A Reinforcement Learning Approach to Tetris

Andres Graterol
University of Central Florida
Orlando, USA
graterol.andres0@knights.ucf.edu

Carlos Herrera
University of Central Florida
Orlando, USA
leono@knights.ucf.edu

Stefan Werleman
University of Central Florida
Orlando, USA
stefanwerleman@knights.ucf.edu

CCS CONCEPTS

• **Theory of computation** → *Algorithmic game theory and mechanism design*; **Reinforcement learning**.

Abstract

The most popular method to test and experiment with reinforcement learning (RL) agents are through games. This method can yield many different results, but most importantly, can demonstrate the overall progression of an agent when playing the game. For this task, we conducted a series of experiments to test and evaluate multiple agents to play the game of Tetris. Tetris is one of most popular puzzle videos games and the main idea is to complete lines by moving and placing differently shaped pieces as it descends onto the bottom of the playing map. To our knowledge, this is a task that has so far not been achieved successfully by the use of RL agents. Through our investigation, all previous work in getting a bot to autonomously play Tetris consisted of constructing search heuristics for certain patterns that would allow for optimal block placements. In this work, we used the Deep Q Network, REINFORCE, and Proximal Policy Optimization reinforcement learning algorithms to play Tetris and then evaluated their performance. The algorithms were built using Tensorflow and were embedded in an existing python game called Petris. https://github.com/CAP6671-Multi-Agent-Systems/Petris

1 INTRODUCTION

When it comes to building agents to play games the first thing that comes to mind is reinforcement learning (RL). Reinforcement Learning is a machine learning method that is based on rewarding and/or penalizing behaviors of an agent in an environment. Tetris is a popular puzzle game where the player must complete as many lines as possible. Other researchers have tried to build agents to play the game of Tetris; however, the most common solution is an exhaustive search process which checks all different placements for the given shape instead of a machine learning learning approach using reinforcement learning. Our goal is train, experiment, and evaluate three different reinforcement learning algorithms that will play this game.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

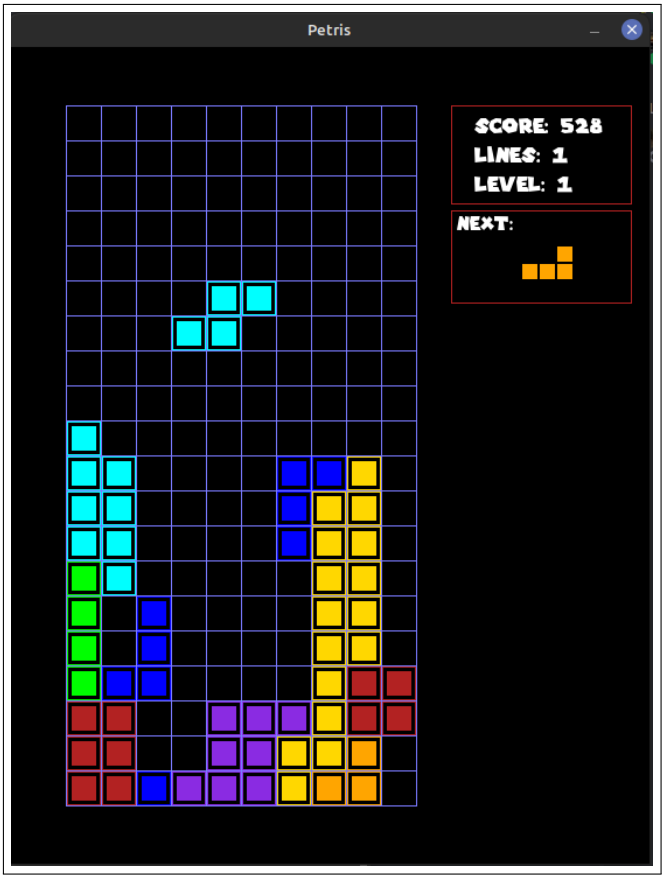

Figure 1: Petris game built using Python and PyGame

1.1 Petris

Petris is the pygame version of the popular puzzle game, Tetris [1]. The goal of Tetris is to achieve as many points as possible while strategically placing blocks of 7 different shapes onto lines (Figure 1). Lines are cleared as soon as every cell in a row has been populated by the unit of a block. The game ends if the user can not clear lines fast enough to prevent the blocks from reaching the vertical ceiling of the game space. In the game, 100 points are earned for every line cleared, and 2 points are earned whenever the player chooses to push the shape down (Figure 2). Levels are progressed whenever the player clears 10 lines. Every time a player advances to another level, their dropping speed increases.

