Implementation of Reinforcement Learning Algorithms for Connect 4

Jonas Buro
Nils Dosaj Mikkelsen
Andrea Nguyen
Jacob Thom

Department of Computer Science
University of Victoria
Victoria, BC

July 20, 2021

Abstract

The goal of our project is to create a reinforcement learning agent which is able to beat a randomly playing opponent at the game of Connect 4. Reinforcement learning is perhaps the most promising sub-field of machine learning, thus the inclination to explore the process of devising and implementing a solution to a problem using research from the pioneers in the field. Two player perfect information games such as Connect 4 are perfect training grounds for an agent to be unleashed in, as the environment is able to be explicitly defined. The agent opponent and environmental impact on an agent was investigated at a high level to keep within the scope of this course.

Contents

1 Introduction

1.1 Introduction to Reinforcement Learning

Reinforcement learning is the branch of machine learning that focuses on training agents on specific tasks using a system of rewards and punishments [21]. Like supervised learning agents, reinforcement learning agents have training stages. However, where supervised agents are trained by labeled data sets, reinforcement learning agents optimize through a process of receiving positive and negative reward based on their actions. In this way, they are conditioned towards actions that result in the greatest accumulation of positive reward. Hence the name: reinforcement

learning. Reinforcement learning has a broad range of applications including robotics, chess and self-driving cars. The types of problems that are suitable for reinforcement learning applications is an ever expanding field [3].

1.2 Aims and Objectives

Our goal for this project is to to apply reinforcement learning to the classic game Connect 4. Since Connect 4 is considered a solved game [6], a game whose outcome (win, lose, draw), can be correctly predicted from any position, assuming that both players play perfectly [22], we felt that this game in particular offered a clear optimal benchmark for our agent to potentially achieve. While the idea of training an agent that could ultimately converge to a solved Connect 4 policy was an appealing, albeit lofty goal, it was not our main objective. Due to our current collective experience level with reinforcement learning, our main focus for this project was to design a functioning agent that performed better than simply making random moves, and then to attempt to understand, improve and quantify what differentiates a successful agent from an unsuccessful one.

2 Background information

2.1 Connect 4

Connect 4 is a two player board game, in which the players choose a colour and then take turns dropping coloured discs, sometimes referred to as checkers, into a seven column, six row vertically suspended grid. The pieces fall straight down, occupying the lowest available space within the column. The objective of the game is to be the first to form a horizontal, vertical, or diagonal line of four of one's own discs [22].

2.2 Terminology

The following is a list of definitions for some of the terminology that will be used throughout this report:

- Policy: The strategy of the agent.
- Reward: A metric of positive reinforcement. Accumulating the max reward possible is the goal of the agent (for better or worse).
- Time Steps: A measure of time used to delineate when discrete actions occur in relation to one another. For our purposes, time steps affect the number of calculations that are made during training.
- Perfect Information: A game has perfect information if each player, when making any decision, is perfectly informed of all the events that have previously occurred, including the initialization event of the game [23].
- PPO vs PPO2: Stable Baselines recommends using their PPO2 algorithm instead of PPO. The only differences are:

1. PPO2 is optimized for GPU's.
2. PPO2 allows for vectorized environments to batch training steps across multiple instances of an environment [12].

*At any point in this paper when referring to PPO implementation, PPO2 is what was used.

3 Proximal Policy Optimization

3.1 Background and Why PPO?

During the planning stage, one of our considerations was which reinforcement algorithm would be best suited to our needs. The process of picking the most appropriate algorithm, along with a reliable implementation, was something that prompted much discussion. Ultimately PPO was chosen for its performance and reported ease of implementation [5].

