CITS3001 Project: The Performance Of AI Agents in *Super Mario Bros.*

Connor Wallis (22506057) & Reilly Evans (23615971)

Introduction

One of the primary purposes of computers over their history has been to automate tasks that humans normally perform. Many different methods of creating agents for this purpose have been designed, from simple rule-based programs to complex machine learning algorithms. In this report we will construct two agents for the task of playing levels from the Nintendo Entertainment System game *Super Mario Bros*: one rule-based agent implemented by hand, and one Proximal Policy Optimisation (PPO) agent trained using stable baselines. We will compare the performance of these two agents in a variety of areas, demonstrating the strengths and weaknesses of each approach. In doing this, we will determine which agent is more suited for the task of playing the game.

Analysis

The two agents we will be comparing are a rule-based agent and a PPO agent. These agents will interact with Super Mario Bros. using the methods in the gym-super-mario-bros package (Kauten, 2018). Our first agent is a rule-based agent, using simple Python functions to make decisions. We chose this algorithm for its simplicity in implementation, and for the ability to compare it with the more complex reinforcement learning agents. The rule-based agent is a simple Python script which reads in data corresponding to the game screen, alongside other information recorded by the package such as Mario's coordinates and the number of lives he has, then chooses what combination of buttons to input based on that data. The rules used to determine what inputs are made were coded by hand over the course of this project's development, with them being tweaked manually in response to testing the agent starting from World 1-1, the game's first level, with 3 lives. For example, if the agent detects that a Goomba is within 70 pixels in front of it and the agent is not airborne, it will press the jump and run buttons and hold them down for the next few frames to ensure it jumps over the enemy. Identifying the different platforms and enemies was done in part with code developed by Lauren Gee (2023), with some modifications to increase the range of enemies and blocks the agent could identify. This agent uses the Super-Mario-Bros-v0 environment from the gym package, enabling the full graphics of the NES game rather than simplifying them like the other available environments do.

For our second agent, we used the Proximal Policy Optimization (PPO) reinforcement learning algorithm alongside the Convolutional Neural Network (CNN) policy from stable-baselines3. We decided on this algorithm as it was in line with the projects scope, being a more simple, stable, and efficient algorithm compared to others. We elected to use the CNN policy as it is suited towards grid-like data – in our case, images of the game screen, which are represented as a grid of RGB values for each pixel. Like the Rule-based agent, our PPO agent read in the game screen observation alongside the package's additional information. However, in contrast to the rule-based agent, the game screen observation data is reduced

to limit the amount of data fed to the agent (to reduce computational load). This was achieved via grayscale of the three RGB channels into a single channel. The agent will then iterate through steps or 'frames' of the game, analysing four game screens at a time. This was achieved through the frame-stacking technique, layering four frames into one in order to give the agent game screens with a sense of motion. After the agent analysed, it would select a decision based off its frame of understanding (model). The decisions were also restricted to 7 possible simple actions, removing rarely used actions like jumping left or crouching, reducing the computational stress and experimentation of the agent. Over millions of iterations of the analyse, predict, learn loop, the agent refined its model attempting to maximise the reward achieved by its predictions.

Selecting the optimal PPO hyperparameters and environment settings proved to be a challenge for our project. As we used newer versions of stable-baselines3 and PyTorch, we believe there were some incompatibilities and lack of publicly available information regarding hyperparameters for a super-mario-bros-gym environment. After 70 iterations of adjusting hyperparameters such as entropy coefficient, learning rate, discount factor, the learning policy, as well as the environment itself (we tried using the simpler super-mario-bros-gym-v3), we had to accept the underwhelming results of occasionally beating World 1-1.

