Reinforcement Learning for Mini Tetris

Max Conway
University of Colorado Boulder
Computer Science
Max.Conway@colorado.edu

Gyanig Kumar
University of Colorado Boulder
Computer Science
Gyanig.Kumar@colorado.edu

***Abstract*—Tetris is a famous game created in 1985 by Alexey Pajitnov is one of the best-selling games of all time. In this paper, we train a reinforcement learning agent to play Tetris. We examine previous attempts to train an agent to play Tetris and analyze the challenges in reinforcement learning that make training to play Tetris particularly difficult. Finally, we train a Proximal Policy Optimization (PPO) and Deep Q Network (DQN) model on a smaller game of Tetris with fewer and simpler pieces to moderate success using the Torchrl library. We investigated the impact of the models architecture by benchmarking their performance on Convolutional Neural Network(CNN) and Fully-connected ANN. Interestingly, PPO outperforms DQN across both architectures. We conclude with a discussion of challenges and future works. The code for the project can be found here https://github.com/mconway2579/TetrisRL**

I. Introduction

Tetris was created in 1985 by Alexey Pajitnov. As one of the top-selling games ever, it is well-known and easily recognizable. You would be challenged to find someone who has never heard of or seen Tetris before. Tetris is a simple game conceptually, in which the player rotates and positions falling pieces (tetrominoes) to form complete horizontal lines, which are then cleared. The game ends when the height of the stack of pieces reaches the top of the board.

Tetris is a challenging reinforcement learning (RL) problem because it contains several quirks that challenge RL algorithms. Tetris challenges reinforcement learning algorithms with sparse reward, reward is infrequent and requires many "good moves" to clear a line, exploration for a reward, a line will almost never be cleared by a random agent, and credit assignment, when a line is cleared, it is unclear to the model what sequence of moves led to the reward. These factors combine to make Tetris a challenge for even modern RL algorithms without significant engineering effort.

To mitigate these challenges, the most common and successful approach is to engineer away the direct piece control [1], [5]. Instead of allowing the model to input raw commands (left, right, rotate, down, no-op) these approaches will get a value estimate from the model of all possible places the current piece can be placed and then handle control in the backend to place the piece in the configuration the model determined to have the highest value. These methods result in superhuman performance, but we feel they miss the point since a core challenge of the game for human players is controlling the piece to do what they intend.

By using RL for Tetris, we can gain insight into how RL solves (or doesn't) long-horizon problems, sparse reward problems, and large state spaces. Here we will review reinforcement learning concepts used in our approach and other attempts to solve Tetris with reinforcement learning, the TorchRL framework, formalize our problem and approach, present our results, discuss future improvements for this system.

II. Background and Related Work

A. Reinforcement Learning for Games

One of the most influential reinforcement learning papers is Playing Atari with Deep Reinforcement Learning [7] This research introduces the Deep Q Network (DQN). Before this paper, tabular Q learning was a popular reinforcement learning method, but tabular Q learning requires $O(|S| \times |A|)$ space since for every state action pair a value must be stored. Whereas, Deep Q learning instead learns a network function f that consumes a state and produces a vector of Q values of length $|A|$ for discrete action spaces $f\colon S \to \mathbb{R}^{|A|}$. f is learned by minimizing the Q loss via gradient descent.

$$\mathcal{L}(f(s)_a) = (f(s)_a - (R(s,a) + max_{a' \in A}(f(s')_{a'}))^2$$

Many improvements have been made to the initial DQN paper, including a Double Deep Q Network [12], which improves training stability by training two separate Q networks, one for acting and another for Q value estimation. Another improvement to the initial DQN paper is the use of prioritized replay buffers [9]. The prioritized replay buffer allows networks to train on samples they are performing worse on, rather than training on samples the network already accurately predicts.

Proximal Policy Optimization Algorithms [10] have emerged as a competitor to DQN; these methods optimize a surrogate objective that constrains the policy update by limiting the change in the probability ratio between the new and old policies to achieve stable and efficient on-policy learning.

