Tetris via Reinforcement Learning: A Lesson in Simplicity

Mustafa Eyceoz
Dept. of Computer Science
Columbia University
New York City, USA
me2680@columbia.edu

GitHub Repo Link
github.com/
maxusmusti/tetris-ppo

***Abstract*—Learning to play Tetris via Reinforcement Learning algorithms has been attempted a number times, but almost always using value-based methods, like Deep Q-Learning, or DQN. This paper attempts to explore the possibility of using policy-gradient approaches to reinforcement learning, like actor-critic and PPO, in an attempt replicate or improve upon the value-based results. Specifically focusing on classic NES Tetris, the paper analyzes the method, results, and shortcomings of policy-based methods that ultimately explain why a value-based approach better suits the task.**

***Index Terms*—Reinforcement learning, PPO, actor-critic, Tetris, policy, Q-learning, CNN**

I. Introduction

As the jump from value-based reinforcement learning methods like Deep Q Learning and DQN to policy-based methods like PPO occurred, the games that people pushed to solve also jumped in difficulty. From board games and single-player arcade games to complex, real-time, multiplayer competitive titles like DOTA, new policy gradient methods that fixed previous shortcomings took the reinforcement learning field by storm, and immediately people jumped to bigger and better. It may be appropriate, however, to take a step back, and compare apples to apples. Specifically, going back to a less complex, yet still challenging problem, like Tetris, it would be valuable to see how PPO performs in the retro space. Even more specifically, though, to conduct this experiment, we will focus on classic NES Tetris. Is PPO the best solution even in smaller, more discrete spaces, or are policy-gradient methods still really only useful in larger, continuous action spaces?

Tetris is a game that has been attempted many times with deep reinforcement learning, but with two major caveats. First, most projects focus on modern Tetris as opposed to classic Tetris. Second, most projects stick to Deep Q Learning or other value-based methods for their final approach. In fact, almost every successful Tetris RL project has been for modern Tetris, a speed-focused game with hard-drops, piece-holding, N-piece foresight, and meaningless fall speed. Some projects even take it a step further and focus on multi-player battle Tetris, where certain combos hold more value due to the disadvantages they create for opponents. Almost all of these projects stick to Deep Q Learning, with any venture into policy-based methods or PPO only being as a surface-level environment test. Furthermore, while there are a few personal or student projects that exist for building or working in a classic Tetris RL environment, the focus remains primarily on DQN, or search-and-heuristic approaches. While a couple policy-gradient-based approaches existed, they paled in comparison to the SotA for classic Tetris, which is a search-and-heuristic approach [1]! Thus, the goal is to better understand whether there is potential for improvement via PPO, or whether there is a reason most people shy away from it in the Tetris space.

II. Environment

There are three critical components to any reinforcement learning environment: the state-space (or observation-space), the action-space, and the reward. For this experiment, the specific environment used is gym-tetris [2], a classic NES Tetris Reinforcement Learning environment, built with OpenAI Gym.

The state-space is incredibly straightforward. Each state provided by the environment is simply a raw image of the entire game, featuring menu, board, and all. The image dimensions are (240v, 256h, 3c). Additionally, an ”additional info” dictionary is provided, containing the current score, number of lines cleared throughout the game, the current piece name, the next piece name, and drop statistics (all of which can be seen on the screen as well).

The environment has two options for action-spaces. A ”full legal” action-space, which includes multiple simultaneous inputs per step (for a total of 26 actions), and a ”simple” action state, which consists of one action per step. Note that a ”step” in tetris does not necessarily mean the piece falls down. In fact, at level zero, five steps pass before a piece naturally shifts one block down, and this number shrinks as the level increases. For this experiment, to make things more condensed and learn-able given the complex nature of the desired approach, the ”simple” action-space is used, consisting of all legal NES Tetris actions: ”rotate right”, ”rotate left”, ”move right”, ”move left”, ”move down”, and ”do nothing”.

As this paper progresses, a number of reward functions will be attempted, though the default reward retrieved from the environment itself is simply the score increase per step.

