall kick. URL: `https://tetris.wiki/Wall_kick`.

[18] Christophe Thiery and Bruno Scherrer. Building controllers for tetris. *J. Int. Comput. Games Assoc.*, 32:3–11, 2009. URL: `https://api.semanticscholar.org/CorpusID:14376165`.

[19] Kaitlyn Tsurunda. A closer look at tetris: Analysis of a variant game. *Atlantic Electronic Journal of Mathematics*, 2010. URL: `http://euclid.trentu.ca/aejm/V4N1/Tsuruda.V4N1.pdf`.

[20] Uglemat. Matris. URL: `https://github.com/Uglemat/MaTris`.

[21] Yu Yan, Peng Liu, Jin Zhao, Chengxi Zhang, and Guangwei Wang. Deep reinforcement learning in playing tetris with robotic arm experiment. *Transactions of the Institute of Measurement and Control*, 0(0):01423312221114694, 0. `arXiv:https://doi.org/10.1177/01423312221114694`, `doi:10.1177/01423312221114694`.

[22] Amy Zhang and Peter Stone. Lecture notes - Modern Landscape: Part I. URL: `https://www.cs.utexas.edu/~pstone/Courses/394Rspring24/resources/week10-modern-rl.pdf`.

[23] Özgür Şimşek, Simón Algorta, and Amit Kothiyal. Why most decisions are easy in tetris—and perhaps in other sequential decision problems, as well. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1757–1765, New York, New York, USA, 20–22 Jun 2016. PMLR. URL: `https://proceedings.mlr.press/v48/simsek16.html`.