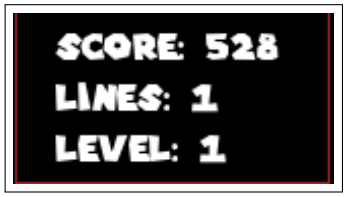

Figure 2: Scoring system of the game.

1.2 RL Agents with Tensorflow

There are many different reinforcement learning algorithms that could be used for this game. We forked this Petris game because it was built using Python with the PyGame module. Furthermore, we wanted to maximize Python's capability with machine learning by using Tensorflow to build our agents. Tensorflow is one of the most popular open-source library for machine learning that was developed by the Google Brain team. Tensorflow has a specific library called Tensorflow Agents which was used to build reinforcement learning agents. With the help of this library, we implemented the Deep Q Network, REINFORCE, and Proximal Policy Optimization algorithms to play Petris. Tensorflow Agents not only provides us with built-in agents, but the API also contains modules that can be used to easily create replay buffers, drivers, policies, and other features that are integral to reinforcement learning.

2 RELATED WORK

2.1 Deep Q Network

A Deep Q Network (DQN) is a reinforcement learning algorithm that utilizes a neural network and Q-learning all together. In Q-learning, the algorithm tries to learn a Q-value function which measures the reward of a particular action in a given state [2].

$$R = Q(s_t, a_t) \tag{1}$$

With DQN, it is the neural network's responsibility for learning an approximate value of the Q-value function. The Q-value function changes during training in which the DQN agent interacts or plays the game environment by receiving various rewards based on the actions the agent has made [6]. The environment in this context is the game itself where the agent can perform a set of actions. Based on the current state, the agent performs an action and will receive a reward for that action. The state, action, and reward are variables that represent an experience and experiences are stored in a replay buffer. The DQN agent uses this replay buffer to improve its overall performance. DQN is one of the most widely used algorithms for Atari games, board games, strategy games, and robotics.

2.2 REINFORCE

REINFORCE is a learning algorithm whose form is best described by its acronym: "*RE*ward *I*ncrement = *N*onnegative *F*actor times *O*ffset *R*einforcement times *C*haracteristic *E*ligibility" [7]. It is a policy gradient method, which is a learning method that relies on optimizing the policy with respect to the cumulative reward. This is achieved through stochastic gradient ascent. While stochastic gradient methods have good theoretical convergence properties, REINFORCE doubles as a Monte Carlo method which introduces high variance and thus slow learning [4]. If a baseline is introduced to REINFORCE, the policy gradient method will derive an update rule that takes this baseline into account - according to the Tensorflow documentation, a state-value baseline was used in their implementation. One thing to take special note of is that Petris is episodic in nature, and episodic REINFORCE has been proven to be especially slow due to its temporal credit-assignment spreading credit across all past times [7].

2.3 Proximal Policy Optimization

Proximal Policy Optimization (PPO) was developed by OpenAI in 2017 and builds upon the work done done for Trust Region Policy Optimization (TRPO). A policy can be defined as a mapping between the observations from the environment and a distribution of possible actions to be taken at that state. At its core, PPO still uses Trust Region Optimization to ensure that policy changes do not become too large and cause issues to convergence. PPO has two main variants - PPO Penalty and PPO Clip - which differ in their methods of capping off the change to the policy. For the sake of brevity, we will only go into detail on PPO Penalty, as that is the implementation that is contained in Tensorflow. PPO Penalty uses an adaptive KL penalty coefficient that is used to achieve some target value of the KL divergence during each policy update [3]. In layman's terms, this penalty term is added to the objective function to encourage the policy to change slowly over time. To conclude, PPO is a policy gradient algorithm, but its added features theoretically make it a more powerful and flexible method than alternatives such as REINFORCE.

3 METHOD

3.1 Command Line Arguments

3.1.1 Speed. This sets the speed of how fast the blocks drop. It works by counting the amount of frames per second, and once it passes the threshold set by us, it lowers the moving block by one.

3.1.2 Debug. This shows extra logs about the execution, rewards given, and more. It is used in development and for debugging errors.

3.1.3 Parameters. This command line argument provides the file input of the parameters needed to run the agents. In detail, it specifies:

- **Agent**: The agent to be ran (DQN, PPO, or REINFORCE).
- **Params**: All the parameters needed to test and evaluate the agent. It contains two objects. The first one, agent, contains hyperparameters related to the agent, such as the layers, epochs to train for, learning rate, etc. The next one, environment, holds all parameters needed for the environment, such as the penalty values for undesirable actions and reward per lines cleared. It was organized in this format to make it easy to test multiple different values for the hyperparameters without needing to manually change them in the codebase.
- **Bounds**: This object holds all hyperparameters bounds. It contains all the same variables as Params, but with arrays defining the upper and lower bounds for each hyperparameter, as needed for Bayesian optimization. It is important not to have extremely large bounds, as it will take longer for the

optimization to search through the hyperparameter bound space to find the best set of parameters.
- **To_maximize**: This string specifies which parameters we would like attempt to maximize for, either the agent, environment, or both. This prevents having to optimize too many parameters, and allows us to test how the optimization changes if other parameters are changed.