The reward value used for training the PPO agent (and comparing its performance with the rule-based agent) is the one that is included in the gym-super-mario-bros. package. This reward is calculated for each step as

$$r = x1 - x0 + c0 - c1 - d$$

where r is the reward value for that step, x0 and x1 are Mario's x position before and after the step, c0 and c1 are the value of the in-game timer before and after the step, and d is -15 if Mario died during the step or 0 otherwise. The reward cannot be outside of the range -15 to 15 (Kauten, 2018). This results in the agent prioritising going right as quickly as possible while avoiding death. Starting from World 1-1 with 3 lives, the rule-based agent earns a total reward of 4126, consistently beating World 1-1 but quickly dying at the start of World 1-2. Early in the PPO agent's training at around 400k steps it would achieve an average reward of 1750. After 4 million steps, the PPO agent would plateau at roughly 1600-1800 average reward per 3 lives. It would go on to occasionally beat World 1-1 with a

With the PPO algorithm and the above reward function, we noticed many strange behaviours. For example, after millions of iterations, sometimes the agent would develop a local optimum such as running into the second pipe indefinitely. This led to us finding a time limit wrapper function for our environment from ClarityCoders (2022). This time limit wrapper indirectly modified the reward function, associating a death penalty for the agent if it did not complete the level in time.

Besides the differences in total reward, there are several other differences in the agents that affect their usefulness. One of the rule-based agent's benefits is its consistency. Neither the agent nor *Super Mario Bros.* itself have any random elements; it will always perform the exact same with no chance of random elements affecting its performance. This means that

running the code once is sufficient to gauge its performance, as opposed to the PPO agent where many trials may be necessary to confirm that its training paid off.

Another notable benefit of the rule-based agent is that it is easy to understand and quick to adjust if something is wrong. If a change to the agent is necessary, e.g. it needs to hold the jump button slightly longer to clear a large gap, the code can be directly edited and reran immediately to see the changes. On the other hand, to see any results from editing the PPO agent it needs to retrain its own model. In our case, we trained the PPO agent on a Nvidia RTX 3070ti using its CUDA cores via PyTorch. Although we had one of the best graphics cards on the market, training still took many days to complete and provided us with underwhelming results.

The obvious disadvantage of the rule-based agent is that it is incapable of learning from its training directly. The agent was primarily designed for beating World 1-1, a level which uses the standard overworld physics; if made to try and play an underwater level like World 2-2 where the controls are changed it would make very little progress due to not knowing how to swim. This will not change unless we directly program in the ability for it to recognise the swimming controls and underwater physics. In contrast, the PPO agent will eventually learn to beat these levels if given enough time to train, with no intervention from the programmer.

Although the PPO agent had an underwhelming outcome, it still posed some strengths in comparison to the rule-based agent. The first of these strengths is that the agent displayed some level of adaptability. The agent would go on to seemingly understand that enemies need to be avoided and pipes/holes need to be jumped over. Given more mechanics, the model would adapt and develop strategies to overcome these if given sufficient training.

Another example of an advantage of the PPO agent is its property of optimisation. As the agent is trying to maximise its reward, and the reward function considers the remaining time, the agent is trying to minimise the time it takes to complete the stage. Therefore, the agent will attempt to create the fastest route possible for completing the stage. As our PPO agent was early in its training, it beat World 1-1 with 1918 steps. The rule-based agent beat World 1-1 in 1618 steps. We believe this was due to a lack of training time, as the PPO agent was still improving its efficiency at a rapid rate.

A final example of an advantage from the PPO agent is that it has a degree of generalisation. When posed with new unseen levels, the agent can extrapolate the mechanics of the stage from previous knowledge. This can be observed when the agent completes World 1-1 after having said level as the one piece of training data it uses. Upon entering the new underground themed World 1-2 and its blue colour palette, the agent will jump over the Goombas and the tall pillars as it learned previously from World 1-1.

However, not all is a positive with the PPO agent. Besides the issue with training time mentioned above, another notable disadvantage is that the stochastic nature of the PPO algorithm's model makes it hard to create tangible results. This is reduced as training progresses but sometimes left us questioning if the agent was learning at all depending on the entropy coefficient.