The PPO network is learned by minimizing the loss

$$\mathcal{L}^{\text{PPO}}(\theta) = -\mathbb{E}_t\Big[\min\big(r_t(\theta)\,\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\,\hat{A}_t\big)\Big] + c_1\,\mathbb{E}_t\big[(V_\theta(s_t) - R_t)^2\big]\ -\ c_2\,\mathbb{E}_t\big[\mathcal{H}\big(\pi_\theta(\cdot \mid s_t)\big)\big],$$

When

- θ are the parameters of the network(s)
- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio
- $\hat{A}_t$ is the advantage estimate.
- ϵ is the clip-threshold.
- c_1 and c_2 are the value-loss and entropy weights.
- $\mathcal{H}$ is the policy entropy.
- V_θ is the value network
- R_t is the discounted future return that V_θ is trying to learn

B. TorchRL

TorchRL [2] is an extension built on top of the pytorch [8] deep learning framework. TorchRL provides implementations for many key components of deep reinforcement learning, including Policy Rollouts, Environment Wrappers & Transforms, Data Collectors, Replay Buffers, Prioritized and Uniform Samplers, Network layers and implementations, and objective function implementations like DDQN, PPO.

TorchRL connects all of these components through an optimized data structure called the tensordict. The tensordict works like a Python dictionary, except it can be indexed by keys or dimensions like a standard tensor. It also supports GPU computations and backpropagation through tensordict operations. The tensordict is the backbone of TorchRL, and many modules consume, produce, or modify tensordicts.

C. Tetris Reinforcment Learning

Tetris is an interesting reinforcement learning problem for several reasons, most notably credit assignment, exploration, and sparse reward. In Tetris, a reward is received when a line is cleared, which will almost never happen with a random agent, making exploring to find a reward challenging. Even if your model does clear a line, a credit assignment becomes a problem, because it was not just the previous action that cleared a line, but actions many time steps away could also be equally responsible for clearing that line.

Many efforts have been made to make Tetris learning easier. One notable effort used a custom reward function [11]:

$$(-0.51 \cdot \text{Height} + 0.76 \cdot \text{Lines} - 0.36 \cdot \text{Holes} - 0.18 \cdot \text{Bumpiness})$$

where Height is the highest block, lines are any lines cleared that frame, holes are the number of empty cells with a filled cell above them, and bumpiness is the sum of height differences across each column.

Superhuman Tetris agents have been achieved with a clever trick. Instead of teaching the model to relay input commands like humans, the superhuman agents are trained by learning a state value network. For the current piece, the environment generates all possible placements for the piece, and the network obtains its value estimate; then, the environment handles the actual controlling of the piece to position the piece in the highest value placement as determined by the network.

A Tetris reinforcement Learning researcher motivated this approach by saying, "When a human is playing Tetris, it is trivial to consider how to move a piece to a certain location; instead, we consider where we should move it. Hence, it is better to provide all possible locations the current piece can drop at, and let the AI choose the best one." [5] We reject this claim since input errors are a common source of mistakes for Tetris players. If the claim is that these agents are playing Tetris, they should risk input errors just as any other player would.

Tetris from raw inputs remains a challenge for reinforcement learning. In a survey of Tetris Reinforcement learning from 2019 [1], they conclude, "No deep learning algorithm has learned to play well from raw inputs." Tetris is notably missing from many common Atari benchmarks. Both [3], [4] evaluate their proposed methods on many Atari Games, but interestingly enough, do not evaluate on Tetris.

Some work has been successful on a smaller version of Tetris [6] trained a Q table for a small version of Tetris with smaller pieces and a smaller game board. We draw inspiration from this work by reducing our environment from regular Tetris to mini Tetris.

