Reinforcement Learning in Two-Player Tetris

Sameer Pai
`sampai@mit.edu`

Abstract

Two-Player Tetris is a competitive variant of the classic arcade game in which players attempt to defeat each other by clearing lines as fast as possible. In this report we present a formulation of the Two-Player Tetris environment, and propose a method for training a controller to play this game. Specifically, we present a two-stage curriculum-based training schedule exploiting limited interaction between players which, despite limited availability of compute power, was able to perform at a similar level as the median active online human player in terms of piece efficiency.

1 Introduction

Tetris is a classic arcade game created in 1984 in which players attempt to stack falling pieces (also known as *tetrominoes*) in order to clear lines. It quickly rose to prominence as one of the best-selling video game franchises of all time, due to its simple problem formulation and controls yet surprisingly complex strategy. However, after some practice, even novice players of modern Tetris games are capable of playing indefinitely without losing the game. Versus Tetris is a multiplayer variant of Tetris in which two players attempt to *top eachother out* by clearing lines as fast as possible (precise formulation to follow). This variant has enjoyed significant success, with the most popular platform amassing over one million players globally. Two-Player Tetris is known for an incredibly deep level of strategy, requiring a much wider array of skillsets than the original game.

There has been significant success in applying reinforcement learning methods to single-player Tetris, with the most successful models being capable of playing the game indefinitely [10, 9]. However, even for humans, simply being capable of playing indefinitely is considered to be only the first step in achieving Tetris mastery. Two-Player Tetris presents a formulation in which models can truly demonstrate their ability in the Tetris environment. The success of a learned controller in this domain would demonstrate yet another success of reinforcement learning methods in gaining mastery over a complex and open-ended task with long horizons.

The model to be trained in this environment will be a deep neural network that takes as input a description of the current board state, s, and outputs an estimate of the discounted reward $V_\theta(s)$. To choose an action, the environment provides the agent all possible states after the next action, and it chooses the one which provides the highest predicted reward. The primary algorithm for training will be Deep Q-Learning (although in this case we are learning a value function of a state, rather than a Q function of a state-action pair). To enable the model to perform well in the multiplayer environment, we propose a curriculum-based method. Specifically, we first teach the model to play Tetris without an adversary, and then gradually start simulating an adversary at increasing levels of difficulty. In doing this, we can train our model to be able to succeed in an adversarial environment without having to utilize computationally intensive methods such as self-play.

6.484 Computational Sensorimotor Learning Final Project (Spring 2022).

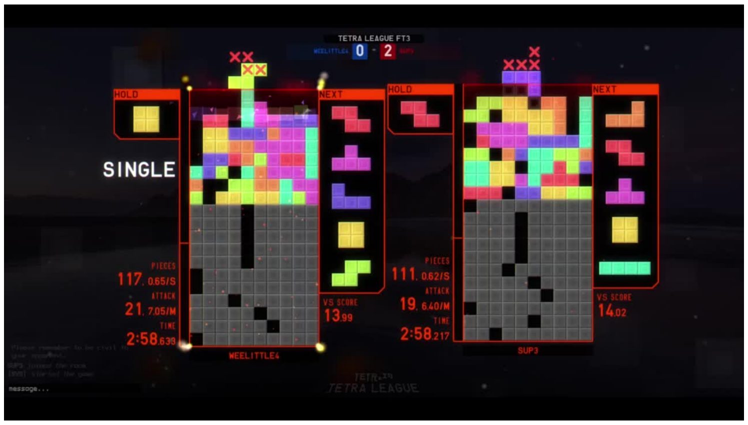

Figure 1: Example of Versus Tetris gameplay (taken from TETR.io). The grey lines at the bottom are *garbage*, which players can send to each other by rapidly clearing lines.

2 Related Work

As mentioned in the introduction, there has been significant progress in reinforcement learning models in Tetris. The first significant result was the use of the cross-entropy method on hand-crafted heuristic functions to achieve an agent that could survive for extended periods [10]. There has also been some success in using Temporal Difference learning on an extremely compressed state space [2]. However, existing models focus only on survival, and play extremely conservatively, opting for single-line clears and staying extremely low. These behaviors would be unsuccessful in a versus environment, as the agent would fail to send any lines. A successful agent in the versus environment would need to play more aggressively, waiting until it can clear more lines simultaneously.