III. Method and Process

A. PPO: A Brief Overview

Proximal Policy Optimization, or PPO, is a policy-based method for deep reinforcement learning. While policy-based methods usually train slower due to direct policy learning, tend to be more fragile in training due to cascading failures based on poor previous policies, and used to be reserved for continuous action space problems (as value-based methods would not work there), PPO manages to provide solutions to these problems. This allows a more direct policy-based approach to leapfrog the performance of value function learning/approximation methods, and PPO has taken over as SotA in many reinforcement learning problems. As a variation of the advantage actor-critic policy-gradient algorithm, and a direct descendant of TRPO, PPO ensures that there are no large differences between old and new policies, and uses Generalized Advantage Estimation (GAE) for advantage computation. The method is essentially:

```
• For each iteration 1 to K
  • For each actor 1 to N
    • Run the old policy in an environment for T timesteps
    • Compute advantage estimates for each timestep 1 to T
  • Compute the policy update using a surrogate loss constructed w/ the NT timesteps of data
  • Update policy (within permitted change)
  • Update critic for advantage estimation
```

Using this method, many games have been "solved", or seen significant AI improvements. Most notably in recent history was OpenAI's Dota 2 AI, which managed to compete at a professional level in a game far more complex than ever before attempted.

B. State Preprocessing

There is a bit of work to be done before actually passing our retrieved state into any model. First, there is only one part of the screen that actually matters: the board. Thus, the initial step taken is to simply crop the state image down to only the board, resulting in a new image with dimensions (160v, 80h, 3c).

Next, the color of the blocks actually means nothing for game strategy, so with a black background, the next step taken is to convert the state image to grayscale. This results in the image dimensions becoming (160v, 80h, 1c). Finally, this can be taken a step further, as once a block is placed, there is no need to distinguish it from other blocks. There is a mass of blocks that have been placed, and a current block not yet placed. Therefore, we convert the grayscale image into simply black-and-white, with any non-zero colors (zero being black) being set to one (one being white). With that, the initial state image is ready.

This, however, is not the entirety of the state. Also created and provided at each step is a feature vector that is also passed into the actor and critic models as part of the state (the models eventually concatenate the inputs, more on this later). This feature vector is comprised of: [current piece (mapped to int), next piece (mapped to int), score, lines cleared, and current max board height]. This covers all of the important details that may be lost when cropping the state image, and results in a comprehensive state.

C. Initial Approach and Issues

Initially, when designing the actor and critic models for the PPO agent, the idea was to pass the image directly into two or three convolution layers with relu activation, each followed by max pooling layers. The result would then get flattened, concatenated with the feature vector input, and passed into feed-forward dense layers. When attempting this, however, a number of issues arose that forced a rethinking of the model design.

Even when trying to minimize model parameters with this architecture, each of the actor and critic models had about 300,000 parameters (and before minimizing had about three million each). This is an absurdly large number for a problem that has a state of essentially a 20x10 grid and a vector of length five, alongside an action-space with a size of six. Even if the models are capable of reasonably learning (both theory and experimentation indicate otherwise), both their massive sizes and the incredibly daunting set of state "possibilities" that a 160x80 image provides makes learning take an incredibly long and unreasonable amount of time. The possibilities are also misleading, as once again, the only real possible states are those that could be a 20x10 image scaled-up. Any other 160x80 black-and-white image would be an invalid state.

D. Actor-Critic: Updated

To solve these problems, "computer vision" portion of the actor and critic models were altered in order to better suit the task. See Fig. 1 for the first nine layers of the updated architecture.

The inputs remain constant: the 180x60 black-and-white board image, and the feature vector of length five. Since the state image is really just a blown-up 20x10 image, the first layer converts the original image to exactly that, using an (8,8) max-pooling layer with a stride of (8,8). The minimized image then gets passed into two layers, one of which being a flatten layer to become a size 200 grid vector (consider this our new input 1, with input 2 being the size 5 feature vector). The second layer that the 20x10 image is passed into is a convolution layer of kernel (3,3) and a single filter with relu activation. This is done in order to extract additional potentially-useful features from the image (holes, bumps, shapes/patterns), with the 3x3 kernel being chosen to match potential piece locations and possible spaces. The output of this layer (18x8) is then passed into a flatten layer for a size 144 vector output (consider these new features as input 3).

