Exploring RL Algorithms for Solving Tetris

Github Repository: https://github.com/kstad21/COGS188_jhek/tree/main

Jiasheng Zhou (A17385701), Harry Wang (A17326675), Elvin Li (A17438307), Kate Stadler (A17301232)

Abstract

The goal of this project was to investigate the ways reinforcement learning can be used to train an agent to play Tetris. Data was collected through live simulations, which made the use of Tetris environments important. We started off with the Arcade Learning Environment (ALE) version of Tetris but discovered that training traditional agents such as Q-Learning and TD(0) on this platform was difficult due to the difficulty in characterizing rewards and the infeasibly large state space. We test Monte Carlo, Q-Learning, PPO, and Deep-Q Networks on ALE and show that learning is minimal for the agents we implemented due to the lack of effective rewards and heuristics. Next, we use a more custom Tetris environment with the same models to utilize heuristics such as bumpiness, number of holes and custom reward functions in our training, and show that reinforcement learning models can be trained to sustain play with the help of these heuristics. Results are measured by analyzing loss during training, steps per episode (more steps means the agent "stays alive" for longer), rewards over time, and the total number of lines cleared for each agent. Through our various experiments, we found some key characteristics that drastically improve performance, such as the importance of deep learning, custom rewards, and well-designed heuristics.

Background

Tetris is a video game created by a Russian software engineer that became popular worldwide. In the game, there is a grid-environment where differently shaped blocks "fall" from the top of the grid to the bottom. It is the user's job to arrange the blocks (by moving them left and right and/or rotating them) so that they can survive as long as possible without any of the blocks accumulating all the way to the top of the grid. Rows can be dissolved (therefore giving the user more space) if every single grid in a row is filled by any part of a block (a 'tetris' is a quadruple of rows filled and cleared at the same time)[1]. Strategy for the game involves trying to maximize the number of cleared rows, especially when the goal is to receive as many points as possible. In the situation where the goal is to "stay alive" as long as possible, the main goal is to keep the highest block as low as possible, often by virtue of row-clearings and/or tetrominoes.

A 2013 paper introduced the Arcade Learning Environment (ALE)[2] that we plan to utilize in this project. ALE's goal is to provide an interface to hundreds of Atari game environments, as

---

[1] Weisberger, M. (13 Oct 2016) The Bizarre History of 'Tetris'. *LiveScience*. https://www.livescience.com/56481-strange-history-of-tetris.html

[2] Bellemare, M. et al. (14 Jun 2013) The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*. *https://jair.org/index.php/jair/article/view/10819*

well as methods to evaluate and compare approaches used to train agents in these games. There are different methods for feature construction and three simple baseline agents: Random, which picks a random action every frame, Const, which picks a single action throughout an episode, and Perturb, which selects a fixed action with a 95% probability and is uniformly random otherwise.

It has been proven that even in an offline version of Tetris, it is NP-complete to "maximize the number of cleared rows, maximize the number of tetrises, minimize the maximum height of an occupied square, or maximize the number of pieces played before the game ends"[3]. These results held when players were restricted to only 2 rotation/translation moves before each piece drops in height, restricted piece sets, and with an infinitely tall gameboard. This is why we are interested in testing the performance of different models and hyperparameters as we train an agent to play the game.

Even before the 2013 ALE was released, there were several attempts at training an agent to play Tetris. In 1996, Tsitsiklis & Van Roy used feature-based dynamic programming (number of holes and height of the highest column) to achieve a score of around 30 cleared lines on a 16x10 grid[4]. In the same year, Bertsekas & Tsitsiklis added the height of each column and the difference in height between adjacent columns as features. They achieved a higher score of 2800 lines using lambda-policy iteration[5]. Later, even further features were added, including mean column height and the sum of the differences in adjacent column height. Least-squares policy iteration achieved an average score of between 1000 and 3000 lines[6].

Due to the design of Tetris, it doesn't really make sense to have a reward function that gives rewards only at the end of the game. One TD(0)-Learning implementation uses linear combinations of weighted features, such as the value of the highest-used column, the average of the heights of all used columns, the number of holes between pieces at each given time, and the "quadratic unevenness" of the profile (which is a result of summing the squared values of

---

[3] Demaine, E. et al. (21 Oct 2002) Tetris is Hard, Even to Approximate. arXiv.org. https://arxiv.org/abs/cs/0210020

[4] Tsitsiklis, J., Van Roy, B. (5 May 1996) An Analysis of Temporal-Difference Learning with Function Approximation. *IEEE TRANSACTIONS ON AUTOMATIC CONTROL*. https://www.mit.edu/~jnt/Papers/J063-97-bvr-td.pdf

[5] Bertsekas, D., Tsitsiklis, J. (1996) Neuro-Dynamic Programming. *Athena Scientific*.

[6] Lagoudakis, M. et al. (2002) Least-squares methods in reinforcement learning for control. *Hellenic Conference on Artificial Intelligence*.

the differences of neighboring columns)[7]. In order to reduce the state space, a constrained height difference between adjacent columns was used to encode each state. In this experiment, it was shown that lower values of ε were beneficial when using an epsilon-greedy policy.

Tetris can also be modeled as a Markov Decision process if its state space is somehow reduced. Without reduction, a simple 20x10 board has 2200 ways to fill it and even with the requirement that no row be completely full, this is still (210 - 1)20. This 2022 paper's[8] most successful approach, "Fitted Value Iteration", chose a small set of features and represented each state in terms of this set of features. This method was contingent upon "featurization" of the MDP and a small number of samples from the original state space needed to represent the state-value relationship. The article also found that the features Max-Height, Num-Holes, and Num-Covers were promising features when it came to training an agent.

We hope to combine many well-researched ideas and progressively build a better agent capable of solving the tetris environment. Through better model architectures, heuristic and reward definitions, we aim to optimize the agent's gameplay performance.