Deep Q-learning has proven to be successful in a wide variety of environments, such as Atari games and Mario [5, 4]. This provides evidence that this method can be readily applied to Tetris, with proper reward design and hyperparameter tuning.

Curriculum-based learning schedules have also been found to greatly speed up learning [6]. For example, West et. al. showed that a curriculum that focused first on a reduced subset of board game states and then widened to the whole game sped up early learning of an agent [11]. Inspired by this, we seek to apply a curriculum to help our Tetris agents adjust to increasingly adverse environments.

Finally, there are many examples of existing versus Tetris AI that do not leverage reinforcement learning, instead opting for methods from planning. The current state of the art is ColdClear, which uses human-crafted heuristics in tandem with a Monte-Carlo Tree Search variant in order to select the best action given the next 6 piece previews [3].

3 Problem Formulation and Methods

3.1 Basic Gameplay

In Two-Player Tetris, each of the players is given a standard Tetris board (20 rows by 10 columns). When either player clears lines on their board, a corresponding amount of *garbage* is sent to the other player. Garbage consists of lines with one missing square that are appended to the bottom of the Tetris board (see Figure 1; the grey lines are garbage). Players lose when their board becomes too tall to place their next piece.

The amount of garbage sent to an opponent depends on a number of factors. First of all, clearing 2, 3, or 4 lines with one piece sends 1, 2, and 4 lines to the opponent, respectively. An additive bonus is applied to this count by *T-Spins*, a special move where a player spins the T piece into a tight space, *combos* of consecutive line clears, and *back to back* sequences of multiple T-Spins or 4-line clears. These last three factors form the majority of complexity in Two-Player Tetris gameplay.

At any point in time, the player is able to see the next five upcoming pieces, as well as the number of incoming garbage lines. High-level human players will use both of these to plan out their strategies.

3.2 Observation Space

3.2.1 Full Observation

Two-Player Tetris is a complete-information game, meaning it has identical observation and state spaces. An observation consists of a 20 by 10 matrix of board values (1 if the square is occupied, and 0 otherwise). The observation also contains a 5 by 7 matrix P representing the preview pieces, where P_{ij} is 1 if the i-th next piece is tetromino j, and 0 otherwise. Similarly, there will be a vectors c, h of 7 values, where c_i, h_i are 1 if the current piece (or held piece, respectively) is tetromino i and 0 otherwise. Finally, we will have a 20 by 10 matrix whose values are one if the current piece occupies that square and zero otherwise.

If we were to flatten and concatenate the observation, we would get a feature vector of length $2 \cdot 20 \cdot 10 + 5 \cdot 7 + 7 + 7 = 449$.

3.2.2 Simplified Observation

The full observation space of Tetris proved to be quite large, and even using convolutional layers to bring down model complexity, it was necessary to develop an abbreviated observation space. We were motivated by the fact that, as a general rule of thumb, Tetris players value two attributes of their board space:

- *Stacking flat*, i.e. keeping the board height roughly constant.
- *Stacking clean*, i.e. minimizing the number of holes present on the board.

A simple observation space that can capture both of these ideas is to have the observation be simply the height of each column of the board, and the total number of holes on the board. There are 10 columns, so this extremely simplified observation space has only eleven features (including the hole count). We can further augment this space with more heuristics to trade-off training time and the amount of information that the agent can see. Some potential augmentations considered were:

- The position of the topmost hole in each column
- The identities of the future piece previews

3.3 Action Space

We considered two models for actions: *button presses* and *compressed actions*.

3.3.1 Button Press Action

Under this action space, there are nine actions the agent can take, corresponding to the possible button inputs in an actual game of Tetris:

- *Max shift left/right*: Move the piece maximally to the left / right (in game this is done by holding the left / right key).
- *Single move left/right*: Shift the piece one to the left / right if possible.
- *Rotate left/right*: Rotate the piece (counter-)clockwise.
- *Hard drop*: Move the piece maximally down and then place it (i.e. queue the next piece).
- *Soft drop*: Move the piece maximally down but do not place it.
- *Hold piece*: Swap the current piece and the held piece.