With input 1 (size 200), input 2 (size 5), and input 3 (size 144), the three inputs are then all passed into a concatenate layer to produce a single feature vector of size 349. This vector

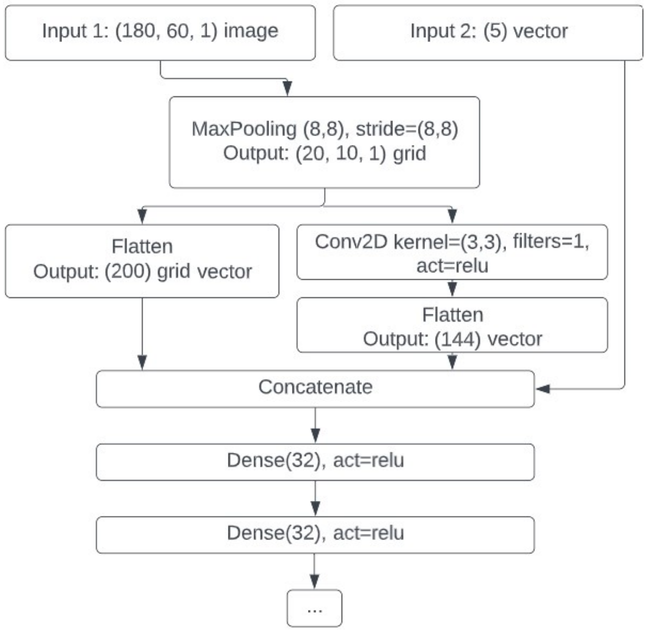

Fig. 1. Common Model Architecture

is then passed into two feed-forward dense layers of size 32 with relu activation.

At this point, the actor and critic models diverge. The critic (value estimation for advantage) model has an additional dense layer of size one (single value output). The model then uses mean-squared-error for loss. The actor (policy estimation) model, on the other hand, has a dense layer of size six (action space size), with a log softmax activation for action selection, and uses the PPO objective for loss, using advantage and prediction inputs. Each model has approximately 12,000 parameters.

IV. Process Cont.: Problems and Solutions

A. Guided Learning

The first major issue encountered with this method and architecture was a probability oversight. It turns out, when playing randomly, the odds of ever clearing even a single line in classic Tetris are low. Not like 1% low, or 0.1% low, after over 10,000 Tetris games played randomly, not a single line was cleared. In reinforcement learning, a balance must be set between exploration (random movement) and exploitation (greedy or policy-based decision making). When starting to train a model, having a decaying epsilon probability of randomness in move selection allows the model to be exposed to new actions, states, and rewards, rather than getting stuck on an initial decision and closing off paths before learning anything. The issue here, however, was that there was nothing to be learned from random play. No reward was ever gained. Even with this giant, complex system, no reward meant no learning. Without reinforcement, what was there to distinguish desirable actions and states from the rest?

To solve this, a guided learning approach was adopted. Instead of making random decisions at first, the environment was modified such that a skilled human would play portions of the game to demonstrate good moves and gain meaningful rewards. Similar to the process for training self-driving cars, the Tetris agent would see the reward gain from the human action and attempt to replicate the results. While this did succeed in making the system learn, it was not successful at reproducing similar results. The learning was shallow, with only two key rules seemingly picked up: drop pieces faster to get the score bonuses, and push pieces towards one edge of the board (the left side, which is actually better) to stack up. Otherwise, how to fit pieces together, when to rotate, and any other meaningful rules were ignored. Despite progress, not a line had yet to be cleared.

B. Heuristics and Reward

To make the system learn to make good decisions, it had to be guided towards and rewarded for those decisions. Telling it that ”scoring points is good” without giving a hint as to how points can be scored is as good as handing a baby a pencil and paper and telling them to learn math, giving them a point for every correct conclusion. Thus, two heuristics were designed to help guide the system. First, given a state, the number of ”holes”, or inaccessible empty spaces, were counted. Next, the number of ”bumps”, or blocks sticking out of a stack, were counted. As both of these are undesirable in a clean, optimal game of Tetris, and are considered as the fundamentals of gameplay, these were the heuristics chosen to improve the system. Added as penalties to the reward, the agent gained reward by removing holes and bumps from the state, and lost reward by creating holes and bumps. Additionally, a third heuristic useful for optimal stacking is maximum board height (the highest placed block on the board). The agent would be penalized for increasing the board height, and rewarded upon decreasing the board height.