3.2 Environment

3.2.1 Observation. The observation space for our environment is a 20x10 2D array, modified to be a 200x1 array to work with our agents. It is taken directly from the game scene, which contains the state and logic for the Tetris game.

3.2.2 Action. The action space is a range between 0 and 3, indicating a valid action in Tetris (push down, rotate, move left, and move right). One of these actions will be picked by our agent, which will then be emulated as a key press and change the state in the game.

3.2.3 Step Function. The step function is responsible for assigning the rewards and penalties. Here is the list of rewards and penalties:

- **Rewards:**
  - *Blocked placed*: When a block is placed down, a smaller reward is given to encourage the agent to continue to place down blocks.
  - *Pressed down*: When the agent presses down and the block has not been placed, a very small reward is given. Since it can be moved down up to 20 times, this plus the block placed reward can compound to give a decent reward for one block.
  - *Line cleared*: When a horizontal line is full of blocks, it is cleared and the agent is given a large reward to encourage it to continue doing so. Additionally, for more lines cleared, it is given a larger reward than the previous one to encourage it to clear multiple lines at a time.
- **Penalties:**
  - *Height difference*: To discourage stacking blocks on top of each other to exploit the reward, we add a penalty for the difference in height between each column. This encourages the agent to keep the height of all columns the same, which should lead to more lines cleared.
  - *Holes*: Holes are defined as an empty space that has a block on top of it. This is undesired, as it prevents the line under from being cleared and increasing the overall height, which can lead to a game over. To discourage this, we check each column and penalize for each hole found.
  - *Game over penalty*: This penalty is applied when the game is over by stacking too high. This penalty serves to discourage it from ending the game and attempt to get more points.
  - *Too little blocks placed*: As we tested, we noticed that the agent would be punished less for stacking as thin as possible, to minimize both the height penalty and hole penalty. To prevent this exploitation, we added an additional penalty where the game over penalty would be scaled depending on how many blocks were placed down. The equation follows:

$$f(n) = \begin{cases} p \times (e - i) & \text{if } e > i \\ 0 & \text{if } e \leq i \end{cases}$$

Where p is the game over penalty, e is the amount of blocks expected to be placed, and i is the amount of blocks placed. e is close to the max amount of blocks that can fit inside the 20x10 Tetris map, meaning to prevent a game over penalty, the agent must efficiently place the blocks to fill the available space, which should lead to more exploration.

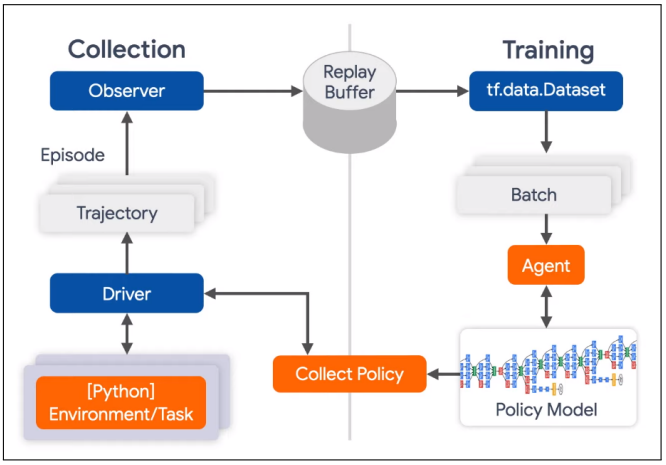

Figure 3: Workflow for using TF-Agents with our Python Environment [5]

3.3 Custom Features

While the default agent provided by Tensorflow were used, some other features needed to be custom made to provide full functionality for our implementation.

3.3.1 Driver. To visualize what the agent is doing in each step, a tensorflow driver was modified to render the screen after every step, letting us see what behaviour the agent picks up while training and respond to it accordingly.

3.3.2 Observer. To be able to collect all of the information needed to create our visualizations, a custom observer was made to collect the information needed. This includes a matrix of where the blocks have been placed, the amount of blocks placed, and the amount of lines cleared.

3.4 Replay Buffer

The replay buffer stores the experiences from the agent. This experience contains the state, action, and reward of the agent when it interacts with its environment. During training, the agent will continuously read a batch from the replay buffer to form an optimal policy (Figure 3).