A final disadvantage of the PPO algorithm is with its debugging and interpretability. Due to the cryptic nature and performance metrics of neural networks, it's hard to decide if modifying a hyperparameter is improving the agent or not. At a minimum, it requires many hours of training to determine if a modification is beneficial or not. Compounding this is the stochasticity of the agent, which can cause great variations early in the agents learning.

Performance Metrics

While the gym-super-mario-bros package by default uses rightward progression as its primary metric of performance, this is not the only factor on which we can compare the agents. One potential alternate metric is to directly reward completing levels as quickly as possible. Under this metric we would provide large reward boosts at the end of levels based on their world/stage numbers, the time remaining on the in-game timer, and the number of steps since the agent was initialised at the start of the game, while falling back on rewarding moving right quickly within individual levels. While this at first seems to not be very distinct from the existing reward function, it benefits in rewarding the use of subareas to skip parts of a level. Many levels in the game, including World 1-1 where our agents were primarily trained, have pipes or vines that lead to coin-filled bonus rooms. These rooms then return Mario to a later part of the level, skipping a large portion of platforming. While these shortcuts make for faster level completion, they result in less rightward movement overall and thus are discouraged by the current reward function; by directly rewarding quick stage completion we encourage the agents to utilise these shortcuts. This benefit is emphasised even more with the Warp Zones, rare rooms that can allow the agent to skip entire worlds. Rewarding agents who manage to locate these will promote an agent that gets to the end of the game as fast as possible. However, this metric's usefulness is limited on our agents due to them not getting very far into the game; the rule-based agent barely makes it past the start of World 1-2, while PPO only occasionally completes World 1-1.

Another potential metric for our agents is points. Like most games of the time, *Super Mario Bros*. includes an arcade-style points system that rewards various beneficial actions, like collecting coins and having a lot of time left on the in-game timer, with points. By tying the reward function to how many points a given action earns, we will be measuring our agent's performance in a unique way. Rewarding points encourages many unique behaviours that maximise points, such as kicking Koopa Troopa shells into other enemies or seeking out coin-filled blocks. The points given for time remaining at the level's end also ensure reaching the flagpole is still a priority, so the agents should still progress through the game. However, one change that must be made for this metric is to either restrict the agent to 1 life for each attempt or apply a heavy penalty to dying. The reason for this restriction is that *Super Mario Bros.* has a method through which a player can gain infinite extra lives by bouncing on Koopa shells on a staircase endlessly. Without one of these changes an agent could be incentivised to stall on one of the levels where this is possible, endlessly gaining points and lives but never progressing through the game or getting a game over. Additionally, the reward function will need to provide some positive reward to moving right, albeit less than the default function, in order to ensure Mario moves far enough into the level to start earning points and give the agents some direction for how to improve. Without this change, untrained agents will earn no reward as they move erratically at the start of the level (where there is nothing that can earn them points), making early training progress very slow. From 1-1 the rule-based agent earns a total of 21400 points, while the PPO agent achieves 17900

points; this difference is mainly due to the rule based agent completing 1-1 more quickly than the PPO agent.

Visualisation/Debugging

Many techniques were used to help visualise the decision process of our agents. For the rule-based agent, statements were added to the code that printed statistics to the terminal that we could use to identify what information the agent was accessing and using to make decisions. Gee's code (2023) already included some commands of this nature, printing lists of locations for each identified entity to the terminal with each step it made. We added statements to print when the agent considered itself to not be mid-air, as well as directly stating what action it performed each frame.