3.2 What is PPO

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm that was designed to out perform other policy gradient methods while simultaneously being easier to implement. The heart of PPO is contained in two steps:

1. Sampling the environment to collect and assess data
2. Optimizing the objective function using stochastic gradient descent [7]

At each step of the game, PPO will run its current policy for T time steps and calculate T advantage estimates (the estimate of the relative value of the selected action [7]. To calculate the surrogate objective function, we take the expectation of the optimal advantage estimate multiplied by the probability ratio denoted as:

$$L^{CPI} = \mathbb{E}[r_t(\theta)\widehat{A}_t]$$

3.3 Clipping

CPI in the above equation stands for Conservative Policy Iteration and refers to the fact that PPO uses a "clipping operation" to help prevent the policy updates from ever taking steps that may be considered excessively large and therefore overstep the optimal policy altogether. This is highlighted in the main objective function for PPO, which is the heart of the algorithm. The main objective function is denoted:

$$L^{CLIP}(\theta) = \widehat{\mathbb{E}}[min(r_t(\theta)\widehat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\widehat{A}_t)]$$

What the above formula expresses is in order to calculate the main objective function we calculate the expected minimum of two terms:

1. $r_t(\theta)A_t$: The default objective for normal policy gradients that yield a high positive advantage over the baseline [5]

$clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\widehat{A}_t$

2. $1 \pm \epsilon^1$: Similar to the first term except we apply a clipping operation of

[1]A typical value forϵ is 0.2

The value of the clipping operation is that if the advantage estimate is positive, the policy will never increase greater than $1+\epsilon$in the event that the current action is a lot more likely under the new policy than the old [5]. Any additional increase will result in no change to the existing policy past this point. A similar stabilization happens when the advantage function is negative and the resulting action is far less likely under the new policy. The latter scenario is the only time where $L^{CLIP}(\theta)$ returns
$r_t(\theta)\widehat{A}_t$ [5].

3.4 Stable Baselines and Optuna

PPO was initially developed by the research company OpenAI [11]. However, OpenAI's implementation of PPO (OpenAI Baselines) is considered to be inflexible due to its inability to modify agents or add custom environments. A more flexible and better documented fork of the OpenAI Baselines repository called Stable Baselines was ultimately chosen [15].

Along with Stable Baselines, we decided to use Optuna [13], an open source hyperparameter tuning framework to automate the process of selecting hypterparameters.

3.5 PPO Hyperparameters

Using Optuna we determined these "optimal" parameters:

Parameters	Value
Gamma	0.99
Steps per update	128
Entropy coefficient	8e-3
Number of Epochs	8
Lambda	0.8365

Here's a brief description of what these parameters are doing:

- Gamma: A parameter which biases the policy towards choosing the advantage estimate that gives the highest most immediate reward. Simply put, a reward this move is more valuable in the same reward three moves from now.
- Steps per Update: The number of steps to run for each environment per update.
- Entropy Coefficient: The coefficient for the entropy loss calculation term.
- Number of Epochs: The number of Epochs when optimizing the surrogate objective function.
- Lambda: A second hyper parameter for the advantage estimate [2].

3.6 PPO Hyperparameter Analysis

The following analysis was done using Kaggle's Connectx Environment. It has the following reward function:

Action	Value
Game Win	+1
Game Loss	-1
Game Tie	1/42
Valid Move	1/42
Invalid Move	-10

3.6.1 Optimized vs Unoptimized

Our expectation prior to running this experiment was that the hyperparameter tuning would have a potentially significant, and at the very least nontrivial, impact on episode reward (the cumulative amount of reward the agent sees). Although we do see an asymptotically faster rise in episode reward for the optimized agent, ultimately by the end, as shown in Figure 1, there appears to be no significant difference in either the total amount of episode reward gained or the asymptotic slope of the entropy loss (which affects the amount of exploration the agent undergoes while searching for the optimal policy).