3.5 Agents

Once the custom environment, driver, and replay buffer are ready, we can now implement an agent in the workflow. The agent receives

the batch from the replay buffer and updates its policy model. The policy model would be used to select an action to perform in the environment. We have to first train the agent in order for it learn an optimal policy. The best policy the agent can learn would be the one that yields the most reward with the help of the training algorithm (Figure 3).

4 EVALUATION

4.1 Initial testing

Once the agents were fully developed and integrated with the environment, we initially started testing by creating a default set of hyperparameters, and then manually changing them to see the results improve. However, there were a few issues with this method. As there are 25 hyperparameters related to the agent and environment that can be changed, it was extremely time consuming to run the agents, analyze the results, and change the hyperparameters.

4.2 Sequential Hyperparameter Optimization

Seeing how manually testing hyperparameters would take an extensive amount of time, we further developed the code to be able to automatically test parameters sequentially. We could define which variables we wanted to test, how much it should change per iteration, and for how many iterations, generating graphs and an output log in the end. This greatly sped up our evaluations, as we could test for a wide range of hyperparameters automatically. However, this method of sequentially finding hyperparameters led to a concern. Since we are using a greedy optimization of testing one hyperparameter at a time, we may miss a better result that may involve two different hyperparameters.

4.3 Bayesian Optimization

To solve the greedy optimization issue, Bayesian Optimization was used. Bayesian Optimization works by having a surrogate function that estimates the agent's result. This method allows the algorithm to quickly figure out what hyperparameters lead to better results and optimize towards them. We divided the hyperparameters into those affecting the environment and those affecting the agent, so we can test the best hyperparameters for each.

4.4 Graphs

To help us understand the results in an easily digestable manner, we generate a number of graphs to show what the agent was doing per epoch and per iteration. The graphs we generated follow below:

- **Average Return**: This visualization shows the average return, or reward, of each evaluation. This makes it easy to see how it trends as it proceeds to learn more with each epoch, and can be compared with others to see how different hyperparameters affect the return.
- **Loss**: Since the loss is the value that the agents use to train (by attempting to minimize it), it's crucial to see how the loss changes over each epoch and iteration.
- **Tetris Heatmap**: This graph is a heatmap of where blocks have been placed the most while training. This is relevant as this gives us a very clear indication of any habits the agent picks up while training. For example, Figure 11 shows how one of our iterations stacks vertically, meaning it is either being punished too heavily and it is attempting to minimize the penalty, or it is being rewarded too greatly and exploiting the reward without exploring the state space enough.

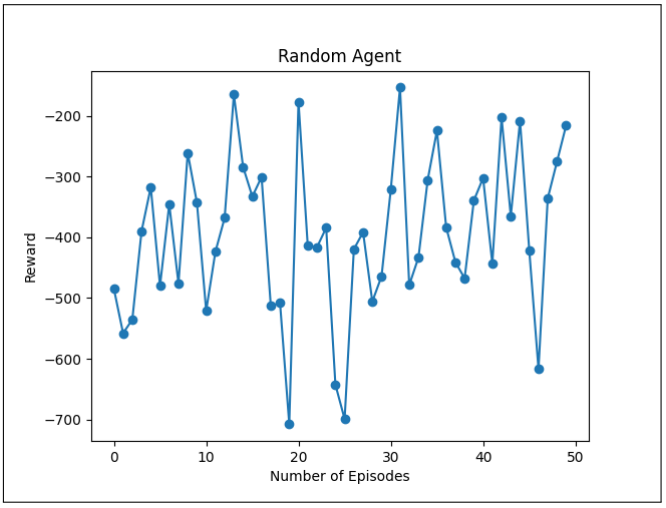

Figure 4: Rewards over each episode for the random agent

5 RESULTS

To generate results, we used the Bayesian Optimization algorithm to find the best results. We ran 25 random exploration iterations for the algorithm to diversify the exploration space, and then 75 optimization iterations for the algorithm to have ample iterations to explore and discover good hyperparameters. We also ran 100 epochs per iteration, as some of the algorithms, such as Reinforce, take longer to converge to an optimal policy. All hyperparameters related to the agent were optimized by the Bayesian Optimization. The results for the three reinforcement learnign agents are all in the Appendix.

5.1 Random Agent

Before moving on to training the three main agents, we wanted to do a preliminary test on the custom Petris environment. We created a random agent that uniformly performs one of the four actions at random. We wanted to ensure the step function described in section 3.2.3 works as properly and handles all performed actions.

After running 50 episodes with the random agent, the custom environment was ready for the main reinforcement learning agents. The random agent did not show a steady growth in terms of reward (Figure 4).

