CITS3001 Super Mario Project

Alex Hawking (23354512) and Connor Grayden (23349066)

October 16, 2023

1 Introduction

This project utilizes deep reinforcement learning techniques to teach an agent to play the game Super Mario Bros using the gym-super-mario-bros environment [2]. We have created two agents, one using PPO and the other with DDQN. The purpose of this project is to compare these two algorithms and see which has better performance.

2 Algorithms

2.1 Proximal Policy

2.1.1 Overview

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm introduced by the OpenAI team in 2017 [1]. It was designed to address several shortcomings in traditional policy gradient methods, including inefficiency in policy update methods and issues surrounding stability.

2.1.2 Basic Mechanics

The foundational architecture of PPO is based upon two neural networks:

- **Actor Network:** This network facilitates action selection. Given a specific state, it estimates the most favorable action.
- **Critic Network:** The Critic Network is used to estimate the expected cumulative rewards from a given state.

These networks are then optimized using a combined loss function:

$$L_t^{\text{CLIP}+V_F+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{\text{CLIP}}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right]$$

This function comprises of three parts:

- **Clipped Objective for Actor:** The L^{clip} objective in PPO aims to optimize the policy by leveraging the advantage of taken actions, with the clipping mechanism ensuring stability by preventing overly large policy updates that deviate excessively from the previous policy.

$$L^{\text{clip}}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta).\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon).\hat{A}_t)$$

- **Value Function Loss for the Critic:** This component gauges the accuracy of the Critic Network by contrasting the predicted value $V(s)$ against the actual observed return from that state. The loss indicates how well the Critic Network models the value function.

$$L^{\text{VF}} = \mathbb{E}[(V(s) - R_t)^2] \tag{1}$$

- **Entropy Regularization (S):** An important way to add randomness and encourage exploration in the early stages of training, encouraging the actor to explore the environment. However, as the policy becomes more confident in selecting actions, the entropy gets lower, leading to less exploration.

2.1.3 Comparison to similar algorithms

Given the complexity of a game like Super Mario Bros, selecting the best policy optimization algorithm is very important. Vanilla Policy Gradient (VPG) and Trust Region Policy Optimization (TRPO) are 2 alternative to PPO that have traditionally dominated this space.

VPG, which optimizes the expected return as

$$\nabla_\theta L(\theta) = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t \right]$$

can sometimes yield substantial policy updates, leading to sporadic agent behaviors. Conversely, TRPO, symbolized by its objective

$$L(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right]$$

with the trust region constraint, is extremely stable but requires significant computational resources.

Given these options, Proximal Policy Optimization (PPO) is the best choice. By employing the clipped objective

$$L_{\text{PPO-Clip}}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t, \text{clip} \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t} , 1-\epsilon, 1+\epsilon \right) \hat{A}_t \right) \right]$$

PPO ensures stable policy updates. Given the diverse challenges in Super Mario Bros, PPO's blend of stability and computational efficiency crowns it the optimal choice. [4]

2.2 Double Deep Q-Network (DDQN)

2.2.1 Overview

Double Deep Q-Network (DDQN), formulated by Hasselt, Guez and Silver in 2015 [6], is an enhancement of the Deep Q-Network (DQN) algorithm created by DeepMind in 2016 [3], which is a popular reinforcement learning method for solving a variety of tasks. DDQN was introduced to address some of the limitations and instability issues associated with the original DQN. This particular DDQN algorithm is from Pytorch's tutorial [7].

The fundamental goal of DDQN is to approximate the optimal action-value function, often denoted as Q(s, a), which represents the expected cumulative reward when taking action a in state s and following a specific policy. This function is crucial for making decisions in an environment where an agent interacts with its surroundings.

2.2.2 Basic Mechanics

DDQN maintains two separate neural networks: a target network and an online network. These networks have identical architectures but different parameters. The online network is used to select actions and estimate Q-values during the agent's interactions with the environment, while the target network is used to provide target Q-values against which the online network's Q-values are compared.

The key difference in DDQN lies in its approach to mitigating the overestimation of Q-values that can occur in the original DQN algorithm. In DQN, the target Q-values used for training are obtained by selecting the action with the maximum Q-value from the online network, which can lead to overestimation bias. In DDQN, the target Q-values are obtained by selecting the action with the maximum Q-value from the target network, which is updated less frequently. This reduces the risk of overestimation through reaching a "true value" too fast, and improves the stability of the learning process.

The loss function used in DDQN is similar to the one used in DQN, and it aims to minimize the mean squared error between the predicted Q-values from the online network and the target Q-values. The loss is updated through backpropagation to improve the performance of the online network.

$$L(\theta) = \mathbb{E}\left[(Q_\theta(s,a) - (r + \gamma Q_{\theta^-}(s', \text{argmax}_{a'} Q_{\theta^-}(s',a'))))^2\right] \tag{2}$$

Where:

- $Q_\theta(s,a)$ is the Q-value predicted by the online network for state s and action a.
- r is the immediate reward obtained from taking action a in state s.
- γ is the discount factor.
- $Q_{\theta^-}(s', \text{argmax}_{a'} Q_{\theta^-}(s',a'))$ represents the target Q-value obtained from the target network for the next state s' and the action that maximizes Q-value in that state.