Problem Statement

The core problem in this project is developing multiple Reinforcement-Learning based algorithms for learning a successful strategy to play Tetris, the classic block-stacking video game. Since the game is made to run infinitely, approximating reward as well as the action space is the most challenging part of the problem. The state space is so large that some of the algorithms we've learned are not feasible, and state representations can be complicated due to how many possibilities a board can hold. We will use algorithms and approximations that we have learned about in this class or researched (see Background) to train an agent to play Tetris on environments either provided by ALE or another source. Along with this goal, the project aims to compare and contrast the various models, to determine the relative strengths and weaknesses of each strategy through means of comparing rewards convergence rate and best score.

Data

---

[7] Thiam, P. et al. (2014) A Reinforcement Learning Algorithm to Train a Tetris Playing Agent. *Artificial Neural Networks in Pattern Recognition*. https://link.springer.com/chapter/10.1007/978-3-319-11656-3_15

[8] Bodoia, M., Puranik, A. (2022) Applying Reinforcement Learning to Competitive Tetris. https://cs229.stanford.edu/proj2012/BodoiaPuranik-ApplyingReinforcementLearningToCompetitiveTetris.pdf

We hope to use the learning algorithms' development of strategies, so the data we use will be generated dynamically through live situations. This will allow the models to learn through interaction with the environment. One such environment that we will be using is Farama's tetris environment (https://gymnasium.farama.org/environments/atari/tetris/), which provides a retro tetris setup for reinforcement learning research. Training models in this environment is analogous to collecting data, as the agent improves its actions and policies through many iterations of gameplay, guided by a reward system. The environment offers a discrete action space with five possible moves: move left, move right, drop down, rotate, and no operation. The game state is represented by an interface that can be launched in Python, with observations encoded as RGB or grayscale pixel values.

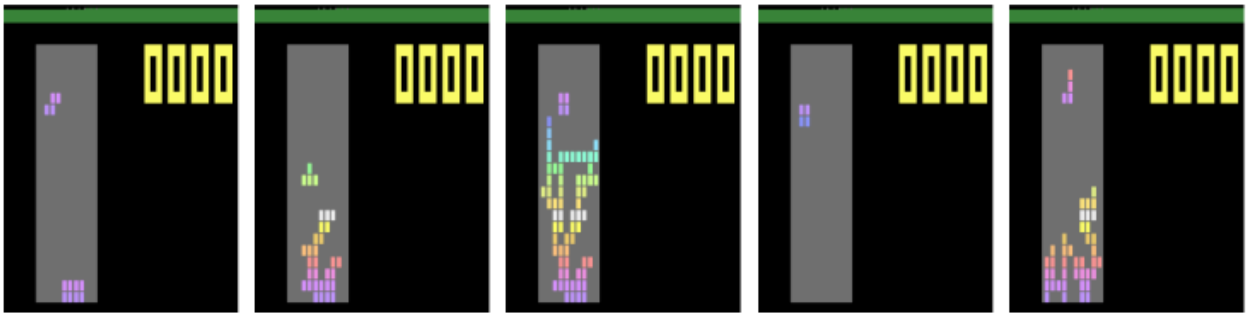

Fig 1: Snapshots of the tetris ALE.

Unfortunately, the default reward function only gives reward when a row is cleared. Especially when the agent is choosing random moves (which is usually the starting point for learning), clearing a row is extremely rare. Because of this, we decided to adapt another Tetris environment (https://github.com/nuno-faria/tetris-ai) and customize it to better suit our modeling needs, where we implement the models from scratch. This environment introduces a reward system that accounts for both the number of lines cleared and the number of timesteps survived, in contrast to the previous environment, which only rewarded line clears. Maximizing timesteps before an episode ends encourages the model to develop early survival strategies, which is particularly useful in the initial stages when clearing lines is more challenging. Additionally, we incorporate key heuristics such as bumpiness and holes to further refine the learning process. By discouraging excessive bumpiness, the model learns to maintain a flatter board, which facilitates easier piece placement and line clears. Similarly, penalizing holes ensures that the model prioritizes minimizing empty spaces trapped beneath blocks, preventing situations that make future moves difficult.

Proposed Solution

Models

We will evaluate a range of models across the two environments, progressing from traditional approaches in the simpler environment to deep learning models in the custom one. This approach allows us to systematically assess the level of complexity required for effectively solving Tetris. By starting with simpler models and reward functions, we can determine whether a less complex solution is sufficient, which would be preferable due to its greater interpretability and lower computational cost. If more advanced techniques are necessary, we can incrementally introduce complexity while analyzing their impact on performance and efficiency.

Q-Learning

Q-Learning learns by updating a table of q-values as it investigates the action and state space. We followed the following policy update: $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$. The algorithm uses epsilon-greedy action selection and updates q-values accordingly. We trained the agent over a given number of episodes with the following parameters: $\alpha = 0.1, \gamma = 1.0, \varepsilon = 1.0, \varepsilon_{min} = 0.1, decay = 0.995$. Applying this to the ALE environment required that we discretize the state space by converting the 128-byte RAM state into a hashable tuple.

Temporal Difference Learning (TD(0))

TD(0) learns by updating estimated state values based not only on immediate reward but also the estimated value of the next state. We followed the SARSA update policy: $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$. In order to select actions, we used epsilon greedy and random tie breaking. We also used this method on ALE as it is one of the less complex methods available.

Monte Carlo (MC)

Monte Carlo simulations estimate the best move through repeated rollouts of future game states. The online version evaluates all possible actions by simulating random sequences of moves to a fixed depth, selecting the action with the highest average reward. This method operates entirely online without training and serves as a baseline in the custom environment.

First-Visit Monte Carlo Control, in contrast, follows an episodic learning approach, updating the action-value function Q(s,a) based on full episodes. It averages returns over multiple visits to

refine the policy iteratively. We apply this method to the ALE environment to assess its adaptability in the state space.