Random Agent	
Cumulative Reward	-49763.80
Mean Reward	-995.48
Std. Dev	129.69

Table 1: Results for the random agent over 50 episodes

Based on Table 1, you can see the random agent performed poorly in terms of reward.

5.2 DQN Agent

The Deep Q Network was the first agent we implemented and evaluated. First step was to define the neural network for the DQN architecture. You can view the layer representation of the sequential neural network in Table 2.

DQN		
Layer	Output Shape	Parameters
Dense 0	Multiple	31758
Dense 1	Multiple	14628
Flatten	Multiple	0
Dense 2	Multiple	372

Table 2: Layer definitions of the sequential neural network of the DQN. Total of 46,758 trainable parameters.

The figures 5-7 show the results from running Bayesian Optimization for 100 iterations on DQN. From these iterations, the iteration that produced the best agent hyperparameters proved to be the last one, iteration 100. As can be seen in Figure 6, the loss is represented by positive numbers, while the loss for the other two agents (Figures 9 and 13) are represented by negative numbers. This is due to the way that the loss functions are calculated. For DQN, the loss function is calculated as the mean-squared error between the predicted and target Q-values, and due to the nature of our reward values, in this case the outcome is positive.

5.3 REINFORCE Agent

Similar to how we calculate the results for the random agent (Section 5.1), we evaluate the performance based on reward over 100 epochs on the 36th iteration. In addition, we also calculated the loss and a heatmap for the agent. We ran a total of 100 iterations to give the agent time to learn from its replay buffer as it builds up and at iteration 36 the reinforce agent yielded the best results compared to other iterations.

Although the reinforce agent was significantly penalized, it managed to maximize its reward by roughly 100 units (Figure 8). After maintaining the loss in the first 30 epochs, it starts to slightly decrease (Figure 9). Reinforce was able to place the shapes in a way that the heatmap looks like a normal distribution (Figure 10). However, for Tetris it is best to place the shapes in a way so that the heatmap looks more like a uniform distribution where there is an equal spread of shapes on the tetris map. Fortunately, the reinforce agent showed promising results, especially in the first 30 epochs of this iteration in terms of reward and loss.

5.4 PPO Agent

The PPO agent's best run had similar results to the previous two agents (Figures 11-13). It had slightly more negative reward than the other agents, and the heatmap can confirm this by showing a narrower column than the other agents (Figure 11). This means it was simply stacking the blocks instead of attempting to clear a line, which indicates that the environment needs to be changed to encourage the agent to explore more of the state space and attempt to clear lines.

It should also be mentioned that the loss only has one initial data point. When analyzing the data, it was noticed that all loss values were NaN (Not a Number). After some investigation on what the issue could be, the likely cause of the issue has to do with exploding gradients. Exploding gradients occur when the error gradients used to train the network become too large, causing the loss to overflow and cause NaN errors. The likely cause of this error is due to the bounds set for optimization, however, more testing is needed to verify this.

6 FUTURE WORK

6.1 Agent Critiques

We may have underestimated the time it takes our chosen agents to learn an optimal policy, so in the future, we will be running for more epochs, to see if our agents can break out of non-optimal policies that they have learned. Alternatively, we could force the agent to explore the state-space more through the use of bandits. Additionally, to combat the high reward variance associated with policy gradient methods, a critic can be added to the system. When it comes to PPO, the version from Tensorflow we used, PPO Penalty, typically tends to perform computationally worse than its clipped surrogate objective counterpart [3]. In the future, we hope to try this experiment again with the PPO Clip agent found on Tensorflow.

6.2 Bandits

As we continue our work, we hope to incorporate a multi-armed bandit hybrid model with our existing agents. Multi-armed bandits are very useful in solving the "exploration-exploitation dilemma", in which agents tend to end up exploiting the action that gets the highest short-term reward, while potentially missing out on exploring the state space to find other actions that may offer even greater reward. This was an especially troublesome problem for our agents, as receiving a disproportionately large reward would cause the agents to over-correct and act on a sub-optimal policy, such as stacking to minimize the number of holes penalty. A hybrid model would have most of the characteristics of our main agent, while utilizing the exploration qualities of bandits to explore and find better policies. The main challenge of creating such a model, would be coming up with an implementation from-scratch, as Tensorflow currently does not support this functionality. Additionally, if we were to evaluate our Petris environment with a bandit, refactoring would have to be done on the custom environment, as bandits need additional exposed methods in order to work with Tensorflow. On top of this, we would need to restructure the way our game runs as bandits do not work episodically, unlike our game, which uses creates a new episode every time the game over screen is reached.

