Independent Study in Reinforcement Learning

Ethan Nhu Tran
North Carolina State University
Raleigh, NC
entran@ncsu.edu

***Abstract*—Reinforcement learning is a type of machine learning where an agent learns from its successes and failures. In my independent study, I learn about reinforcement learning and try out some basic algorithms. I attempt to apply what I learned about reinforcement learning to a platformer game but find that without further reward engineering, the agent achieves quite poor performance.**

I. Introduction

Reinforcement learning (RL) is a type of machine learning method gaining traction for a variety of applications. Most notably, it has been applied to train a chatbot known as ChatGPT [1]. For my independent study, I watched online lectures about reinforcement learning from David Silver [2], read the textbook by Sutton and Barto [3], and attempted to implement some basic reinforcement learning algorithms in Python. For my final project, I attempt to apply a reinforcement learning algorithm to have an agent to play a platformer video game called Celeste.

II. Background

A. Reinforcement Learning

The basic principle behind reinforcement learning is to give the computer rewards or punishments (negative rewards) as it tries to complete a task. As the agent, the software doing the decision-making, interacts with the environment, the environment sends back a reward signal the agent can utilize to learn from. The agent learns by improving a value or policy function from the received rewards as it transitions through different states. Examples of RL algorithms include state-action-reward-state-action (SARSA), Q-learning, deep Q-networks (DQN), and proximal policy optimization (PPO). RL algorithms have been applied to several video games already, a notable example being a 2013 paper in which agents were able to play Atari games with great success [4].

Reinforcement learning can be divided into tabular and approximate methods. Tabular methods can be used to find exact solutions to problems while approximate methods are efficient for large problems where the space of actions and observations are impractical or impossible to store in a table. For tabular methods, dynamic programming, Monte Carlo methods, and temporal-difference learning can all be used to find solutions to reinforcement learning problems. With approximate methods, algorithms such as SARSA, REINFORCE, actor-critic, and Q-learning can be used. These approximate methods can also be made even more effective by being extended to use deep neural networks, but the utilization of neural networks requires some extra components for the algorithms to work well. For my basic algorithm implementations, I mainly attempt tabular methods and some approximate methods.

At its core, reinforcement algorithms use some value function, policy function, or action-value function to estimate how good it is to be in some state or take some action. The value function takes in an observation from the environment and returns a number representing how good being in that state is. The policy function takes in an observation and returns the action it thinks should be taken next. The action-value function is similar to the value function, but it also takes in an action as a parameter for determining how good a state is.

B. Celeste

Celeste is a 2D platformer video game developed by indie studio Matt Makes Games. The gameplay of it is not unlike Mario. At its core, the player can control a character named Madeline by moving left or right, jumping, squatting, grabbing onto a wall, dashing for a speed boost, or some combination of these moves. The game has multiple levels that each consist of several rooms. These rooms are populated with spikes, pitfalls, and other obstacles that the player has to navigate around. The goal of the player is to navigate through the different rooms of a level to reach the top of the mountain. This paper focuses on the first level of Celeste as it has the simplest mechanics. More complex mechanics are added to the game during later levels, making it more difficult for a machine learning algorithm to solve.

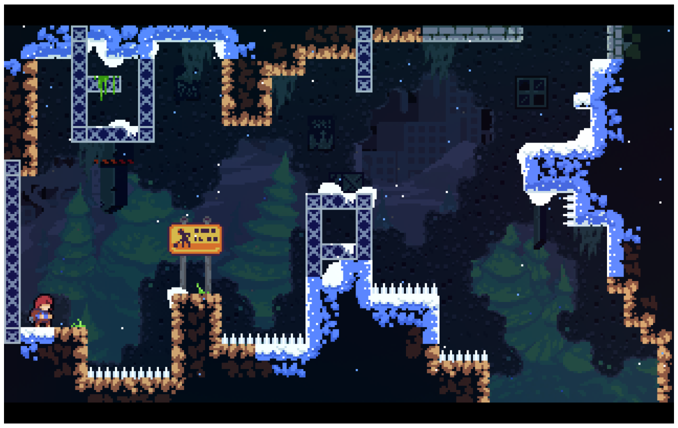

Fig. 1. Example Celeste Room

III. Algorithm Implementations

All the implementations done here can be found on GitHub at https://github.com/iSkytran/rlexercises/.