Deep Q-Network (DQN)

The DQN learns to maximize long-term rewards by approximating the Q-value of each board state using a neural network. At each step, the agent evaluates all possible placements for the current tetromino and selects the action with the highest estimated Q-value, occasionally exploring random moves based on an epsilon-greedy strategy. The agent stores past experiences in a replay memory and periodically trains by sampling batches of past transitions, updating the Q-network using the Bellman equation. The training process refines the model's ability to predict optimal moves, enabling it to improve over time. We utilize this model on the custom Tetris environment as our "premier" model meant to solve the environment.

Proximal Policy Optimization (PPO)

Proximal Policy Optimization is a recent breakthrough in reinforcement learning[9] in the family of policy gradient methods, which model the policy directly through taking the gradients of the loss to estimate potential rewards. The main advantage of PPO is the use of clipped updates through either a surrogate objective or through measuring KL-divergence, which prevents the model from making too large of a step and keeps it more stable during training. In other words, policy updates are kept "proximal" to their original state, preventing it from making an unstable jump. Due to the promising results discussed in its original OpenAI paper, we chose this model to be a potential modern alternative to Deep Q-Learning, hopefully yielding comparable or superior results to the methods we learned in class, and as such is also applied to ALE and the custom environment.

For PPO specifically, due to how recent of a development it is and the lack of documentation for it apart from the original OpenAI preprint, several optimizations, loss calculations, and other minor improvements were taken from this blog: https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/. As the high-level description of PPO given by the paper skipped out on several details, the contributions from this article are worth mentioning due to the improvements they yielded on PPO's efficacy.

Our Tetris environment's discrete action space and constrained board size make it well-suited for reinforcement learning. The finite tetromino placements allow algorithms to explore and optimize decisions efficiently. Monte Carlo methods estimate expected rewards through

[9] https://arxiv.org/pdf/1707.06347

episodic simulations, while DQN and PPO use function approximation to generalize across states. Implemented in PyTorch or standard Python, our models will incorporate heuristics like timesteps, bumpiness, holes, and line clears to learn policies that maximize survival and rewards.

Evaluation Metrics

To ensure consistency across environments and models, we will use a standard 10x20 Tetris board and evaluate the models based on several key metrics. For models that fail to learn effectively, we will analyze their loss/reward graphs, examine their code, and observe the live performance to identify potential causes for poor performance. For successful models, we will compare their reward graphs, assessing patterns such as peak rewards, learning efficiency (e.g., plateaus or rate of improvement), and overall stability. Ultimately, we will measure each model's number of lines cleared in the live environment as a direct performance indicator.

Results

We present the progression of models applied to different environments, highlighting how our solutions improved over time. Each section details the environment, the models used, key performance metrics, and the rationale for transitioning to more complex approaches.

Q-Learning and TD(0), and Monte Carlo on ALE

We initially attempted to run some basic algorithms from this class on the ALE-Tetris environment. Although ALE does not give the most information regarding rewards, we felt this was appropriate due to the relatively smaller state space, and hypothesized that the smaller environment would be simpler to solve. The 3 algorithms we attempted to implement in the ALE environment were Monte Carlo, Temporal Difference Learning (TD(0)), and Q-Learning. Each of these algorithms was implemented according to the formal policy update equations and trained for 1000 episodes to see if they would learn the state space. Overall, these algorithms performed poorly and were unable to solve the environment.

For Monte Carlo, the algorithm did appear to decrease the training loss initially, but this may be because it learned to align pieces (without clearing many rows) and extend the episode somewhat.

Fig 2: Snapshot of how Monte Carlo performs on the board (left) and the training loss progression as episodes increase (right)

In the case of TD(0) and Q-Learning, the reward and loss definitions prevented the algorithm from exploiting the environment altogether, and the end result after testing was an agent that essentially did not move. Even after modifying the action choice method to incorporate some variance when the Q table was 0, we did not observe any improvement in the agent.

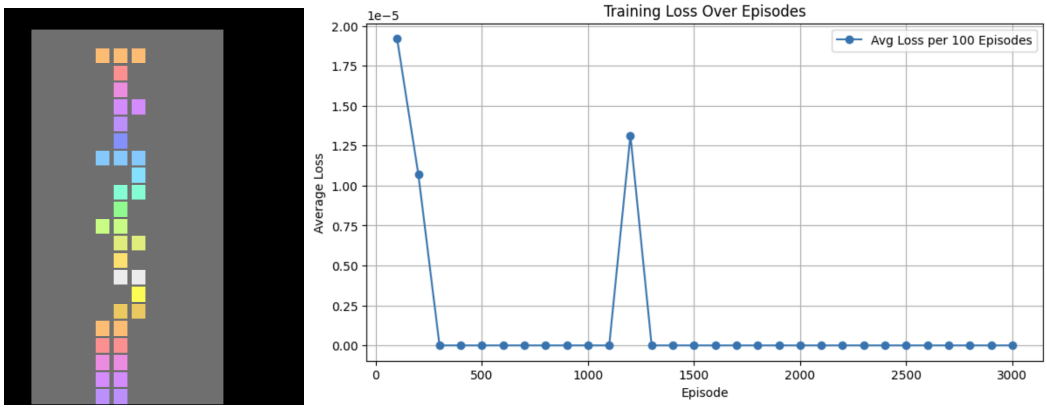

Fig 3: Snapshot of how Q-Learning performs on the board (left) and the Q-Learning error progression as episodes increase (right)