V. Analysis and Conclusion

A. Policy Failure

The experiment had everything: a sound architecture, a flawless environment, solid learning techniques, heuristic guidance, and the result that it all yielded: a few cleared lines. That is all. The agent was able to get at most 2-3 cleared lines in a game, with a score maybe breaking 100. This is a far cry from the results seen with value-based methods, and even heuristic-search approaches. So what went wrong? It all boils down to the core underlying concept behind the experiment: policy-based approaches are not one-size-fits-all.

Thinking about it further, a policy-based approach simply does not make sense for a game like Tetris. For each new piece, there are many, many routes to the same outcome (where it is ultimately placed). No actions have meaning in any order for a given state. How the piece gets to the bottom does not matter, all that matters is where it lands. Seeing a state and thinking ”move left” really means nothing if not playing on a very high level with meaningfully fast drop speeds. Really, ”actions” themselves do not matter when trying to play Tetris well. They are nothing but a mindless means to an end. When convincing a friend to meet you at the mall, you are not concerned with how they walk, only that they make it to the right place in time. Furthermore, the state does not even always change given the action. As mentioned earlier, one action does not mean falling down one block. This method of attempting to play the game is so inefficient and lacking in understanding. Disregarding the incredibly untouchable state-action space one would need to learn (which could only be trained through feeding a model millions of perfectly played games), it is fundamentally the wrong way to frame the game.

B. *Value: Simple is Better*

So if policy-based approaches are so bad, what makes value-based methods so much better? Well, let us start with how any human plays Tetris. When a new piece appears at the top, do we look at the state and say "I should move to the left", and then look at the state again and say "I should move to the left", and then say "I should rotate now", and then look again and say "I should go down"? Not at all. When a new piece is introduced, we look at the current board and say "where would this fit". We map the piece onto the stack in all the ways we can think to reasonably fit the piece, then pick the end location we think is optimal. Then we move the piece to that final place, in a sequence of actions that exists only to achieve the envisioned result. This is Tetris. Given a piece and a stack, where should the piece be fit into the stack. Given a current state and a new piece, what is the best next state, out of all of the possible places it can land. Not all of the places it can be. We do not think about placing the piece in the air at some random location first. Given a "state" (current stack), what is the best "state" (stack with one new piece). There are hundreds of routes to achieve this state, but the route means nothing. This time, it is not about the journey at all, only the destination.

This is exactly how a value-based method works. Every successful Tetris reinforcement learning project works this way [3] [4]. Given the board state, the set of next states is not where a new piece can be moved once, it is simply all of the places that piece can end up. No longer dealing with any placement in a 20x10 grid, we just care about 10-20 end states, and the question: "which one has the highest value?" The model estimates the value of each state, picks the best one, and there is now a new state. Then a new piece is introduced, it is mapped to 10-20 new end states, the values are estimated, and the next state is chosen. Not only does it significantly simplify the game computationally, but it makes more sense fundamentally. Why is the machine taking an image of nothing but a block floating above a pile and trying to extract enough information itself to somehow understand what single next move makes sense in order to get a reward payoff twenty actions later in a state that could have been reached via any decision it could make? Nobody could ever succeed playing like that, and nobody does. Instead, the machine sees a state, and a set of ten or so next states to pick from, and is rewarded (as the game would reward us) based on its decision. Reaching that state is a trivial and uninteresting problem, and the wrong one to solve.

This experiment was so focused on how to pull of a policy-based method to see whether it could surpass value-based methods, that it never stopped to question what a policy-based method means in the context of Tetris. After hundreds of hours of attempting to make the wrong choice work, it wasn't until every way of trying to prove it right was exhausted, that I could finally accept that it simply was not.

References

[1] Gregory Cannon. Stackrabbit. GitHub, 2022.
[2] Christian Kauten. Tetris (NES) for OpenAI Gym. GitHub, 2019.
[3] Rex L. Reinforcement learning on tetris. 2021.
[4] Hanyuan Liu and Lixin Liu. Learn to play tetris with deep reinforcement learning, 2020.