A. Dynamic Programming

For dynamic programming, I created a grid world problem environment. In this grid world, I have the agent attempting to reach the northwest and southeast corners. The agent gets a penalty of -1 for each movement to a non-terminal location and a reward of 0 for moving to a terminal location. I tried both policy iteration and value iteration to solve this problem. Policy iteration evaluates a policy and improves it over and over until the policy converges to some solution. Value iteration finds an optimal value function then extracts the optimal policy from the value function after the value function converges.

For a four by four grid, policy iteration took three policy improvement phases before converging. The first evaluation took 46 iterations, the second evaluation took 4 iterations, and the third evaluation took 1 iteration. As for value iteration, only three iterations were needed before the value function converged and a policy could be extracted.

B. Monte Carlo

For Monte Carlo, I created a racetrack environment where the agent could change its velocity. In this environment, there is a chance for a velocity change to not be applied to imitate stochasticity. The way that Monte Carlo works, full episodes are run and the actions, states, and rewards during each step of the episode are stored, and these values are "backed up" to update the action-value function. The rewards are discounted so actions taken at the beginning of the episode are weighted less than those at the end. Actions are also chosen with an epsilon-greedy policy, a policy that chooses the best action known with some chance to take a random action, to ensure that the full action space is explored. My implementation of this performed quite well, converging within under 100 episodes.

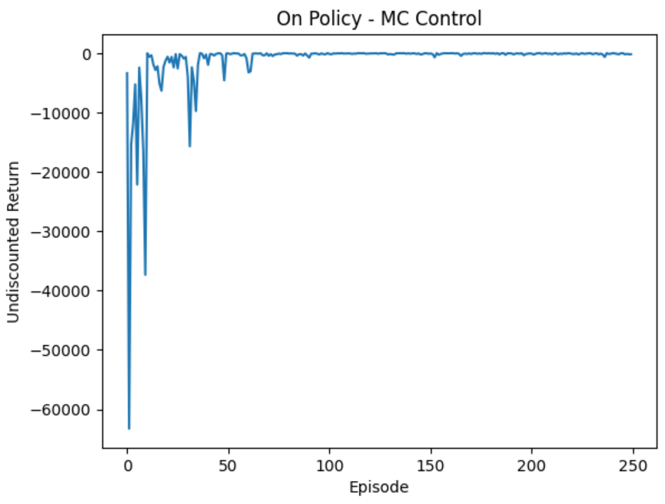

Fig. 2. Monte Carlo Performance

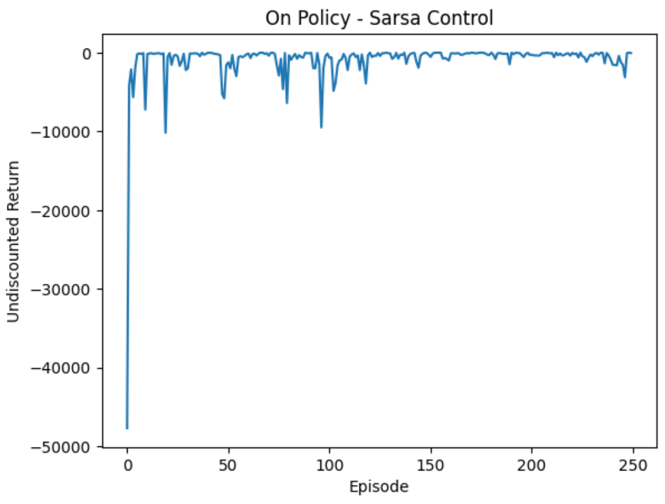

Fig. 3. SARSA Performance

C. SARSA

I used the same racetrack environment that I used for Monte Carlo for SARSA. The main difference between Monte Carlo and is that with SARSA, the agent does not wait for the whole episode to finish before updating its action-value function. It updates its action-value function at every time step, using the action-value error and reward received from taking some action. This action is chosen with an epsilon-greedy policy just like the Monte Carlo algorithm used.