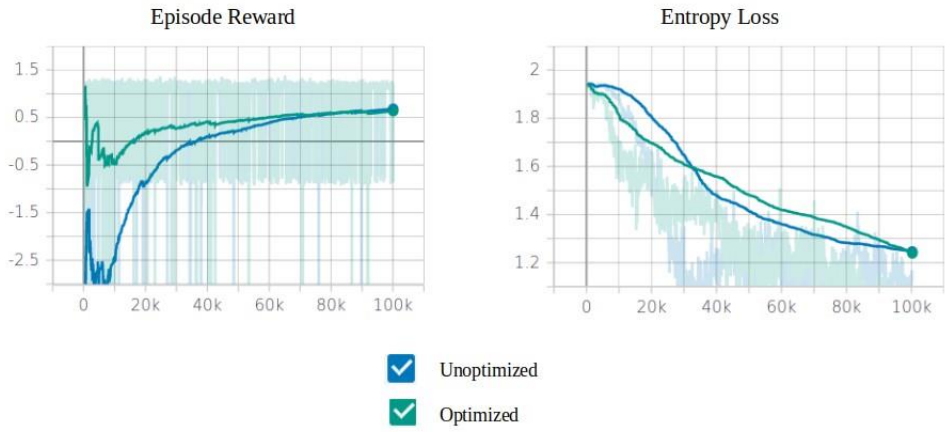

Figure 1: Optimized vs Un-optimized PPO Agent in the Kaggle Connectx Environment

3.6.2 Static vs Non-Static Learning Rates

One of our theories as to why the optimized agent was not reaching a higher episode reward than the un-optimized agent was that Optuna only returns static values for hyperparameters. Learning rates typically decrease over time [4] so a callable function was substituted for the initial floating point value obtained from Optuna. Interestingly enough, this resulted in a far worse performance

by the optimized agent. As shown in Figure 2, the optimized agent's max episode reward never rises above even -1 after some initial noise early on.

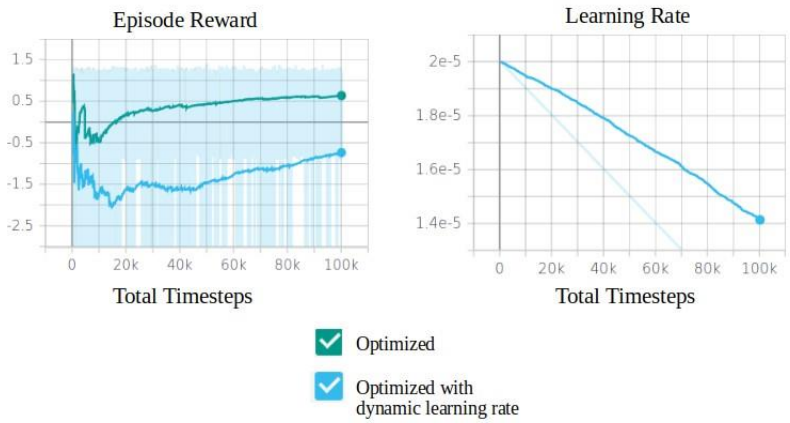

Figure 2: Optimized vs Un-optimized PPO Agent with Static vs Non-Static Learning Rates

4 Environment

In order to train a reinforcement learning agent, it is necessary to precisely define the environment in which it operates. To get things started, we used a pre-built Connect 4 Kaggle environment, and later shifted to developing our own. The Stable Baselines implementation of the PPO2 algorithm requires an environment of a particular type, namely one which is derived from OpenAI Gym's environment class. The derivation involves defining the following:

1. Action space: The set of all possible moves the agent can take.
2. Observation space: The set of all possible scenarios the agent can encounter.
3. Environment functions: Functions such as altering its state given some input.

We were interested in how agents which are trained on either environment differed. The following sections are an analysis of how they did.

4.1 Change in opponent behaviour

Initially the environment was developed to play versus a random opponent. Upon making a turn the environment would randomly select a legal column to place its move. This allowed the agent to develop its playing ability. Without an opponent to play against, it would not be able to learn as the environment would not be active. In an active environment, the agent's interaction must be met with a reaction in order to obtain a relative reward to the training goal. Upon observation of the agent's modelled on this environment, it was clear to the team that an alternative agent was required. A more intelligent environment agent theoretically conditions the learning agent to play better, as it must find ways to win in the environment in order to gain rewards. This led to the

development of a Minimax environment to mirror the implementation of Negamax found in the Kaggle environment.