III. Problem Formulation & Solution Approach

We formulate our problem as MDP (S, A, T, R, γ) with:

$$S = \{0,1\}^{H \times W \times 3}.$$

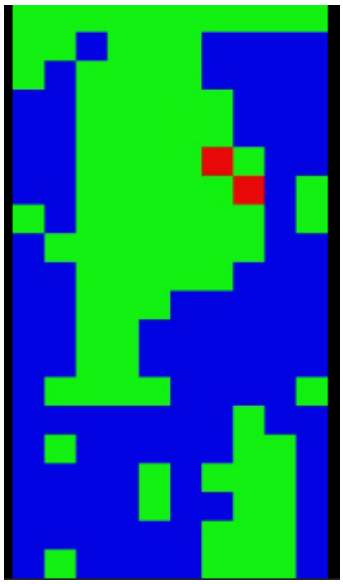


Fig. 1: Visualization of state

S is the set of all height by width by 3 boolean tensors where the first channel contains activations for the current moving piece, the second channel includes activations for all the placed

pieces, and the third channel contains activations for all the empty cells. The size of this state space is $2^{H \times W \times 3}$, meaning that traditional MDP solvers like value iteration are out of the question for even small boards.

$$A = \{Left, Right, Down, Turn\}$$

Left and right move the current piece in the respective directions, down accelerates the piece's descent, and turning rotates the piece 90 degrees clockwise.

Transitions are deterministic, moving in units of 1 cell for an action direction and rotating 90 degrees for the rotate action. For the reward function, we took inspiration from [11] and used a custom reward function:

$$R(S) = c_1 \,\text{lines}^2 + \frac{c_2}{\text{holes} + 1} + \frac{c_3}{\text{height} + 1} \tag{1}$$

$$+ \frac{c_4}{\text{bumpiness} + 1} + c_5 \left(\frac{\sum_i \text{row}_i}{\text{board width}} \right)^{c_6} \tag{2}$$

$$+ c_7 \,\text{gameover} \tag{3}$$

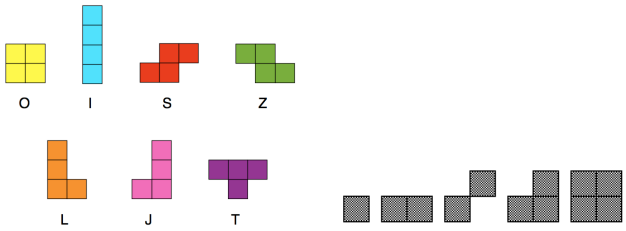
The most significant difference between our reward function and the reward function in [11] is that we only send a negative reward for a game over. Instead of punishing frequent occurrences of bad state attributes like height, holes, and bumpiness, we instead reward infrequent occurrences of bad state attributes by multiplying a positive reward against: $(bad\ attribute + 1)^{-1}$. The reason for doing this is with too many negative rewards the model learns to end the game as fast as possible. Additionally, to incentivize filling in rows, we add a reward term with a positive reward that scales with more partially filled rows.

Finally, we use a discount term:

$$\gamma = 0.99$$

To maximally weigh the importance of future states.

Unlike in normal Tetris, when the pieces are famously the O, I, S, Z, L, J, and T pieces we use the pieces from [6] in what we call mini Tetris.



(a) Original Tetris Pieces (b) Mini Tetris Pieces

IV. Results

We run one experiment in which we train four models for $4,194,304$ steps on a board that is 10 cells wide and 20 cells tall with the mini Tetris piece set. Each collector run collects 512 steps at time, our replay buffer holds 4096 steps and is sampled from using the temporal difference error as a priority, we use a learning rate of $1 * 10^{-4}$, a mini batch size of 32, and a learning intensity of $\frac{32*5}{512} = 0.31$. Every 5120 steps we evaluate the model by running 5 1000 step roll-outs and collect metrics on average reward, cumulative reward, number of steps, and lines cleared. We save the best performing model for each of these metrics and run a final evaluation where models are scored by the number of lines cleared in 32 1000 step rollouts.