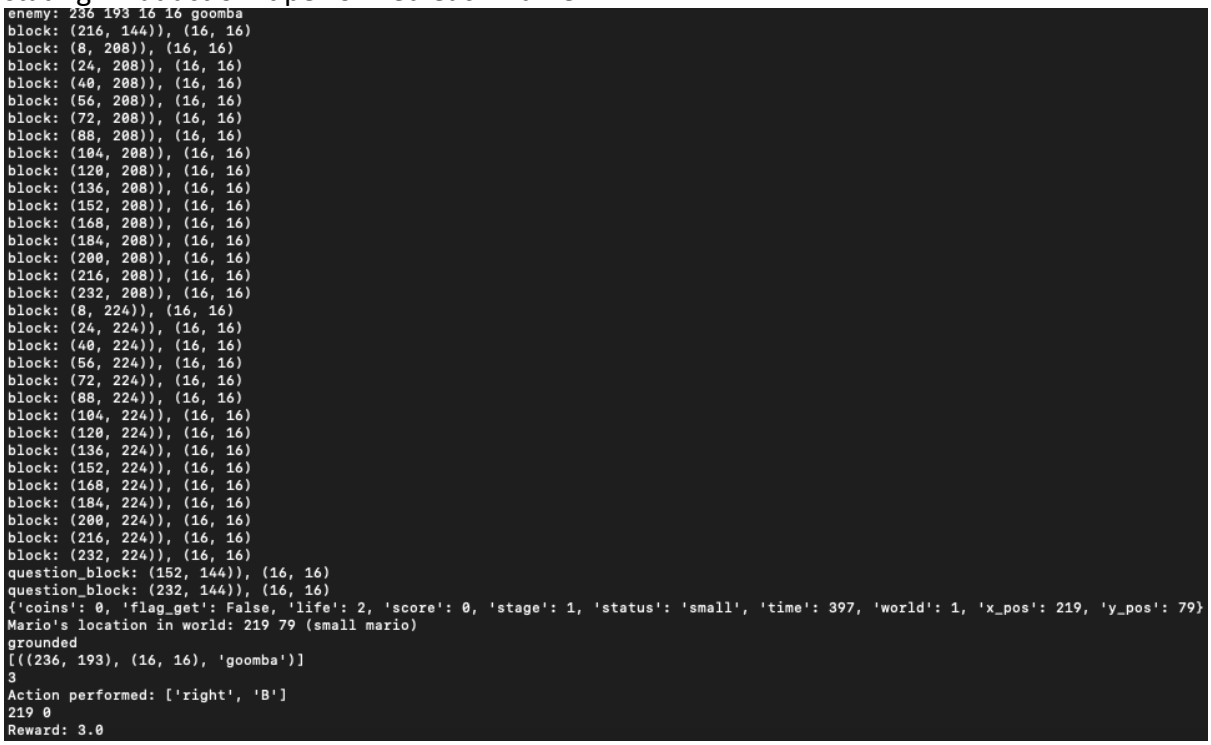

Figure 1: a sample of the details printed to the terminal by the rule based agent while we tested it, including details like block locations and reward

Another trick used to help visualise the decision-making process was to freeze the game the instant the agent decided to jump, giving us time to analyse the game's state and see what conditions made it decide to do so. This was achieved through the novel method of printing thousands of lines to the terminal, forcing the agent to wait until they were printed while we analysed the game.

For the PPO algorithm, we were using limited visualisation techniques early on. After running into many hurdles in the development of the PPO model, we adopted the Monitor wrapper from stable-baselines3. This gave us a great amount of insight into what was going wrong with our models, as we could not stay awake all night to monitor their progress in person.