D. Approximate Methods

I tried various algorithms and implementations of approximate methods to solve cart pole and mountain car from the Farama Gymnasium library. Overall, the performed of these implementations were less than desirable, not really solving the problems very well. The implementation that worked the best is my linear approximation SARSA implementation. This implementation uses the `RBFSampler` class from scikit-learn to represent the action-value function. The results of this implementation on cart pole are shown in Fig. 4.

However, when I applied this implementation to mountain car, the algorithm was unable to solve the problem. This is shown in Fig. 5 as the agent is unable to reach the top of the mountain before it hits the time step limit of 200 steps. This is likely due to a linear approximation method not being powerful enough to solve this problem.

I also tried implementing policy gradient methods, those that try to optimize a policy function rather than an action-value function, with both `RBFSampler` and using neural networks from PyTorch, but the agents ended up acting in a manner than is no better than taking random actions as seen in Fig. 6, Fig. 7, and Fig. 8.

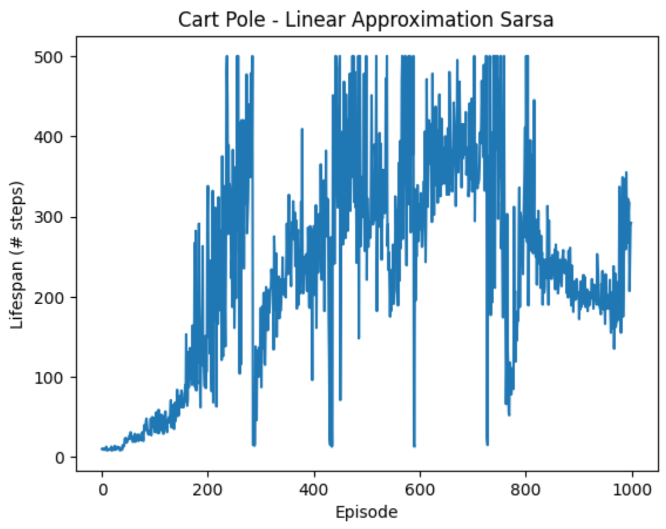

Fig. 4. Linear Approximation SARSA Cart Pole Performance

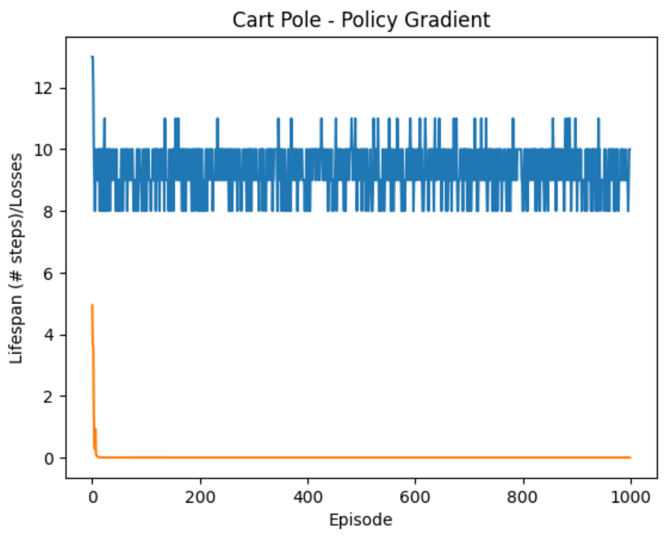

Fig. 6. Neural Network Policy Gradient Cart Pole Performance

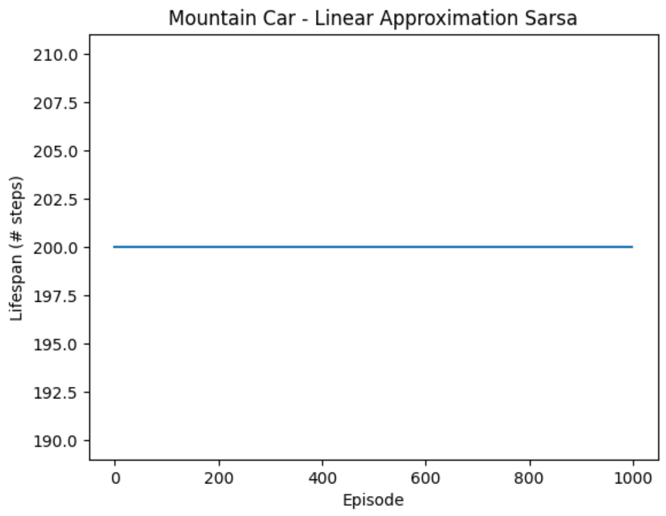

Fig. 5. Linear Approximation SARSA Mountain Car Performance

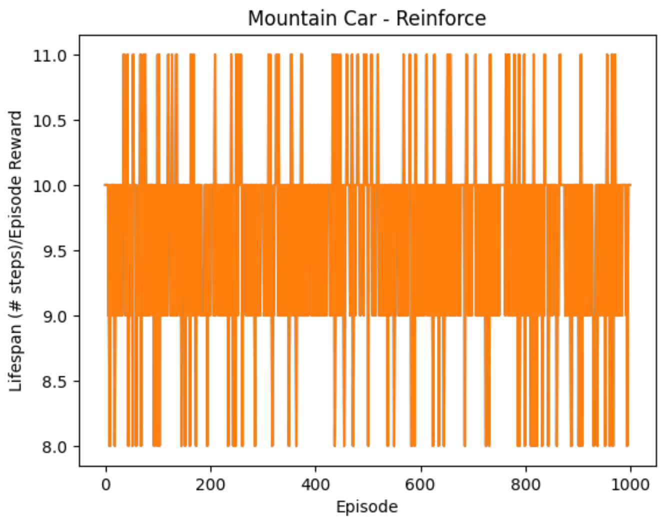

Fig. 7. Linear Policy Gradient Mountain Car Performance

IV. Project

All of the relevant code for the project can be found on GitHub at https://github.com/iskytran/ctrl.

A. Approach

Instead of creating a deep learning reinforcement learning implementation from scratch, I decided to make use of a well-tested library known as Stable-Baselines3. This library is written in Python and has support for a variety of algorithms. Most notably the library has an implementation of PPO that I use in this paper. PPO is a type of policy optimization algorithm that tries to refrain from making large policy updates. PPO was chosen for use because of its previous use on Atari games and has shown that it outperforms other policy gradient methods [5].

In order to control the character in Celeste, I decided to make use of the Celeste community's well-supported modding framework known as Everest. As Celeste is written in C# using the Microsoft XNA/FNA framework, mods are written in C# as well. I used this framework to allow the agent to send movement commands to the game, get rewards out of the game depending on if the character has moved or died, and determine when an episode has terminated or not.

Stable-Baselines3 expects a Farama Gymnasium environment, so custom environments that interact with Stable-Baselines3 must inherit from the `Gym` class and implement the `reset` and `step` method calls.

In order to connect the custom `Gym` environment with my C# mod, I needed to find a programming language agnostic interprocess communication (IPC) method. Socket programming was one option for accomplishing this but using sockets would