We use the reward function

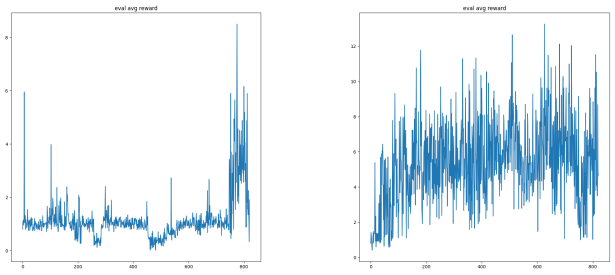
$$R(S) = 10 \,\text{lines}^2 + \frac{2}{\text{holes} + 1} + \frac{5}{\text{height} + 1} + \frac{2}{\text{bumpiness} + 1} + 10 \left(\frac{\sum_i \text{row}_i}{\text{board width}} \right)^4 - 100 \,\text{gameover} \tag{4}$$

All of these coefficients were found heuristically and certainly can be improved upon.

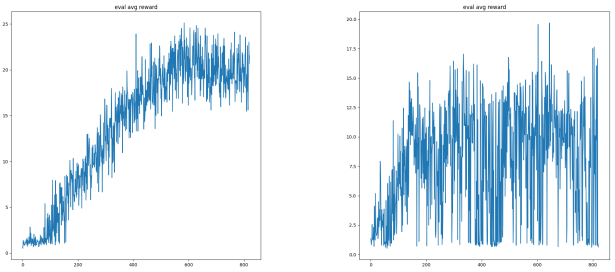
We evaluate 2 model architectures with PPO and DDQN. The first architecture is a convolutional neural network in which the state image $S = \{0,1\}^{20 \times 10 \times 3}$ is first convolved by 32 3×3 kernels and then convolved again by 64 3×3 kernels, before the output is flattened and passed through a series of 3 hidden layers with dimension 512 before a final output layer outputs values for each of our four actions. The second model architecture is a standard artificial neural network in which we directly flatten the state image into a vector $S = \{0,1\}^{600}$ before passing the the state vector to 3 hidden layers with dimension 512 before a final output layer outputs values for each of our four actions. The reason for selecting these architectures is to compare a direct representation, the flattened state, with a learned representation, the convolution output, for decision making for mini Tetris. The results of the experiment can be seen in the table below:

Training & Architecture	Lines Cleared
DDQN CNN	3
DDQN ANN	1
PPO CNN	7
PPO ANN	5

We also visualize the average reward per evaluation step in the figure below:



(a) DDQN ANN (b) DDQN CNN



(c) PPO ANN (d) PPO CNN

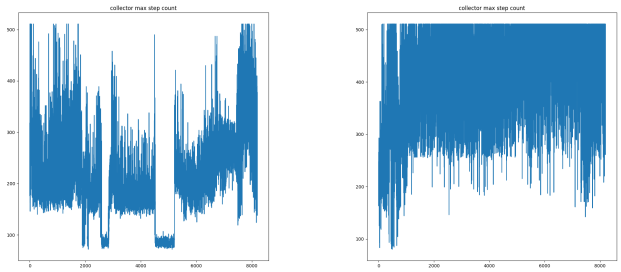
Fig. 3: Average Reward at each evaluation step for each model

From the graphs of average reward, it is evident that both stability and random reward pose challenges for our models. Additionally, from the table it is clear that the CNN models outperform their respective ANN models and the PPO models outperform their respective DDQN models. This is a surprising result, we initially tested the ANNs because we hypothesized a CNN on a normal Tetris display would need learn to "see" the state of each cell requiring more training time. This was not the case, however, as the CNN outperforms the ANN, indicating that the CNN must be learning to extract features beyond individual cell states. One possible explanation for this observation would be that the CNN, using 3×3 kernels, is spatially biased to learning relationships between neighboring nodes whereas the ANN contains no spatial bias making learning spacial relationships during training much more difficult. Another observation would be worth noting that the CNNs appear to be much more unstable than ANNs for learning Tetris likely because they are learning feature extraction that later dense layers must use.