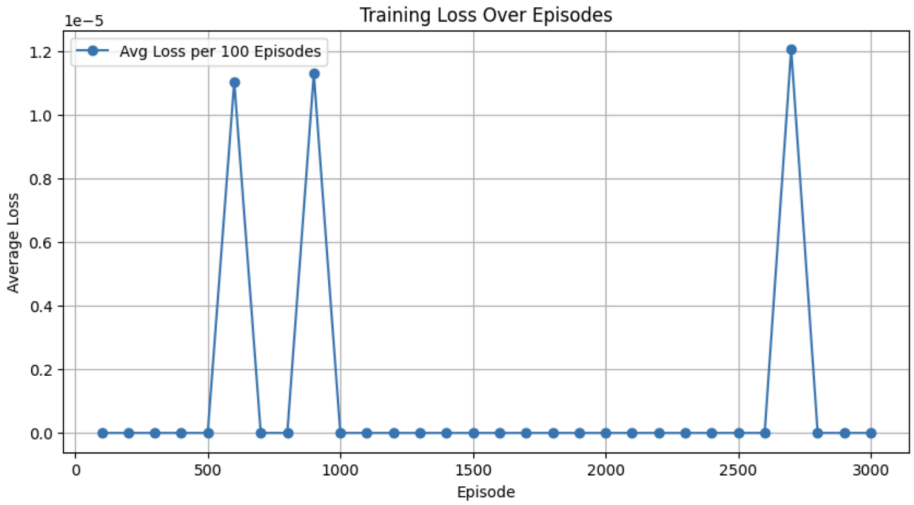

Fig 4: Training loss for TD(0)

Note that even though the loss is 0 for the majority of the time, this is mainly due to a programming design choice to avoid division by 0 when calculating average loss, and the non-zero terms don't actually converge which indicates that the agent is still failing to learn the environment at a reasonable pace. In fact, we propose that this reward is the result of exploration during training and not exploitation, since the agent fails to display a similar reward during evaluation.

Takeaways from the ALE environment:
The models were largely incapable of fully learning and solving the environment, and at best managed to delay stacking up by clumping the pieces haphazardly. We attribute this difficulty to the large state size of the Tetris environment: with a 20 x 10 board, and 7 pieces in hand, we calculated that the state size is at least 10,000. Even with high learning rates, hyperparameter tuning, and boosting the variance in action choice by breaking ties randomly, the agents did not learn enough of the action space at a reasonable pace to produce decent results.

DQN and PPO on ALE
We also attempted to train an agent on the Tetris ALE using a Deep-Q network. Because of the lack of informative reward, we tried a few different (simple) custom rewards to train our agent. We started off by training an agent using only the given reward in the ALE: 1 point for clearing a row.

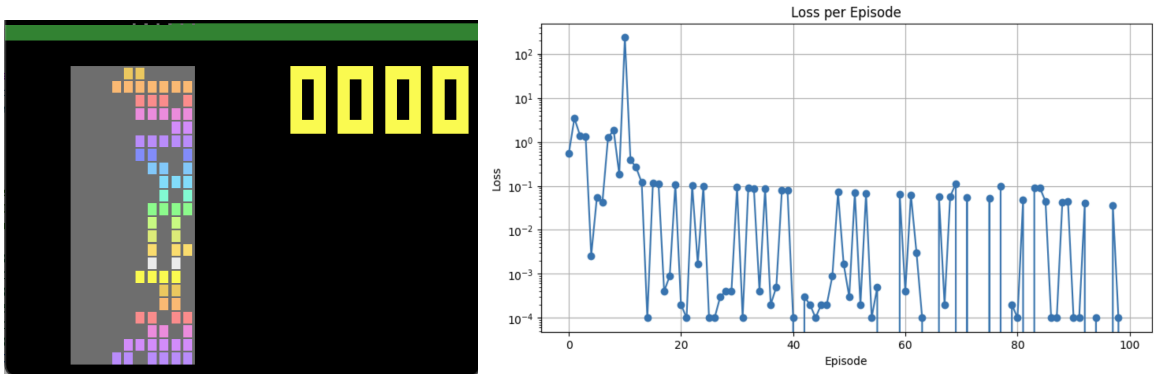

Fig 5: Visualization & loss per training episode for DQN, simplest reward

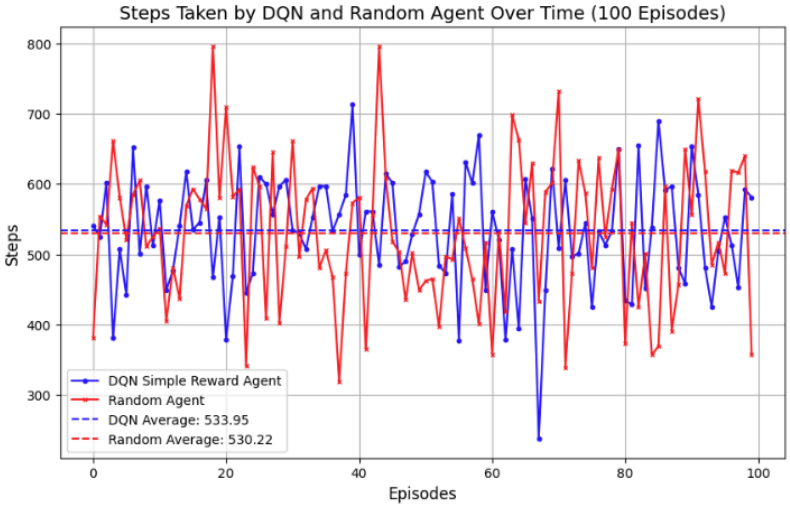

Fig 6: Steps per episode, random versus simple DQN agent

The DQN Agent (trained with a simple reward) did about as well as a random agent at “staying alive”, which makes sense because there is very little to learn from getting a reward when a row is cleared, especially starting from random actions. It is extremely rare to even discover the reward, making it difficult for the agent to actually learn anything.