4.2 Minimax Implementation

After training an agent primarily using a random move opponent, it became clear that its performance was sub-optimal when pitted against a human opponent. In order to increase the agent's capabilities, we decided to train it against a stronger opponent, namely a search based algorithm which we developed ourselves. The opponent which we created for the agent was a Minimax search algorithm with alpha-beta pruning. A Minimax search algorithm is a type of back-tracking algorithm which is used to find the optimal move for a player assuming that the opponent or opponents also play optimally [14]. It is typically applied to two player, perfect-information games like chess, so it was the perfect candidate for our Connect 4 opponent. The algorithm creates a tree like structure which represents the game starting from the current position p as the root of the tree. The tree then branches based on all the actions A which can be taken from p. Each of the children nodes of the root therefore represent the game state after each of the opponent's possible moves. Usually expanding the game tree entirely is unfeasible as there will be many states, which is clearly the case for the game of Connect 4, which has $(4.53199 \cdot 10^9)$ possible states. The solution is to set a depth parameter which builds the tree up to a certain amount of future moves. At each of the nodes, a static evaluation function assesses the position's value for the player we want to optimize. Our implementation relies on some basic heuristics to evaluate positions such as the number and length of streaks of markers the maximizing player has obtained in that position. The algorithm then recursively backtracks to find the optimal move for the maximizing player at position p. The implementation can be found in the *minimax* and *game* python files within our submission.

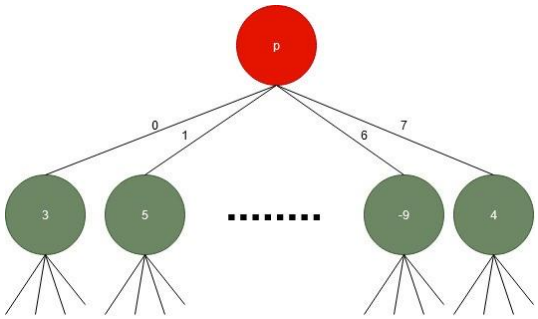

Figure 3: A minimax tree, with current gamestate p, decisions $[0,1,\ldots 7]$, and resultant gamestates with their static evaluations.

4.3 The Effect of Environmental Agents

Due to restrictions related to the custom environment's implementation of the minimax algorithm, data was not able to be gathered relating to it's specific effects. In lieu of minimax, the negamax opponent implementation in the pre-built Kaggle environment proved to be a successful adversary

to train the agent against, in the sense that it provided a challenge. The increase in difficulty, and subsequent decrease in mean reward, can be seen in Figure 4.

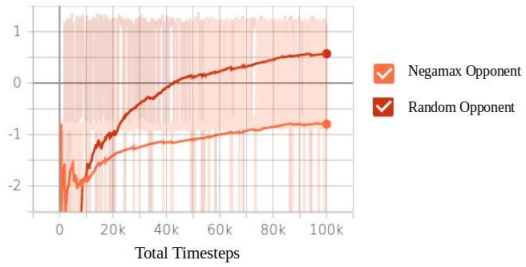

Figure 4: Episode reward for Kaggle agent trained against a random opponent and a negamax opponent.

In the figure, the model can be seen obtaining an average score that is analogous to losing a game when training on a negamax based agent. Notably, the figure displays a more shallow logarithmic curve than the curve of the agent trained on the random environment. This points to the fact that a stronger opponent is more challenging to compete against, as is expected. Finally, in learning against a difficult opponent it follows that the agent must adapt to a policy that more closely resembles the penultimate policy. This statement is corroborated by the table found below, which exemplifies the effect of the training an agent on different environmental agents. While the featured agents do not show notable performance in terms of winning Connect 4, the comparison between their mean reward is where conclusions may be drawn from. The specific conclusion to note is that the agent trained against a negamax opponent posses a greater mean reward than that of a random agent, suggesting that it is performing better in comparison.

Environmental Learning Agent	Mean reward:	Games Completed
Random	-1.2	123
Negamax	-0.7	111

4.4 The Effect of Training Length

As the number of episodes an agent undergoes increase, it follows that the agent is able to interact with, and therefore generate, increased data with the environment. In theory, with more information to learn from, the agent is able to generate an increasingly more sophisticated policy network. Interestingly, our initial runs of 100,000 timesteps demonstrated an agent that performed significantly better in the custom environment with a reward convergence at 0.84 compared to 0.57 of the Kaggle agent (Fig 5a). After this threshold the reward of the agent within it's training environment appeared to converge on a final value representing its ability to play the game against its opponent agent. To further investigate the effects of training length, the agents were trained for one million timesteps.