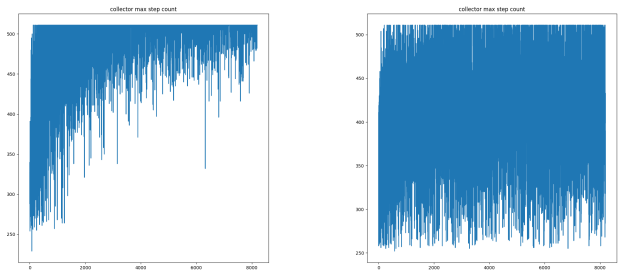
V. Conclusion & Future Work

This project was a moderate success because of the challenges with credit assignment sparse reward, and exploration learning to play Tetris from raw inputs is very difficult because of the challenges with credit assignment, sparse reward, and exploration learning to play Tetris from raw inputs is very difficult. Although, we ultimately achieved our goal of learning how to use TorchRL. By down-scaling from Tetris to Mini Tetris, we are able to train a model that clears a couple of lines before losing the game. We are constrained by both time and compute to push this implementation to its limits. Further research would include ablation studies of our reward function to really understand what each component is contributing and delve into the reward coefficients $c_1 ... c_7$. Further research would also train larger networks with more advanced reinforcement learning techniques as well as perform a hyperparameter search instead of guessing reasonable hyperparameters. We believe a system that performs well on mini Tetris would perform well on regular Tetris, given enough training time and computing power.

One collected graph particularly sheds light onto our compute bottleneck. We can see the max step from each collection epoch and observe that all models except the DDQN ANN eventually terminate the episode by truncation, not by ending the game, we theorize that this leads to poor performance in late game states since they are under represented during training.



(a) DDQN ANN (b) DDQN CNN



(c) PPO ANN (d) PPO CNN

Fig. 4: Number of steps per episode

With more compute and time we would not have to truncate episodes and allow our models to learn more about playing in a late game.

In order to learn effective board representations, it could be useful to learn an auto encoder for boards in which a UNet or vanilla auto encoder could learn a latent space in which the board can be fully recovered. Another novel approach would be to learn an image and action auto encoder together so when the embeddings for the action and image are added together and the resulting vector is decoded the next board is produced. This learned embedding would likely work very well as a state representation as it would represent the current board

and encode the possible future boards all in one vector. Finally, compute could be more effectively used training could likely be speed up by making use of torchrl's multi-process collectors and trainers instead of training on a single process and GPU.

VI. Contributions and Release

Max Conway built the infrastructure and pluming that organized the project, created the network architectures, data collectors, and replay buffers, as well as vibe coded the environment with GPTo3, and wrote the PPO training loop. Gyanig Kumar wrote the DQN training loop and researched the TorchRL provided objects. Both authors worked on the writeup.

The authors grant permission for this report to be posted publicly.

References

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

[2] Albert Bou, Matteo Bettini, Sebastian Dittert, Vikash Kumar, Shagun Sodhani, Xiaomeng Yang, Gianni De Fabritiis, and Vincent Moens. Torchrl: A data-driven decision-making library for pytorch, 2023.

[3] Jiajun Fan. A review for deep reinforcement learning in atari:benchmarks, challenges, and solutions, 2023.

[4] Jiajun Fan, Yuzheng Zhuang, Yuecheng Liu, Jianye Hao, Bin Wang, Jiangcheng Zhu, Hao Wang, and Shu-Tao Xia. Learnable behavior control: Breaking atari human world records via sample-efficient behavior selection, 2023.

[5] Rex L. Reinforcement learning on tetris. https://rex-l.medium.com/reinforcement-learning-on-tetris-707f75716c37, June 2021. Medium; 9 min read. Accessed: May 2, 2025.

[6] S. Melax. Reinforcement learning tetris example. https://www.melax.com/tetris/tetris.html. Accessed: May 2, 2025.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[8] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[9] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

[10] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[11] Matt Stevens and Sabeek Pradhan. Playing tetris with deep reinforcement learning. Course project report, Stanford University, CS231n Course Reports, 2016. Accessed: May 2, 2025.

[12] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.