This forms the complete action space of the agent. This is a more realistic action space, as the agent has access to the same inputs as a human player would. Furthermore, this is significantly easier to integrate into existing Tetris games than more complex action spaces. However, this space is extremely hard to train on, due to the fact that it has incredibly long horizons, and the agent can take infinitely many steps before even placing a piece. Furthermore, in order for DQN to succeed on this action space, the observation would have to include position and rotation information about the current piece, and model complexity would have to be much higher to leverage this information.

3.3.2 Compressed Actions

With compressed actions, we precompute all possible ways to place the following piece, by performing a breadth-first search over the above button presses, terminating at the first hard drop. Then, all the agent has to do is choose out of a list of possible future states. The fact that the size of the action space is not always constant may pose an issue for algorithms such as Proximal Policy Optimization (PPO) and the usual Q-Learning, but since our model is only learning a value function on a single state, this is a viable action paradigm. Furthermore, this significantly reduces horizons by forcing the agent to place a piece every action.

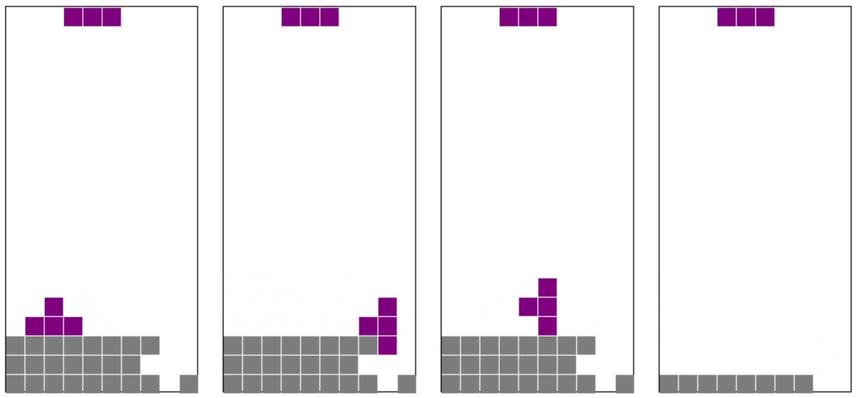
Figure 2: Four possible compressed actions with a T-piece. The last action involves a *T-spin*, and clears two of the garbage lines.

3.4 Learning Algorithm and Reward Design

The value function estimator will consist of a deep neural network which takes as input the observation of the current board state. We will learn a policy using standard deep Q-learning, as described in the algorithm box below. As a reward, we attempt to alleviate the long-horizon issue by providing a constant reward $(+1)$ with every piece placed, as well as a large negative reward (-20) if the agent loses the game. In the short term, this rewards surviving for as long as possible, both to maximize the piece placement reward and minimize the loss penalty via discounting. After the agent learns to play consistently, the per-piece reward will become obsolete, as over a constant horizon the piece placement reward is constant. On top of this survival-oriented reward, we add an *attack reward*, which values clearing lines. The specifics of the attack reward are based on the official Tetris guideline and are beyond the scope of this report, but in general clearing more lines at once will result in a quadratically growing reward, incentivizing simultaneous clears over single-line clears.

The first training step will involve training our model in an environment that simulates single-player Tetris, without an adversary. This stage of training will continue until the agent is able to consistently reach the episode length cap, at which point we can be confident that it can play indefinitely in the single-player setting. After this, we would like to simulate an opponent for our agent to learn to survive in a versus environment. One computationally easy way to do this is to exploit the fact that the only way players in Versus Tetris interact with each other is via sending garbage to each other. Therefore, if we can model the pattern of lines sent by the opponent, then we don't actually have to fully simulate the opponent as an agent. A simple first-order approximation would be a Poisson process with mean *attack rate* λ to represent the number of lines sent by the opponent. We can then form a curriculum for our agent by starting with a low value of λ, and increasing it as the agent adjusts to the increasing rate of garbage being sent to it. This curriculum method should train the agent to deal with being attacked, a significant component of Versus Tetris play.