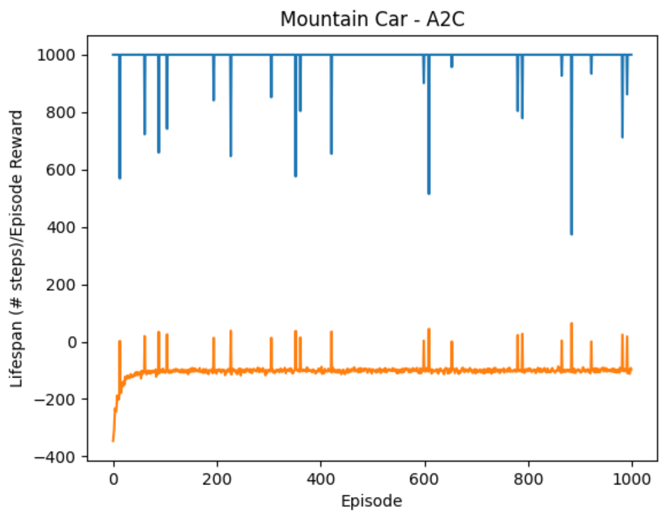

Fig. 8. Advantage Actor-Critic Mountain Car Performance

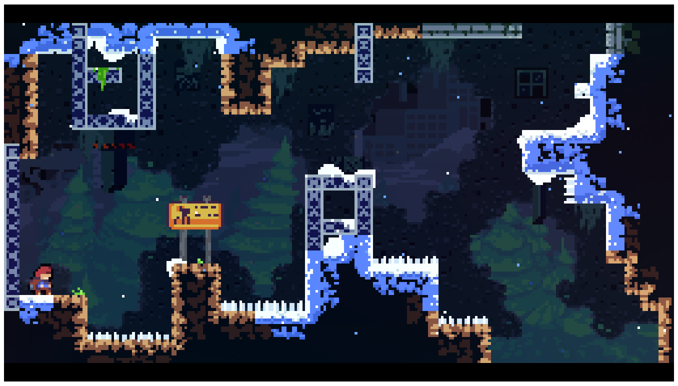

Fig. 9. Downsampled Observation of a Celeste Room

require a lot of extra networking implementation. The method I decided to go with was to use the ZeroMQ messaging library. ZeroMQ is a universal messaging library that abstracts away the details of bare socket programming. ZeroMQ allows the Stable-Baselines3 agent to make requests containing an action to take to the C# mod, and it allows the C# mod to reply to that request with the rewards and termination state.

In order for the reinforcement learning algorithm to make decisions and take an action, the agent needs to be fed some observation about the environment. In this case, the observation I feed to the agent is the visual rendering of the game. The first way I tried to accomplish this was to read the rendering buffer directly from the game using the C# mod and sending it over the ZeroMQ connection. However, this approach seemed to slow down the game a lot and induced a lot of overhead. The approach I went with instead was to have the Python side of the pipeline scrape visual data from the screen using the `mss` library. Using the `mss` library proved to be a much quicker solution that induced less overhead. This scraped image could then be directly fed into the Stable-Baselines3 agent which can be configured with a convolutional neural network for images. However, feeding the raw 1080 by 1920 pixel image the agent was infeasible as that amount of information would result in the agent crashing due to the sheer memory requirements. I instead had the image downsampled to a 144 by 256 pixel image. As Celeste is a game utilizing pixel art, this downsampling of the image does not drastically affect how the game looks.

The major issue with making sure the agent would work properly is to make sure that the agent is working in lockstep with the game. That is, every time the agent called `step`, the game should advance some fixed amount. In order to accomplish this, I injected code into the game's main loop that would block until a step message was received from the agent.

Additionally, to prevent the game from freezing up, the C# mod modifies the XNA/FNA game loop to use variable time steps rather than best effort fixed time steps. Using best effort fixed time steps would often lead to the rendering component of the game stalling forever as the game engine is designed to continue running updates but bypass rendering when the game is running slow. Attaching the Stable-Baselines3 agent with blocking logic in the game loop would cause the game engine to run slower than normal at some point, leading to the fixed time step game loop continually doing updates without rendering the changes to the screen. Using variable time stepping logic in the game loop prevents this issue from occurring because the variable time stepping version of the game loop always renders a frame every update.