The Kaggle agent displayed dramatic improvement up to 0.91 while our custom agent improved to 0.96 (Fig. 5b).

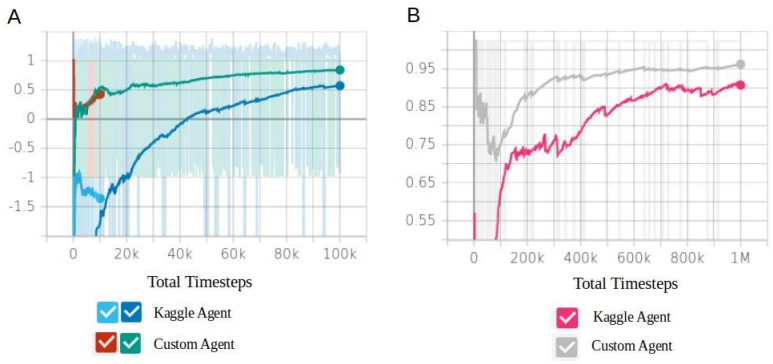

Figure 5: Effects of timestep on the agent's episode rewards in the Kaggle and custom environment.

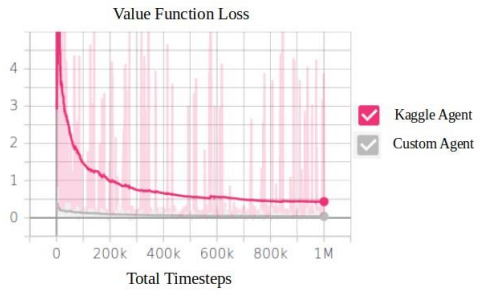

Figure 6: Stabilization of the value function loss for Kaggle agent and custom agent.

To confirm, we used Stable Baseline's policy evaluation helper function [2] on one million timestep trained agents and obtained similar mean reward values of 1.0 and 1.0 for the Kaggle and custom agents respectively. This indicates that the agent has settled on an optimal policy. When approaching the optimal policy the agent lowers the learning rate in order to maintain the chosen policy with minimal changes. This results in the episode reward graph taking a logarithmic shape as the length of training increases. The agent appears to continue converging towards a penultimate policy after the threshold, but at a significantly decrease rate. Similarly, the value function loss in Figure 6 shows a decrease as learning progresses, indicating that reward has become stable [1].

4.5 The Effect and Implementation of the Reward Function

The reward function for both environments followed similar scales. An agent trained on either environment obtained a reward of 1 for winning, and inversely, -1 for losing. Upon making a legal move, the agent was given a reward of $1/42$. Where the two environments differ is in their approach to illegal moves done by the agent during it's learning process. The agent in the Kaggle environment will receive reward of -10, leaving it up to the agent to discover and learn where it went wrong with its interaction with the environment. In the custom environment built by the

team, the agent is not given the ability to make an illegal interaction. The agent is instead allowed to continuously make moves while receiving no reward. Upon making a successful move, the agent is awarded the $1/42$ reward for making a legal move. This method for the reward was chosen to accelerate learning time, similar to coaching the agent on the legal options rather than punishing it without explanation on the illegal options.

Despite the learning and reward stabilization observed earlier, it would appear that the Kaggle agent was still making more random decisions than the custom agent as seen in the entropy loss graph Figure 7a. Although the difference in the policy gradient loss (Figure 7c) is small, the lack of oscillation in the Kaggle agent signifies that the agent believes it has found an optimal policy. Given these discrepancies in policy optimization we suspect that the reward implementation can influence the way an agent learns.