For PPO trained on the same Tetris ALE environment, results were similar: Occasionally rows were cleared, but the rewards were too sparse for the agent to effectively learn anything, and even a heavily optimized PPO utilizing several wrappers to improve the environment functionality did not yield a strong model. While the metrics did show a decreasing policy loss and decreasing entropy, the actual average timesteps and rewards per 10 episodes painted a different picture, as shown below:

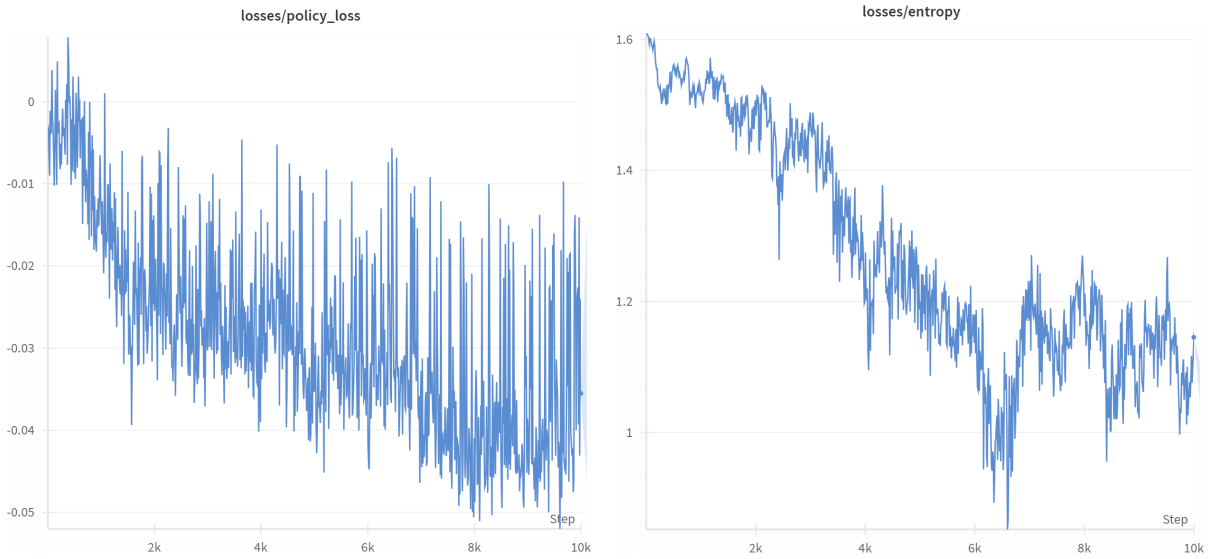

Fig 7: Policy Loss and Entropy for PPO - visibly decreasing

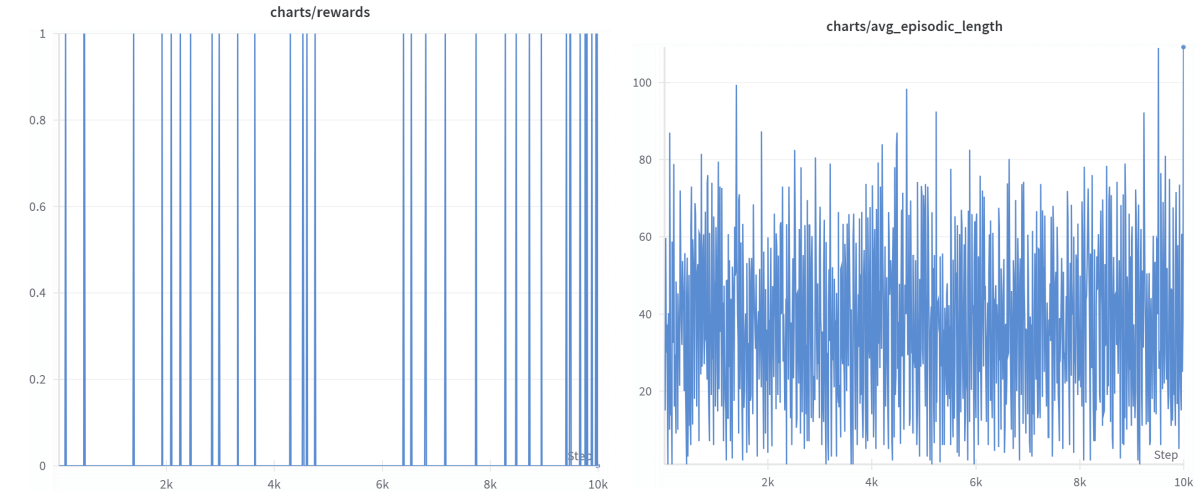

Fig 8: Rewards and Average Episode Length for PPO - no clear sign of improvement

Next, we added a small reward for “staying alive” each timestep for DQN. This agent did slightly better than the Agent trained with only the row-clear reward.

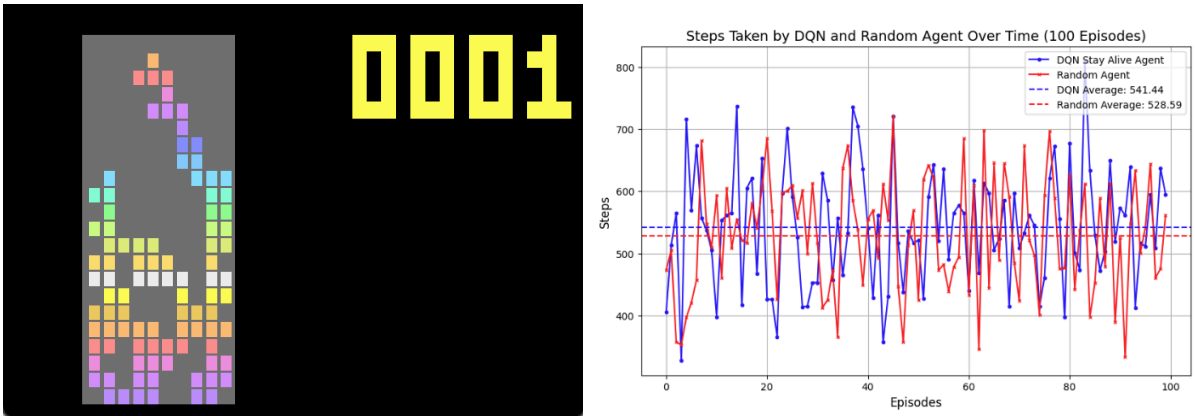

Figure 9: Visualization and steps per episode for "Stay Alive" vs random DQN agent

While adding a simple "staying alive" reward is somewhat helpful, it's clear that training an agent in the ALE environment is very difficult, especially because it's complicated to use heuristics. In the next section, we will use a custom setup to make quantifying rewards more straightforward and less vague for the agent.