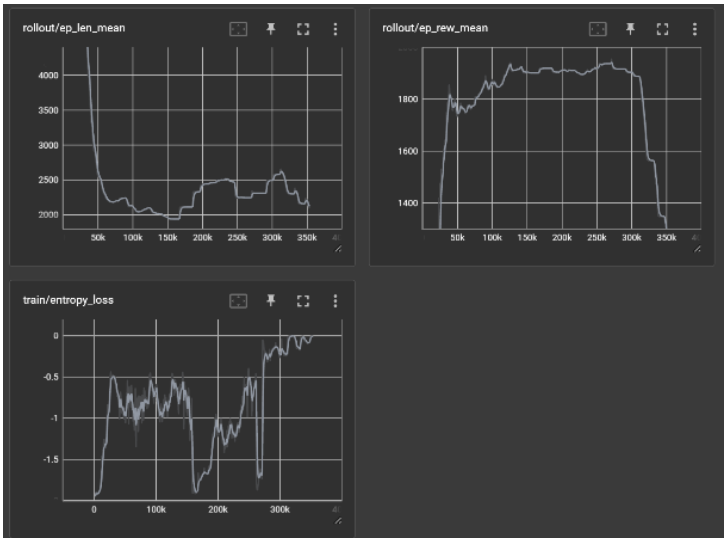

Figure 2: the rollout/ep_rew_mean (average episode reward) graph depicts our model falling into a local optimum, dying as fast as possible on repeat.

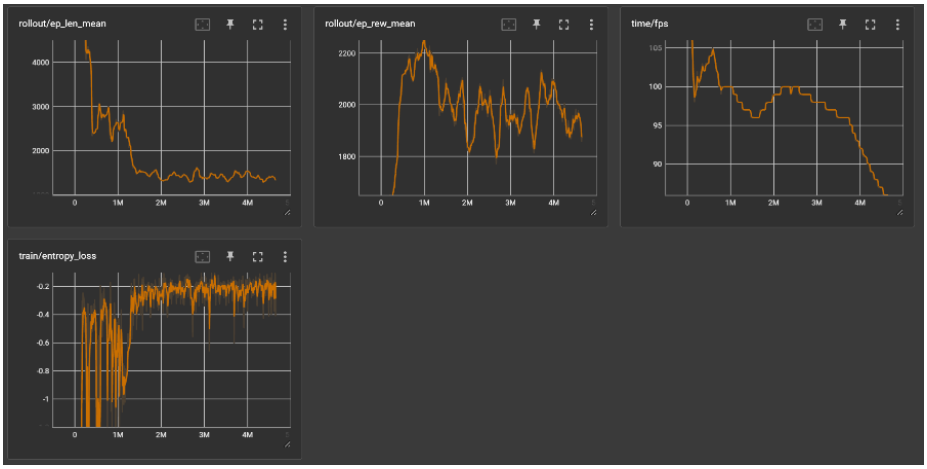

Figure 3: a set of graphs that display our most stable model, using a learning rate of 0.00001. The model plateaued after roughly 1 million steps (note the ep_rew_mean begins at 1600). Notice the entropy loss stabilising.

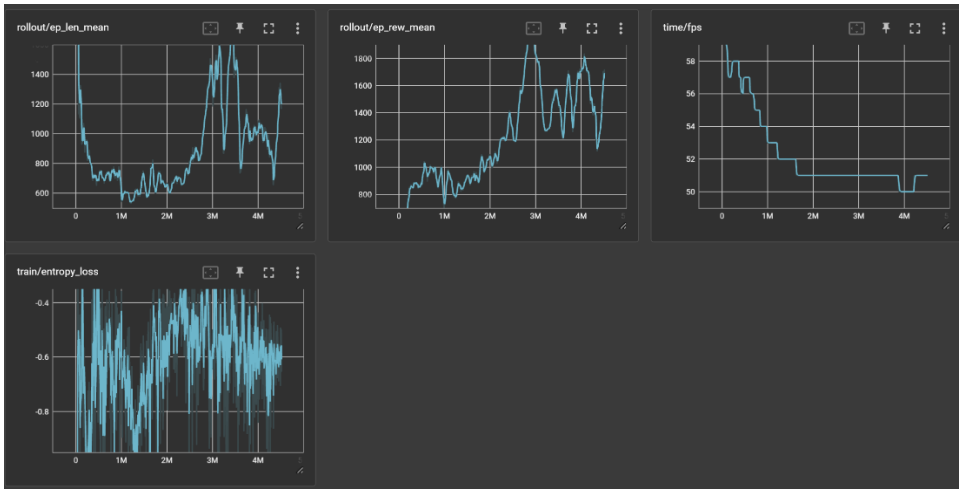

Figure 4: Our best model – after discovering some reward issues involving 1-2, we isolated training to 1-1. Although the ep_rew_mean seems lower, the model performed much better overall and can deterministically reach the final part of 1-1. Training had to be cut off early but we believe this model could deterministically beat the level given more training.