6.3 Agent and Environment Optimization

In our testing, we only used Bayesian optimization for all agent hyperparameters, as to avoid having a large exploration space, which may take a large time to explore. However, after seeing the results of our agents, attempting to optimize all hyperparameters will likely lead to much better results as it can change the reward and penalties the agent receives to encourage it to exploit clearing lines

and surviving for a longer time, instead of stacking and attempting to minimize the penalty given.

7 CONCLUSION

Instead of creating an agent that utilizes the common brute force searching solution to play Tetris (Section 1), we wanted to explore the reinforcement learning by implementing and utilizing multiple algorithms such as Deep Q Learning, REINFORCE, and Proximal Policy Optimization. We created a custom environment based off of an existing implementation of Tetris and implemented agents REINFORCE, PPO, and DQN. We then attempted to find the best set of hyperparameters through several methods, ultimately using Bayesian optimization to quickly and efficiently find the best values. After 100 iterations, all results under-performed, indicating that the environment has a large impact on the average return, or the bounds provided for the optimization were not large enough. Out of all three, REINFORCE performed the best on average, having the most variance in the heatmap and better performance by 200-300 reward points. Future work for this paper involves refactoring the agents to include critics or agents, tuning the environment reward hyperparameters, and using alternative agents, such as PPO Clip.

REFERENCES

[1] AlexArtuga lothar986. 2020. Petris. https://github.com/lothar986/Petris. (2020).

[2] Tambet Matiisen. 2015. Demystifying deep reinforcement learning. (2015). https://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/.

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

[4] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. (2nd. ed.). *Adaptive computation and machine learning series*. The MIT Press, Cambridge, MA.

[5] [n. d.] Tensorflow documentation. https://www.tensorflow.org/agents. ().

[6] Koray Kavukcuoglu Volodymer Minh and David Silver et al. 2015. Human-level control through deep reinforcement learning, 529–533. DOI: https://doi.org/10.1038/nature14236.

[7] Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 8, (May 1992), 229–256. DOI: https://doi.org/10.1007/BF00992696.

A APPENDIX

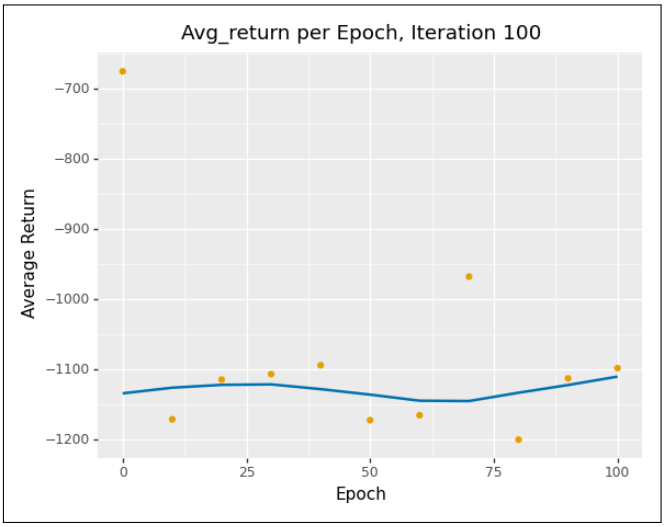

Figure 5: DQN - Average Return. Here we can see the best performance of greater than -700 occurred in the first Epoch of the iteration. This occurred as an outlier, and going forward, DQN remained consistently in the -1200 to -1100 range.

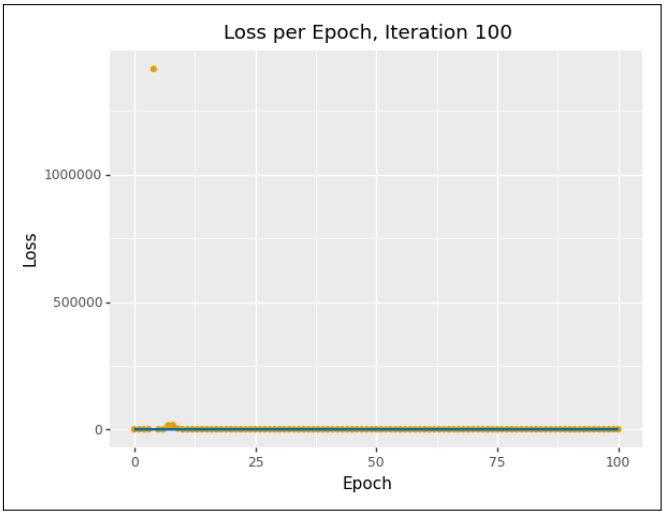

Figure 6: DQN - Loss. The greatest loss occurred near the beginning of the epochs for the iteration. Afterwards, the loss was stable for the rest of the epochs.