DQN, PPO, MC on the custom Tetris environment

In the previous section, we chose to apply deep reinforcement learning algorithms to solve the state infeasibility problem, including DQN and PPO to train on the ALE environment. In this custom setup, we implement a dual reward system: the agent receives a reward of 1 for each timestep survived and an additional 10 for every line cleared. This design addresses a key challenge in the ALE environment, where clearing lines early in the game is difficult due to the randomness of initial moves. By incentivizing survival, the model learns to extend each episode, increasing the likelihood of discovering line-clearing strategies over time. Other heuristics to minimize such as bumpiness and holes are also introduced, encouraging the models to create a flat surface which is more ideal for clearing lines.

The first strategy we implemented was the online Monte Carlo simulation:

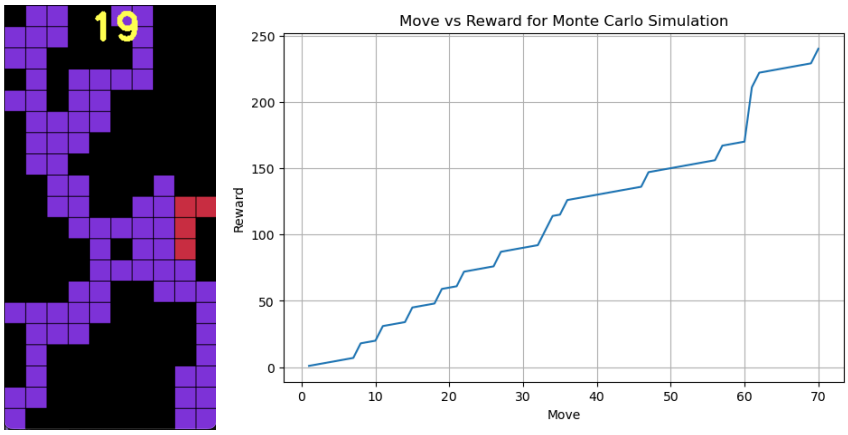

Fig 10: Visualization of the online Monte Carlo strategy (left) and the rewards graph (right)

The strategy was terminated once the pieces reached the top row. The rewards graph illustrates the accumulation of rewards over time, reflecting the progression of moves. Since this approach is not episodic and rewards increase by 1 per timestep, the graph naturally trends upward. However, as shown in the left figure, this corresponds to only 19 lines cleared - an improvement over previous methods but still falling short of our desired performance. Notably, the agent struggled to maintain a flat board, which is crucial for long-term play. Instead, it exhibited a tendency to place pieces somewhat randomly, prioritizing immediate line clears only when they were obvious. Additionally, it did not rotate pieces effectively to optimize multi-line clears or follow a structured placement strategy, demonstrating that there are areas for improvement in decision-making.

For DQN and PPO, we ran the models with 3000 training episodes and various hyperparameters until we found a setup that could effectively learn the environment within the timeframe.

DQN:

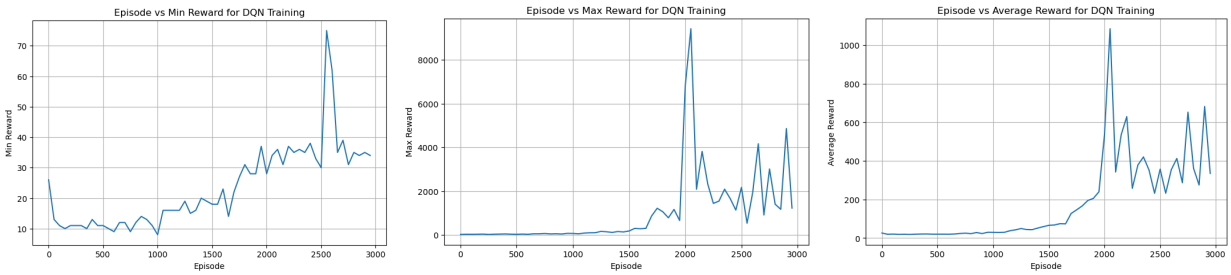

Fig 11: DQN's Minimum reward every 50 episodes (left), maximum reward every 50 episodes (middle), average reward every 50 episodes (right)

The rewards data was recorded every 50 episodes, allowing us to track the model's progression over time. We generated three graphs visualizing the maximum, minimum, and average rewards, providing insight into both the best and worst-case performance as well as overall consistency. At its peak, the model achieved a maximum reward of nearly 10,000, marking a significant improvement over the Monte Carlo simulation, which struggled to sustain long-term play.

To further evaluate its effectiveness, we saved the best-performing model and tested it in a live environment. During evaluation, the model demonstrated a strong ability to sustain gameplay and optimize board management, successfully clearing over 1,000 lines as shown in the figure below. Unlike earlier strategies, which often resulted in random placements and early terminations, the DQN agent on this custom environment exhibited a structured approach, prioritizing flat board formations that facilitated easier line clears. This behavior suggests that the model effectively learned to balance short-term survival with long-term optimization, allowing it to extend gameplay and maximize rewards more effectively.