Algorithm 1 Value Function Learning Algorithm

```
$\epsilon, \lambda, \alpha \leftarrow$ hyperparameters
initialize a cyclic experience buffer
$\theta \leftarrow$ randomly initialized
$\theta_t \leftarrow \theta$
for $0 \leq i < MAX\_EPISODES$ do
    $s \leftarrow$ initial state
    for $0 \leq j < EPISODE\_LEN$ do
        $T \leftarrow$ set of possible future states
        $s' \leftarrow \arg\max_{s' \in T} R(s, s') + \gamma \cdot V_\theta(s')$
        add $(s, s', R(s, s'))$ to the experience buffer
        step to $s'$ with probability $1 - \epsilon$ or randomly otherwise
        add garbage to the current state sampled according to $\lambda$

        sample a batch $S, S', R'$ from the experience buffer
        compute loss $L = \mathcal{L}_{Huber}(V_\theta(S), R' + \gamma \cdot V_{\theta_t}(S'))$
        perform a gradient descent step on $\theta$ minimizing $L$
        $\theta_t \leftarrow \theta_t \cdot \alpha + \theta \cdot (1 - \alpha)$
    end for
end for
```

4 Results

4.1 Model Environment

We were unable to find a Python-based simulator for Tetris that was compliant with Tetris standard guidelines (more specifically, most were non-compliant with the modern piece randomization and rotation rules). Therefore, we wrote our own simulator in Python, and wrapped it in an OpenAI Gym environment for training. Writing the simulation ourselves enables easy configuration for the addition of garbage to facilitate the curriculum approach above, as well as the action compression paradigm.

4.2 Training

All training was conducted on a personal computer with a NVIDIA 1660 Super GPU (6GB VRAM).

Our first attempt at training a model involved using the full observation format described above, with the value estimator being implemented as a convolutional network over the board matrix followed by multiple fully-connected linear layers. After multiple days of training, the model failed to make significant progress in the single-player environment, and so training was stopped. At this point, the decision was made to pivot to the simplified observation format, as this would almost certainly be significantly less computationally intensive, and also quicker to converge.

The value function estimator for the simplified format was implemented as an MLP network with two hidden layers mapping the observation to a single scalar representing the estimated value. Unsurprisingly, this model converged orders of magnitude quicker, with the model being capable of playing single-player Tetris indefinitely after only 1500 epochs.

We then transitioned into the simulated-opponent phase of training, starting with a small attack rate of $\lambda = 0.01$, and increased this value by 0.01 every 100 episodes, up to a maximum of $\lambda = 0.07$, after which the agent was unable to survive consistently regardless of how long it was trained.

We also experimented with disallowing the use of the *hold* feature during stage 1 of training. Even without the ability to hold pieces, the agent was still able to achieve indefinite survival in stage 1 of training. However, it was unable to adapt to the simulated opponent. This suggests that the ability to hold significantly improves the performance of the agent in the versus setting, more so than in the single-player setting. Figure 3 shows the return curves for both agents during both stages of training.

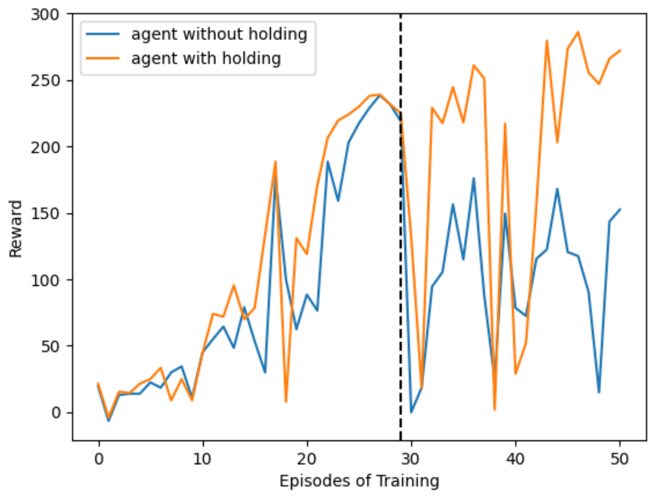

Figure 3: Return curves for both the agents with and without holding. The black dotted line represents the transition between the single-player environment and the curriculum stage. We observe that both agents are capable of performing roughly equally as well in the single-player case, but when the simulated opponent is added, only the agent with holding is able to recover its original performance (and actually performs better with the opponent than without).