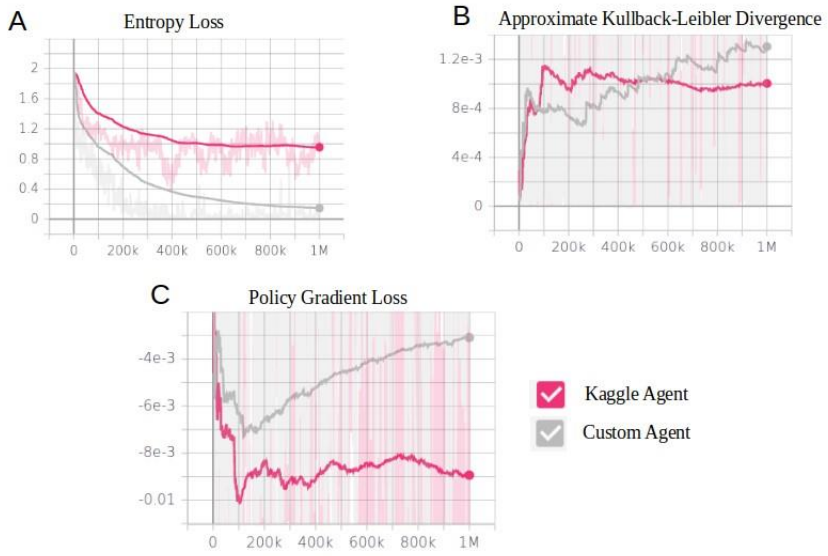

Figure 7: Policy convergence for Kaggle agent and custom agent.

5 Discussion

While hyperparameter tuning did result in some initial increase in episode reward (fig. 1a), ultimately the gains were insignificant. This is most likely due to the fact that PPO is considered far less sensitive to hyperparameter tuning in comparison to other policy gradient methods [5]. Typically early in training we can see how tuning increases the learning efficiency without having to increase the learning rate. However, final convergence seemed unaffected and the agents performance was very poor. An increase in timestep amount did appear to improve agent learning curve significantly but this did not translate into performance.

Our initial assumption pertained to a potential agent bias problem. In reinforcement learning it can be the case that an agent performs well in the training environment but poorly outside of that setting. Unfortunately, the agent's poor performance seen in Figure 4 was not indicative of an

opponent bias. Upon further inspection it was observed that the agent's reward does stabilize, Figure 6. It is possible then that the agent stable but making inaccurate value estimations, which is a characteristic of a high bias agent [9]. It is possible that our reward system is too naive and therefore hindering the learning rate. There has been evidence demonstrating enhanced learning using multiple goal states [10]. It is also the case that a poor reward function can result in a policy which demonstrates less optimal behaviour [7].

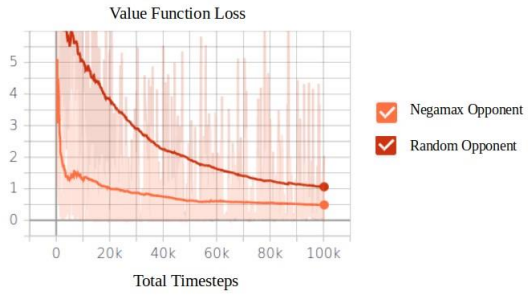

Figure 8: Stabilization of the value function loss for Kaggle agent trained against a random opponent and a negamax opponent

This bring us to our second assumption: The environment, more specifically the environment's reward system has a large influence on an agent's learning rate. Although both environments had a naive reward system, there was a noticeable difference in the policy changes seen in figure 7: The punishment system implemented by Kaggle resulted in a lower performance playing connect 4 than our custom agent which had no punishment system. This begs the question: Can B. F. Skinner's theory of operant conditioning [18], reward and punishment systems, be applied to produce intelligent machine learning agents?

5.1 Poor performance of Agent

The agent's poor results can be attributed to the lack of competition it faced during its training process. The custom minimax agent, which relied on exponential time complexity heuristics and state checking proved to be too cumbersome to train against for large numbers of episodes. In the future, we would look at reconstructing the heuristic evaluation function such that the time complexity of the algorithm was not as bad as it is in its current state. Further, the heuristic function itself can be optimized, which is an avenue which we did not pursue in the scope of this project.

In addition to creating a somewhat intelligent opponent for our agent, if we were to continue working on this project we would consider playing the agent against previous, less competent versions of itself. This idea of bouncing an agent off of its previous iterations in order to continually improve is a true and tested approach within the field of reinforcement learning.