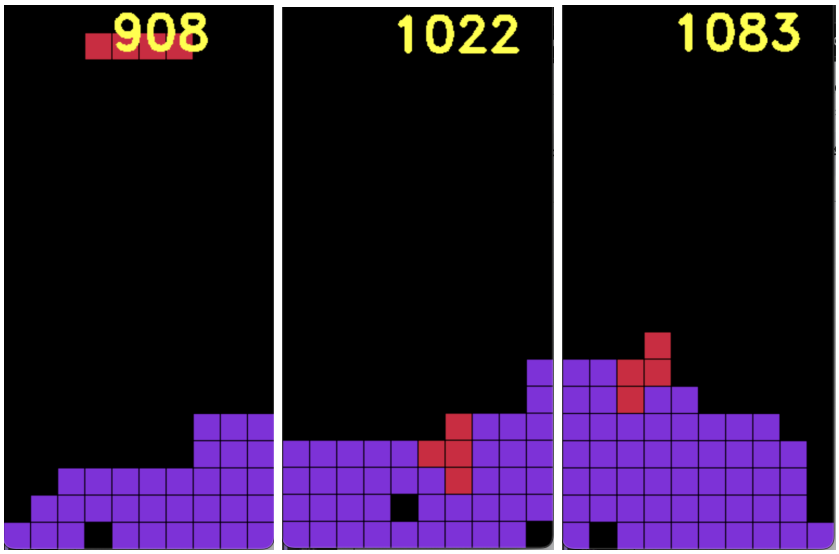

Fig 12: Snapshots of DQN's live performance. It generally is able to fill out bumps and minimize holes consistently.

PPO:

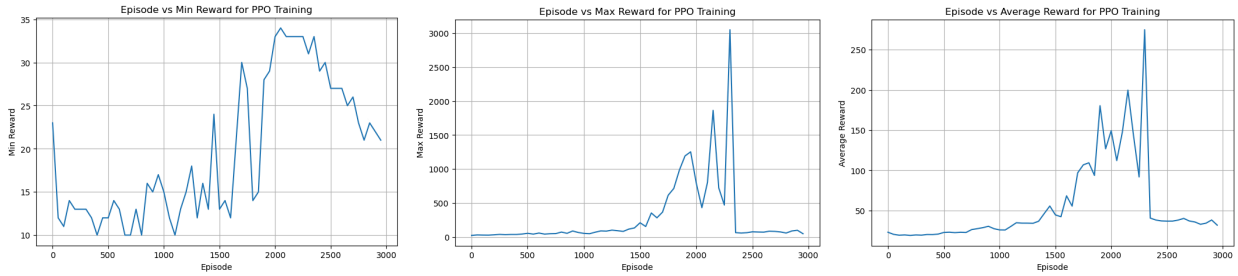

Fig 13: PPO's Minimum reward every 50 episodes (left), maximum reward every 50 episodes (middle), average reward every 50 episodes (right)

Similarly, we recorded the maximum, minimum, and average rewards every 50 episodes during PPO training to analyze its learning trends. One key observation was that PPO demonstrated early improvements, showing steady progress around 1,000 episodes, whereas DQN only began learning effectively after approximately 1,500 episodes when comparing their respective mean reward curves. However, despite PPO's faster initial learning, DQN significantly outperformed it in the long run, achieving a maximum reward of 10,000 compared to PPO's peak of only 3,000. Additionally, PPO exhibited signs of overfitting later in training. After around 2,500 episodes, its performance began to decline sharply, whereas DQN continued improving steadily. This suggests that while our PPO implementation may converge faster, it lacks long-term stability and generalization.

We also evaluated the live performance of PPO on the board, as shown in the figure below. While PPO outperformed our earlier solutions, it still fell significantly short of DQN's performance within the same environment. During live testing, PPO demonstrated moments of effective decision-making, successfully filling holes and maintaining a degree of consistency in managing bumps. However, it was not as proficient as DQN, and as a result, holes accumulated more frequently, ultimately leading to game termination after approximately 240 line clears. While PPO is capable of learning reasonable strategies, it struggles with long-term board management and lacks the sustained optimization observed in DQN.

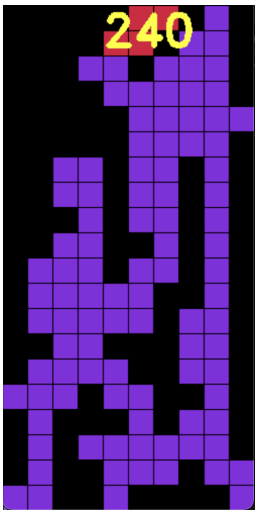

Fig 14: The board state of PPO on the custom environment right as the game terminates, clearing a total of 240 rows

Discussion

Interpretations

Through our experiments, we explored various approaches to solving Tetris while adhering to the game's core rules. Our findings indicate that deep reinforcement learning is essential for developing an effective algorithm, and that reward structuring and heuristic design significantly influence learning efficiency and overall performance.

Non-deep learning models performed poorly due to the vast state space of Tetris. Algorithms like Q-Learning and TD(0) update policies based on states, making them impractical for a game with an exponential number of possible states, influenced by factors such as holes, bumpiness, and piece placements. Our initial experiments with Q-Learning, TD(0), and First-Visit Monte Carlo on the ALE environment confirmed these limitations. Even after 3,000 training iterations, none of these algorithms converged to a stable policy. As shown in Figures 2, 3, and 4, their loss graphs exhibit erratic behavior, lacking a clear convergence trend. In contrast, Figure 7 illustrates the loss graph of PPO on ALE, which shows steady improvement over time, reinforcing the infeasibility of tabular RL methods in such a large and dynamic state space.

We found that reward structuring plays a crucial role in the performance of learning algorithms. The original ALE environment rewarded the agent only for clearing lines, making it difficult for early-stage training when the agent explored moves randomly. To address this, we introduced an additional survival-based reward, incentivizing the agent to prolong gameplay before it learned to clear lines. This modification enabled gradual skill acquisition, as the agent

first learned to maximize survival and then progressed to clearing rows once it discovered more effective strategies. This impact is best demonstrated in Figure 10, where the original first-visit Monte Carlo method failed to clear any lines in ALE, but the online Monte Carlo method in the custom environment successfully cleared 19 lines.