Additionally, some shorter hand monitoring to ensure the learning was occurring involved console printing as well. The following image is an array of statistics provided by stable-baselines3's libraries.

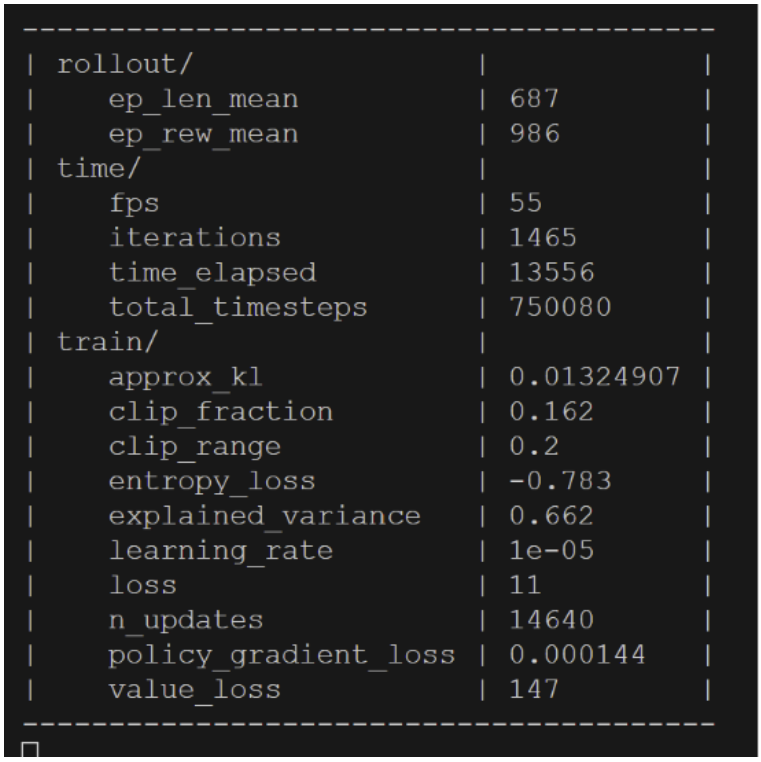

Figure 5: the statistics returned by stable-baselines3 for one of our models, showing statistics like number of iterations and learning rate.

References

- Kauten, C. (2018). *Super Mario Bros for OpenAI Gym*. GitHub. Retrieved October 10, 2023, from https://github.com/Kautenja/gym-super-mario-bros
- Gee, L. (2023). *mario_locate_objects.py*. UWA Learning Management System. Retrieved October 5, 2023, from https://lms.uwa.edu.au/bbcswebdav/pid-3405777-dt-content-rid-43562900_1/xid-43562900_1
- ClarityCoders. (2022). *Mario PPO*. GitHub. Retrieved October 15, 2023, from https://github.com/ClarityCoders/MarioPPO/tree/master
- Nicholas, R. (2022) *Build an Mario AI Model with Python | Gaming Reinforcement Learning*. Youtube.com. Retrieved October 10, 2023, from https://www.youtube.com/watch?v=2eeYqJ0uBKE
- *Super Mario Bros. with Stable-Baseline3 PPO*. (n.d.). Kaggle.com. https://www.kaggle.com/code/deeplyai/super-mario-bros-with-stable-baseline3-ppo
- *Reinforcement learning in Super Mario bros*. (2022, May 28). The Ops Community. https://community.ops.io/akilesh/reinforcement-learning-in-super-mario-bros-48a