5.2 UI visualization

In order to give our agent a stage upon which to showcase its abilities, we created a user interface using Python libraries Pygame and Pygame GUI. The program allows the user to play Connect Four

against a human opponent or a trained agent. It also allows the user to watch the agent play against itself.

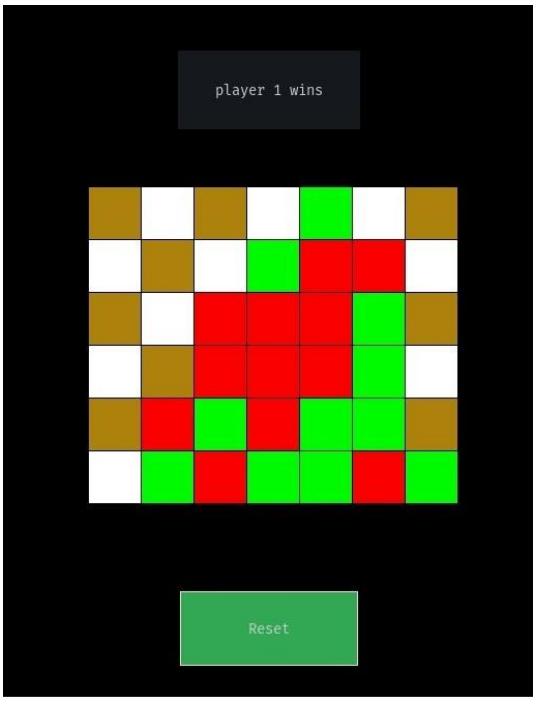

Figure 9: A screen capture of our UI after the completion of a game.

6 Conclusion & further research

In this paper, we have demonstrated how robust PPO is to hyperparameter initialization when it comes to reinforcement learning. However that is not to say that hyperparameter tuning cannot improve, or on occasion worsen, results. Further demonstration showed that a PPO based agent's learning rate can be influenced by the environment itself, such as the reward system and the difficulty of the environmental agent.

While analysis was done at great length upon the implemented methods and tools, there are notable improvements that can be done for future research. One improve could be made by implementing a broader range of policy networks and comparing their individual performance before attempting hyperparamter tuning, thus giving the best agent the best chance of developing a solved Connect 4 policy. In terms of developing a solved Connect 4 policy, a beneficial factor may be an increased set of rewards which coach the agent on a strategy that resembles the solved Connect 4 algorithm. It may be argued that the implementation of rewards related to the defined Connect 4 algorithm detract from the machine learning component of these experiments - this distinction is left to be made by future researchers.

References

[1] aureliantactics. Understanding ppo plots in tensorboard. https://medium.com/aureliantactics/ understanding-ppo-plots-in-tensorboard-cbc3199b9ba2, [Online] accessed on 2020-08-03.

[2] Stable Baselines. Ppo2. https://stable-baselines.readthedocs.io/en/ master/modules/ppo2.html, [Online] accessed on 2020-07-10.

[3] S. Bhatt. 5 things you need to know about reinforcement learning. https://www. kdnuggets.com/2018/03/5-things-reinforcement-learning.html, [Online] accessed on 2020-08-04.

[4] J. Brownlee. How to configure the learning rate when training deep learning neural networks. https://machinelearningmastery.com/ learning-rate-for-deep-learning-neural-networks/, [Online] accessed on 2020-08-01.

[5] Arxiv Insights, 2018. https://www.youtube.com/watch?v=JgvyzIkgxF0& feature=youtu.be, [Online] accessed on 2020-06-30.

[6] T. Jakobsson J. Persson. Self-learning game playerconnect-4 with q-learning. https://pdfs. semanticscholar.org/b504/0910e9ab62925a29a15a3aeb895967ea2cf0. pdf, [Online] accessed on 2020-06-16.

[7] P. Dhariwal A. Radford O. Klimov J. Schulman, F. Wolski. Proximal policy optimization algorithms. https://arxiv.org/pdf/1707.06347.pdf, [Online] accessed on 202006-25.

[8] Kaggle. Kaggle-environments. https://github.com/Kaggle/ kaggle-environments, [Online] accessed on 2020-07-30.