4.3 Evaluation

As described earlier, the most successful model was the one with the simplest observation space, which has access to the hold feature. In this section we will first provide a qualitative description of the model behavior, and then some objective metrics to measure its performance in the versus environment.

Gameplay samples of the agent can be found here (with $\lambda = 0.07$) and here (with $\lambda = 0.1$). Qualitatively, the agent plays in a very human-like fashion, stacking generally flat and clean in most cases. Furthermore, it seems to adopt some of the high-level strategies used by beginning to intermediate-level human players. For example, the agent frequently places pieces cleanly in only nine out of the ten columns, leaving a "well" in which the I piece (the long bar) can be placed for a large attack reward. However, there are also some decisions made by the agent that were non-ideal, such as placing pieces above its well, blocking the opportunity for future rewards. Figures 4 and 5 show emergent behaviors of the agent that resemble human play.

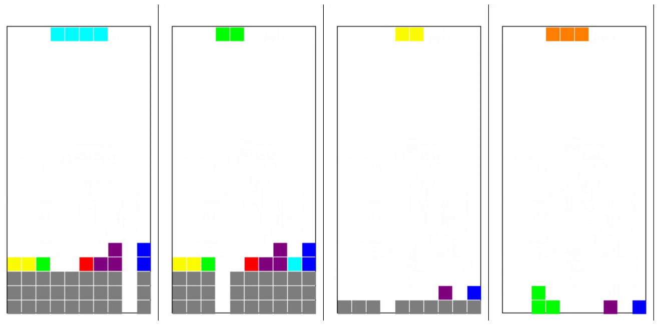

Figure 4: An example of *downstacking*, where the agent performs successive line clears to get to the bottom of the board, rather than waiting to be able to get a 4-line clear. This is an important emergent behavior because the agent needs to recognize when it is at risk of being topped out by its opponent, and play defensively when necessary.

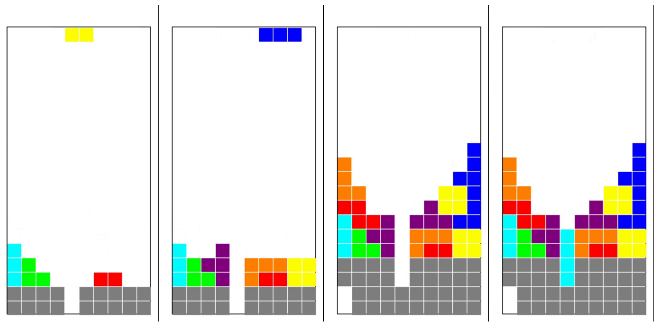

Figure 5: A technique called *upstacking*, where the agent places pieces around a well so that it can eventually get a 4-line clear for maximum reward.

To gain a more objective measure of the performance of our model, we attempt to benchmark the agent against a human baseline. In order to do this, we can use the player ranking system on TETR.io, the most popular online versus Tetris platform with over one million active users. The website uses a *matchmaking rating* (MMR) system to rank its players based on relative skill, and publishes data on individual player statistics. In order to compare our model to a human player, we need to account for the unfair mechanical advantage that a computer has against a human. A fair metric for this is *attack per piece (APP)*, or the amount of lines of garbage sent per piece placed. Table 1 shows the APP of our trained models, as well as the APP of human players at various skill levels. Our model is able to meet the performance of the median active human player by this metric.

Agent / Player	Attack per Piece	Equivalent λ value
DQN without Hold	0.210	0.084
DQN + Hold	0.285	0.114
Median TETR.io Player	0.286	0.114
Top TETR.io Player	0.864	0.346

Table 1: Attack per piece (APP) of both of our trained agents, as well as the median and top player on the TETR.io platform. Our model with hold is capable of meeting the performance of the median player, but a significant amount of improvement is necessary to reach top human performance. Also shown are the equivalent values of λ necessary to simulate an opponent with this APP value.