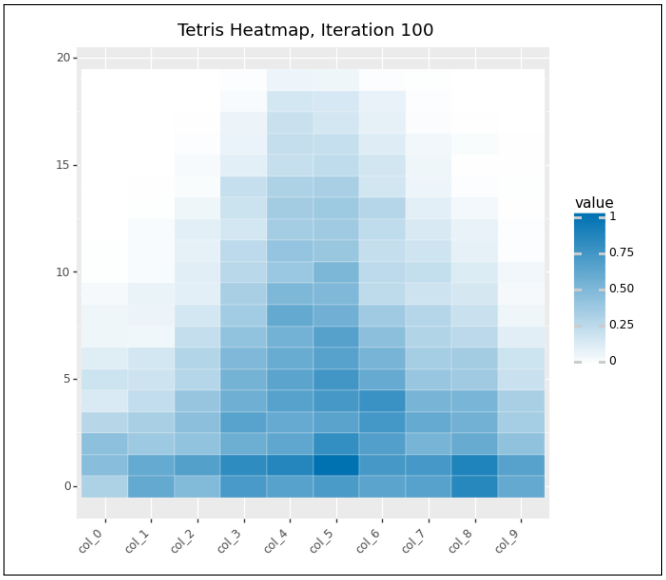

Figure 7: DQN - Heatmap. This shows an ideal performance where the blocks do not stack and instead are disbused along the width of the game space.

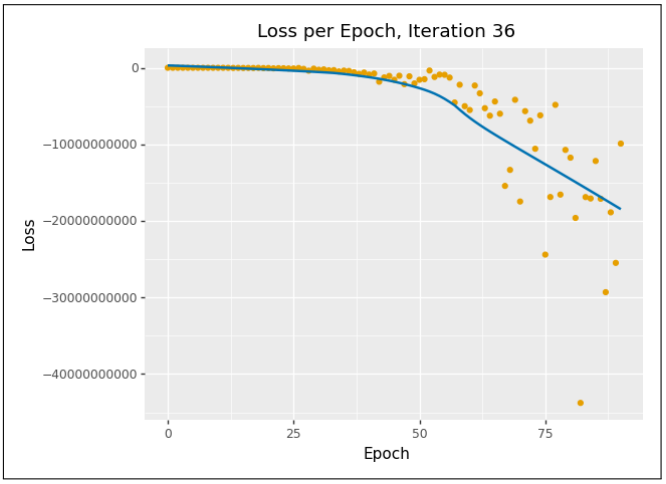

Figure 9: REINFORCE - Although REINFORCE was penalized heavily, it was still able to minimize the change in loss; hence, why the curve does not have a steep slope.

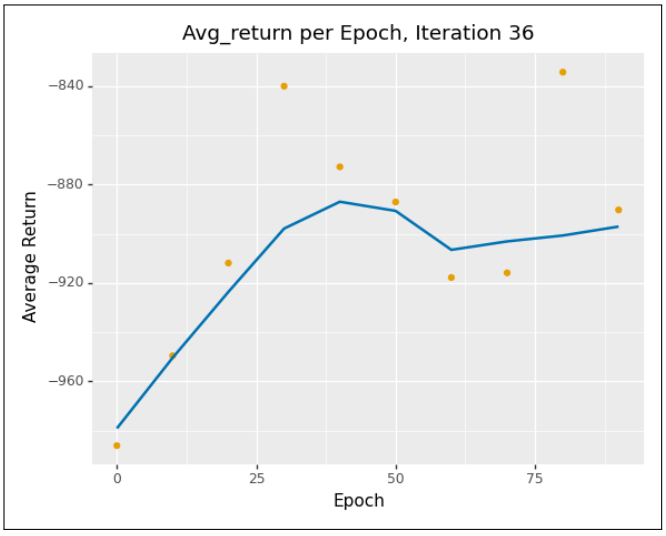

Figure 8: REINFORCE - You can see a steady increase in the first 30 epochs before plateauing.