[9] R. S. Lee. Bias-variance tradeoff in reinforcement learning. https://www.endtoend.ai/ blog/bias-variance-tradeoff-in-reinforcement-learning/, [Online] accessed on 2020-08-04.

[10] M. J. Mataric. Reward functions for accelerated learning. http://www.sci.brooklyn. cuny.edu/~sklar/teaching/boston-college/s01/mc375/ml94.pdf, [Online] accessed on 2020-08-05.

[11] OpenAI. Proximal policy optimization, . https://openai.com/blog/ openai-baselines-ppo/, [Online] accessed on 2020-06-30.

[12] OpenAI. Openai baselines, . https://github.com/openai/baselines/issues/ 485, [Online] accessed on 2020-08-01.

[13] Optuna. Optimize your optimization. https://optuna.org/, [Online] accessed on 2020-07-23.

[14] X. Cui H. Dong F. Fang S. Russell S. Li, Y. Wu. Robust multi-agent reinforcement learningvia minimax deep deterministic policy gradient. https://people.eecs.berkeley.edu/ ~russell/papers/aaai19-marl.pdf, [Online] accessed on 2020-08-05.

[15] T. Simonini. On choosing a deep reinforcement learning library. https://medium.com/data-from-the-trenches/ choosing-a-deep-reinforcement-learning-library-890fb0307092, [Online] accessed on 2020-06-30.

[16] TensorFlow. Tensorboard: Tensorflow's visualization toolkit, . https://www. tensorflow.org/tensorboard, [Online] accessed on 2020-07-30.

[17] TensorFlow. An end-to-end open source machine learning platform, . https://www. tensorflow.org/, [Online] accessed on 2020-07-30.

[18] Wikipedia. Operant conditioning, . https://en.wikipedia.org/wiki/Operant_ conditioning, [Online] accessed on 2020-08-07.

[19] Wikipedia. Numpy, . https://en.wikipedia.org/wiki/NumPy, [Online] accessed on 2020-07-30.

[20] Wikipedia. Numpy, . https://en.wikipedia.org/wiki/NumPy, [Online] accessed on 2020-07-30.

[21] Wikipedia. Reinforcement learning, . https://en.wikipedia.org/wiki/ Reinforcement_learning#:~:text=Reinforcement%20learning% 20(RL)%20is%20an,supervised%20learning%20and%20unsupervised% 20learning., [Online] accessed on 2020-08-02.

[22] Wikipedia. Solved game, . https://en.wikipedia.org/wiki/Solved_game, [Online] accessed on 2020-08-04.

[23] Wikipedia. Perfect information, . https://en.wikipedia.org/wiki/Perfect_ information, [Online] accessed on 2020-07-20.

[24] Wikipedia. Pygame, . https://en.wikipedia.org/wiki/Pygame, [Online] accessed on 2020-07-30.

Appendices

A Tools Used

This project was programmed using Python 3, specifically Python 3.6 and 3.7. There were a number of additional third-party libraries, many of them new to us, that were used as well. The following is a list of libraries that were specifically and directly used by us, however, additional libraries are used implicitly within some of these as well:

1. Numpy: Adds support for large, multidimensional arrays and matrices [20].
2. Gym: A toolkit for developing and comparing reinforcement learning algorithms [19]. Gym also functioned as a wrapper class for our custom environment.
3. Pygame: A library of computer graphics designed to aid in writing python based video games [24].

4. Pygame Gui: An extension of Pygame that adds additional functionality for building graphical user interfaces.
5. TensorFlow: An open source machine learning platform consisting of extensive set of tools and libraries [17].
6. Tensorboard: Tensorflow's visualization toolkit used for tracking and visualizing metrics such as reward, loss and accuracy [16].
7. Stable Baselines: A fork of OpenAI's Baselines implementation of PPO.
8. Kaggle Environments: A library to aid in episode evaluation [8].
9. Optuna: An open source hyperparameter optimization framework [13].

Venv, a module that helps create virtual environments, was also used and is part of the Python Standard Library.

B Github Repository

https://github.com/selfsim/Connect4