Deciding what rewards to give the agent took a few iterations of tweaking. Initially, the rewards given to the agent were a large positive reward for moving to a new room and a large negative penalty for dying, and the only condition for termination for terminated was when the character died. I made modifications to the rewards as I tested the pipeline out before running it for the whole duration.

B. Results

Overall, the results of the PPO agent in Celeste were less than stellar. There were refinements I needed to make to the rewards, and even after tweaking the rewards, the agent did not learn very well. I ran the pipeline for ten million time steps, which took about a week to complete.

The first tweak I made was to make the episode terminate when the agent stood still for some specified time frame. Afterwards, I added code that rewarded the agent for moving upward and toward the right of the world (towards the approximate ending location of the level). Lastly, I reduced the values for moving to the next room and for dying, and I gave the agent a very large penalty for standing still. Without the large penalty and episode termination for standing still, the agent would find a locally optimal solution of not moving in order to escape the penalty of dying. This issue was also partly due to the difficulty and sparsity of obtaining the positive reward for finding a new room. Adding the reward for moving upward and to the right was done to mitigate this issue and to prevent the agent from finding another suboptimal solution of just moving back and forth on safe ground. Without this extra piece of reward engineering, sometime the agent would just spam the crouch action over and over.

There were, however, at least two instances where I visually noticed the agent discovering a new room. These instances happened early on in training, and there could have been more instances of room transitions, however, as I ran the agent unattended, I did not see any other occurrences. In hindsight, I should have added a logging mechanism for each time a room transition occurred.

Fig. 10, Fig. 11, and Fig. 12 show some metric value with respect to iterations. One iteration of the agent consists of data collection and a policy update. Fig. 10 shows that as the agent trains, there is no noticeable improvement. In fact, the more the agent trains, it seems that the performance got worse as the agent would get penalized more. This is mirrored by Fig. 11 with episodes taking longer to complete. Fig. 12 shows that there is some decrease in the loss of the agent's neural network, but not enough to signify that the agent was able to figure out the game properly. The two instances where I noticed a room transition likely correlate to just before iteration 1000, where the average reward and loss approach zero.

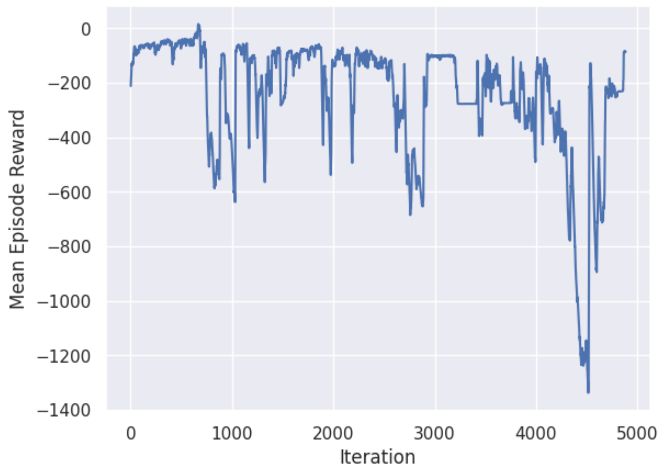

Fig. 10. Average Reward Per Iteration

C. Limitations

One problem with the current implementation is that it is not possible to vectorize the environment. In order for the pipeline to run, an instance of Celeste should be running. Celeste is run through Steam, a game distribution platform, but the Steam launcher limits how many instance of a game can be launched at a time. Due to this restriction, only one environment of Celeste can be run at a time, leading the agent take longer to converge to a solution.