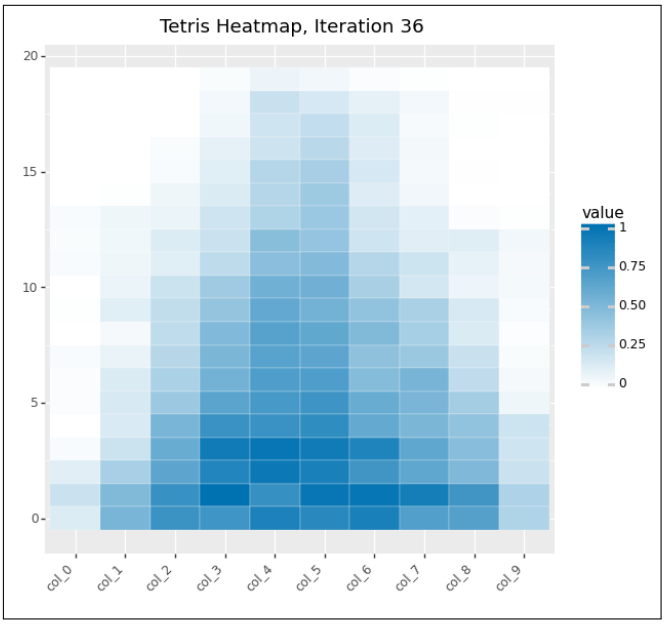

Figure 10: REINFORCE - Here the REINFORCE agent seem to stack the shapes at the center, but it still tried to place the shapes in the open regions on the sides which helped it complete more lines.

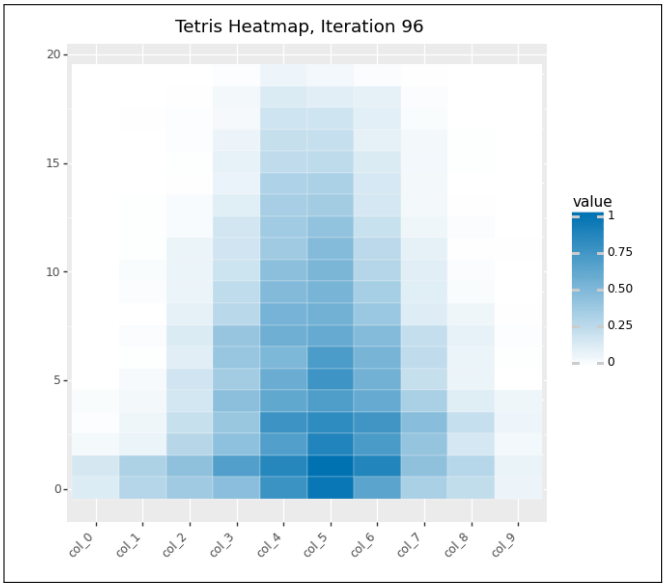

Figure 11: PPO - Heatmap demonstrates that the heatmap seems to value stack shapes more in the center compared to the other results.

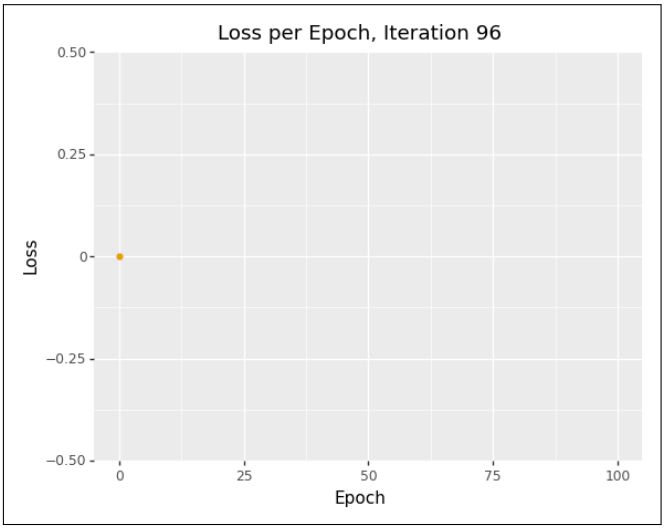

Figure 13: PPO - There is only one data point, as the rest of epochs return NaN. This indicates there is an issue with exploding gradients.

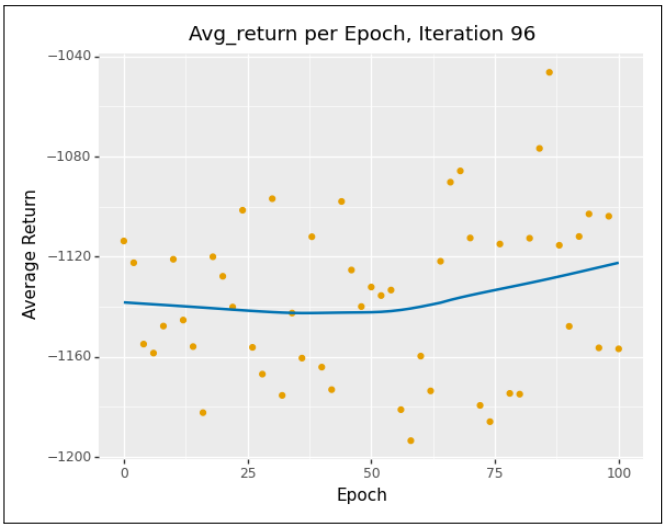

Figure 12: PPO - Over 100 epochs, PPO is kept at around reward -1140 to -1120. Therefore, it does not seem to learn during the training process.