Another metric for performance evaluation is by the ability of the agent to tolerate incoming garbage. The attack value of $\lambda = 0.07$ that our best-performing model capped out at corresponds to a simulated opponent with an APP of roughly 0.18, implying that our agent is able to consistently survive against an opponent with this APP. For comparison, the median human player would correspond to a simulated opponent with $\lambda = 0.1$.

5 Unexpected Issues

The main limit in this project was training time. Models took much longer than expected to converge, forcing us to greatly simplify both the model architecture and the observation space. The author also contracted COVID-19 in the weeks preceding the project due date, resulting in a lack of time to refine and iterate further on the model.

6 Conclusion and Further Work

In summary, we were able to use Deep Q-learning, along with heavy compression of both the observation and action spaces, to train a Tetris-playing agent that is capable of surviving in the Versus environment. Furthermore, in terms of piece efficiency, our agent was able to match the performance of the median active human player, and exhibits emergent behaviors that resemble human play in many aspects.

However, this model is not without its shortcomings. First of all, simply due to the structure of the observation, the agent is unable to see the exact location of holes in the board. This makes it hard to fix mistakes, as the agent will frequently cover up its well and holes (as described above). Also, our agent never learned to exploit higher-level techniques such as T-spins and combos, which are indispensable at high levels of play.

Additionally, by the nature of the value function estimation, the agent is very intolerant of changes in reward design, i.e. if we change the weights of parts of the reward, then we essentially have to retrain from scratch, as the value function of states change significantly. This makes it difficult to incentivize the model to take advantage of different techniques without breaking its ability to maintain basic survival. One possible way to alleviate this is to utilize behavior cloning to transfer the behavior of a pre-trained model onto one with a new reward scheme, which might be able to refine and learn new techniques.

There were many different avenues for future work that were considered but not pursued due to lack of time and compute. The most obvious improvement on this model would be to increase the observation space and model architecture. With more time, and possibly some hyperparameter tuning and reward design, it is likely that we can achieve a model that can achieve better piece efficiency.

Additionally, other learning algorithms were considered. One possibility is self-play, which has proven to be very successful in other competitive 2-player game environments [1]. However, self-play requires large compute and high levels of parallelism to train efficiently, which we did not have access to. Another possibility was Monte-Carlo Tree Search augmented with a neural value and policy estimator, in the vein of DeepMind's AlphaZero and AlphaGo [8, 7]. However, MCTS has already proven to be a very successful planning method on Tetris, using hard-coded heuristics [3], and so we felt it was unlikely that a neural network would improve on this method.

7 Author Contribution

Since this is a single-author project, Sameer is responsible for the entirety of the project.

8 Appendix

All code written for this project can be found at https://github.com/sampai20/tetris-rl, including the simulation environment and wrapper, training algorithm, and rendering code.

References

[1] Trapit Bansal et al. *Emergent Complexity via Multi-Agent Competition*. 2017. DOI: 10.48550/ARXIV.1710.03748. URL: https://arxiv.org/abs/1710.03748.

[2] Donald Carr. "Applying reinforcement learning to Tetris". In: *Department of Computer Science Rhodes University* (2005).

[3] Mark Karlson. *MINUSKELVIN/cold-clear: Tetris Bot*. URL: https://github.com/MinusKelvin/cold-clear.

[4] Sean Klein. "CS229 Final Report Deep Q-Learning to Play Mario". In: ().

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

[6] Sanmit Narvekar et al. "Curriculum Learning for Reinforcement Learning Domains: A Framework and Survey". In: (2020). DOI: 10.48550/ARXIV.2003.04960. URL: https://arxiv.org/abs/2003.04960.

[7] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. DOI: 10.48550/ARXIV.1712.01815. URL: https://arxiv.org/abs/1712.01815.

[8] David Silver et al. "Mastering the game of go without human knowledge". In: *nature* 550.7676 (2017), pp. 354–359.

[9] Matt Stevens and Sabeek Pradhan. *Playing tetris with deep reinforcement learning*. 2016.

[10] István Szita and András Lörincz. "Learning Tetris using the noisy cross-entropy method". In: *Neural computation* 18.12 (2006), pp. 2936–2941.

[11] Joseph West et al. “Improved reinforcement learning with curriculum”. In: *Expert Systems with Applications* 158 (2020), p. 113515.