Additionally, heuristics such as bumpiness and holes significantly improved our solutions, a capability largely made possible by deep learning models. Bumpiness, which measures the height difference between adjacent columns, plays a crucial role in board stability as a high bumpiness value makes placements difficult and increases the likelihood of holes. By incorporating bumpiness into our reinforcement learning models, the agent learned to prioritize flatter board structures, improving long-term playability. Another key heuristic was hole count, which strongly correlates with a poor board state. Unlike non-deep learning models that struggle with high-dimensional state spaces, deep reinforcement learning effectively minimizes holes over time, balancing short-term rewards with long-term survival strategies. Figure 12 best illustrates these improvements, showing that DQN, trained in our custom environment with modified rewards and heuristics, effectively reduces bumpiness and holes. This resulted in our final DQN model clearing over 1,000 lines, which is substantially better than all other attempts.

Our experiments demonstrate that deep reinforcement learning, when combined with well-designed heuristics and reward structuring, is highly effective in solving Tetris. While traditional RL methods struggled with the game's vast state space, DQN successfully learned to optimize board stability and maximize line clears. By incorporating heuristics like bumpiness and holes and new reward functions, we enabled the agent to develop more strategic gameplay, leading to significantly improved performance.

Limitations

Given the nature of reinforcement learning and that evaluating positions for tetris can take a long time, we did not allocate a lot of time in tuning hyperparameters for our deep learning models. For DQN/PPO in the custom environment, fully training the models takes around an hour each, so we did not have the time to search the hyperparameter space. If we did, we could have potentially improved both models by a significant margin, but given this time constraint we believe the performance of our current models was strong evidence to highlight their effectiveness.

While we implemented heuristics and investigated ways to quantify states in Tetris, there are also still many other avenues to explore on that front, some of which are talked about in our

background section. We used some heuristics like bumpiness, hole count, and rewarding time of gameplay, but due to the huge state space of Tetris there are countless other ways to attempt to quantify the game.

Future Work

There are many directions we could further explore, ranging from the environment to the selection of models to just further optimization. To start off, variations in the Tetris environment could yield interesting results. Modern-day Tetris drastically varies from the original NES Tetris that we focused on for this project: New functions such as the hold piece (allowing you to store a piece and swap to it when convenient), piece previews (allowing for the model to see what is coming up and plan accordingly), and score modifiers (Performing Tetrises (4 line clears) or T-spins (turning a T block under an overhang so no holes are created in the board) could affect the model's decision making) could create drastic changes in the learned policy of our models. One especially exciting direction would be to train a model to play Tetris normally and then test it on modern multi-player versions of the game, to see how it would perform against human players. Another way to modify the environment would be to implement more heuristics rewarding the agent for "good" behaviors: While we did experiment with a few modifications such as rewards for minimizing bumpiness or holes which were successful in improving the model, the agent still does not play at the same skill level as a human due to it's sometimes odd decision-making and there might be room for improvement through the design of a more complex heuristic.

For this project, we used a selection of models that we already had learned about in class (Q-Learning, DQN, TD-Learning, Monte Carlo, etc), as well as one model from the current reinforcement learning landscape that we encountered during our own research (PPO) as it seemed promising and relatively simple to implement. However, there are still a plethora of RL paradigms that we did not have the chance to explore, and one clear opportunity for future work would be through implementing more models. To name just a few, Actor-Critic models (A2C/A3C), Transformer-based methods (Decision Transformer), search-based algorithms (Monte Carlo Tree Search), and genetic algorithms could potentially be adapted for Tetris, although the results may vary. One observation we definitely noticed during the development of this project was the sheer amount of reinforcement learning algorithms that have been created in recent years, and there were many that we would have liked to try but now remain only as future directions for this project.

Lastly, as mentioned in the limitations section, we did not have time to properly perform hyperparameter tuning on our models, which is a key future direction for this project. In

general, our implementations were pretty unoptimized - both the runtime and convergence for training could potentially be improved. As an example, we found during our tests with the ALE environment that there were several Atari environment wrappers that improved our models (with functions such as introducing a random number of NoOps at the start of a reset, clipping the rewards to a certain range to reduce error, and rescaling the raw ALE frames to better fit neural networks). We weren't able to make use of these in all of our tests, and there are probably a number of other minor improvements we could have left as well.

Ethics & Privacy

While there seem to be very few privacy implications of training an agent to play Tetris using reinforcement learning, the larger concept of training bots to play games does have ethical implications. Simple games like Tetris, Minesweeper, and even slither.io are special and nostalgic for many people because of the way we grew up playing them. We understand that the point of these games is to provide simple, pure entertainment that is easy to access. These games were designed by humans for humans, and because of this one could argue that training bots to beat these games goes against the spirit of the games themselves. Now, with AI becoming more popular and more and more environments being provided to easily train agents to play games, the spirit of the arcade-style game has definitely changed. Instead of the point of the game being to have fun and win, the point becomes utilizing the state space and AI algorithms to sit back and watch an agent play it for you. While this change of attitude certainly has merits, specifically in the realm of advancing reinforcement learning, improving state representations, and learning new things, it unarguably changes the nature of these types of games. As an extreme, this thought process contrasts the way we (and generations before) grew up animatedly playing slither.io in the middle of class or playing games with our friends at the arcade—newer generations may be more inclined just to train a bunch of bots, sit back in their gamer chairs and watch the game be won for them.

Also importantly, to avoid the possibility of a rogue superintelligence, we will avoid giving our models internet access capabilities during the training phase.

Conclusion

From our findings, it appears that reinforcement learning methods can be effective in solving the video game Tetris. Models such as Deep Q-Learning and Proximal Policy Optimization, when paired with a well designed environment for Tetris, were able to perform very well, clearing hundreds or thousands of lines before topping out. However, the success of these models was very dependent on both the heuristics being used and the design of the environment. When placed in the original ALE environment, none of the models performed

well due to the sparseness of rewards making policy/model updates infrequent. After adding rewards for timesteps survived, as opposed to only just lines cleared, as well as heuristics for measuring the messiness of the board, the models began performing much better. While there were time constraints that prevented us from exploring hyperparameters, other models, and variations of Tetris (which we hope to explore in future work and invite others to do so as well), we still managed to confirm the viability of RL for a game like Tetris that features a large state space.