Another major limitation of the current implementation is with the game loop. Even with variable time stepping, if the game engine is running faster than the desired number of frame-per-second (FPS), it will sleep the game in an attempt to

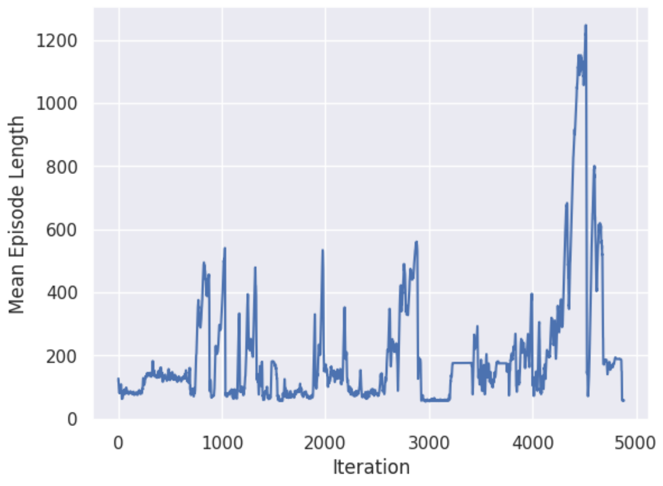

Fig. 11. Average Episode Length Per Iteration

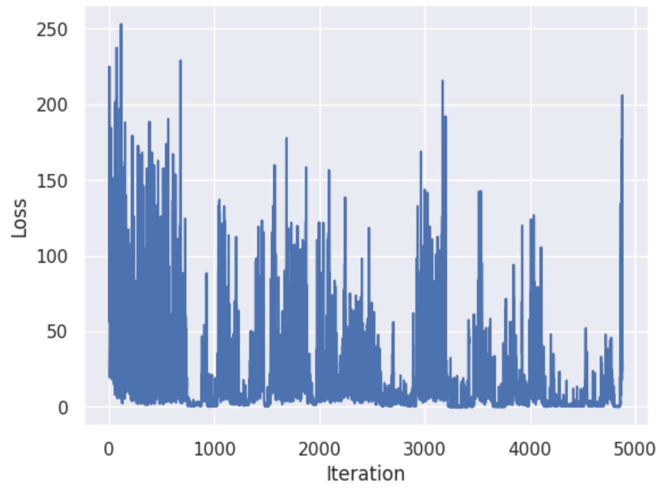

Fig. 12. Loss Per Iteration

make the game run no faster than that FPS (approximately 60 FPS). This is to prevent the game from running faster than human players can keep up with and ensure consistent gameplay. This is a problem because it prevents the reinforcement agent from running efficiently as its time steps are fixed to how fast the game engine is running. So even though the agent may be able to run hundreds of time steps a second, it is forced to run at a max of approximately 60 FPS or fewer.

Conversely, when the game engine runs slower than normal, issues in the game engine may arise. While variable time stepping fixes the problem of the game engine never rendering updates when running slowly, it doesn't fully fix an issue with the physics engine acting in unintended ways. Some update logic in Celeste utilizes the real time passed since the last update in its calculations. This means that when longer than usual periods pass between successive updates, the physic calculations in Celeste may be abnormally large.

Most of the stepping logic for Celeste is protected code and difficult to modify, making it hard to change to support a more simulator like game loop that a reinforcement learning agent can use. My attempts to change this part of the game loop were futile, but if this could be successfully done, it would likely result in quicker agent training and lead to better results as the agent would be able to train for more time steps in the same amount of real world time.

V. Conclusion

Through this independent study, I was able to learn much about reinforcement learning. I explored both implementing reinforcement algorithms from scratch and applying reinforcement learning libraries toward a video game. Implementing some algorithms from scratch gave me a good foundational understanding of reinforcement learning. These implementation problems taught me much about tabular methods and gave me a sense of the difficulties with reinforcement learning. For the project, while the pipeline was difficult to initially develop as there is not much documentation on how to mod Celeste, the integration between the Python component and C# mod ended up being seamless. The main issue with getting the agent to work with Celeste was with making sure the rewards were correctly engineered. Additionally, getting the agent to train in a time efficient manner made it difficult to get results. Overall, while the results of the project were not amazing, further work could be done to fine-tune and improve the pipeline.

References

[1] OpenAI, "Introducing chatgpt," Nov 2022. [Online]. Available: https://openai.com/blog/chatgpt

[2] D. Silver, "Introduction to reinforcement learning with david silver," May 2015. [Online]. Available: https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, 2020.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347