By using the target network's Q-values to create the target Q-values, DDQN reduces the overestimation bias and leads to more stable and reliable training. Overall, DDQN represents a valuable tool in reinforcement learning, especially for tasks involving decision-making and action selection. Its ability to address overestimation issues and enhance training stability makes it a reliable choice for various applications in reinforcement learning, and it outperforms the original DQN in these aspects.

3 Theoretical Comparison of Algorithms

Super Mario Bros demands a strategic choice of reinforcement learning algorithm to successfully and efficiently play the game. PPO and DDQN, as leading choices, have distinct attributes and mechanisms.

3.1 Method Type

PPO operates using an *actor-critic* approach, a subtype of policy-based methods. The actor is responsible for determining the policy and the critic evaluates the value of these states, assisting

in computing the advantage function which drives policy updates. This dual mechanism ensures a balanced and stable policy optimization. DDQN, on the other hand, adopts a *value-based* approach. It focuses on approximating the Q-values for state-action pairs, aiming to find the true value of taking a specific action in a particular state.

3.2 Stability

PPO's clipped objective function provides stability in the dynamic environment of Super Mario Bros by bounding policy updates, ensuring that the policy is neither too aggressive nor too conservative, maintaining a balance between exploration and exploitation. DDQN's stability comes from its double Q-learning mechanism, which reduces the overestimation bias found in traditional DQNs [6]. However, the inherent oscillations of value-based methods might render PPO's actor-critic stability more consistent in the varying terrains of Super Mario Bros.

3.3 Sample Efficiency & Experience Replay

PPO is generally more sample-efficient than DDQN. PPO often requires fewer interactions with the environment to achieve good performance. DDQN may require more samples to learn a good policy, making it appear slower in terms of training. DDQN typically uses experience replay, where it stores and samples past experiences to improve learning stability and efficiency. While experience replay can enhance learning, it requires more samples and time to collect sufficient experiences for effective training.

3.4 Convergence

The exploration-exploitation trade-off can affect training speed. DDQN primarily focuses on learning an action-value function and relies on epsilon-greedy exploration strategies, which can sometimes lead to slower convergence. PPO inherently balances exploration and exploitation through its policy optimization process, potentially leading to faster convergence. [5]

3.5 Conclusion

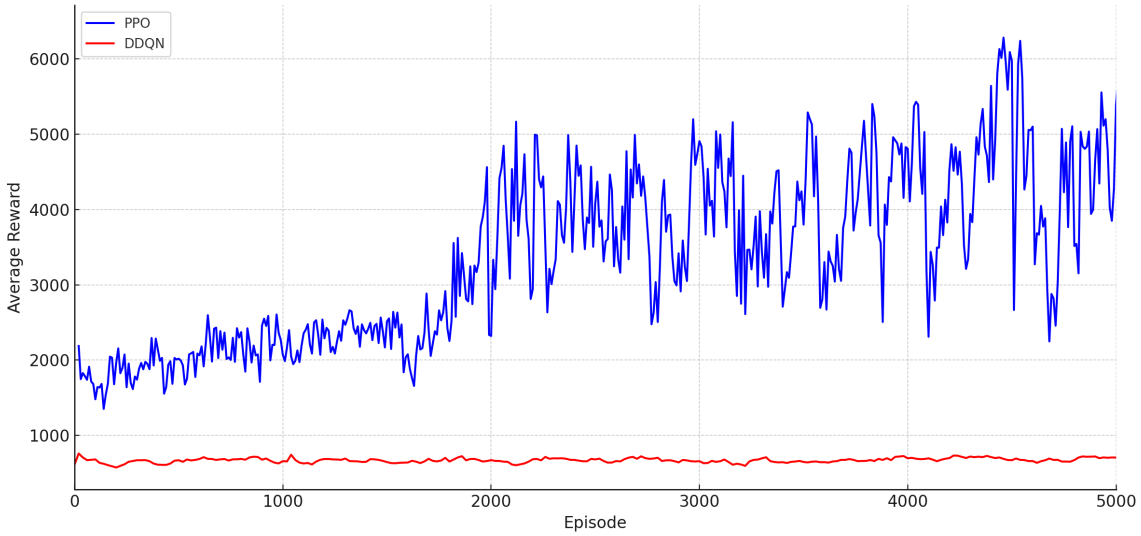
In summary, the unique strengths and challenges of PPO and DDQN make them good candidates for training on Super Mario Bros. However overall, PPO comes out on top, as its policy optimization, sample efficiency, stability and lack of experience replay makes it a better, faster and more effective choice of algorithm for this use-case.

4 Analysis

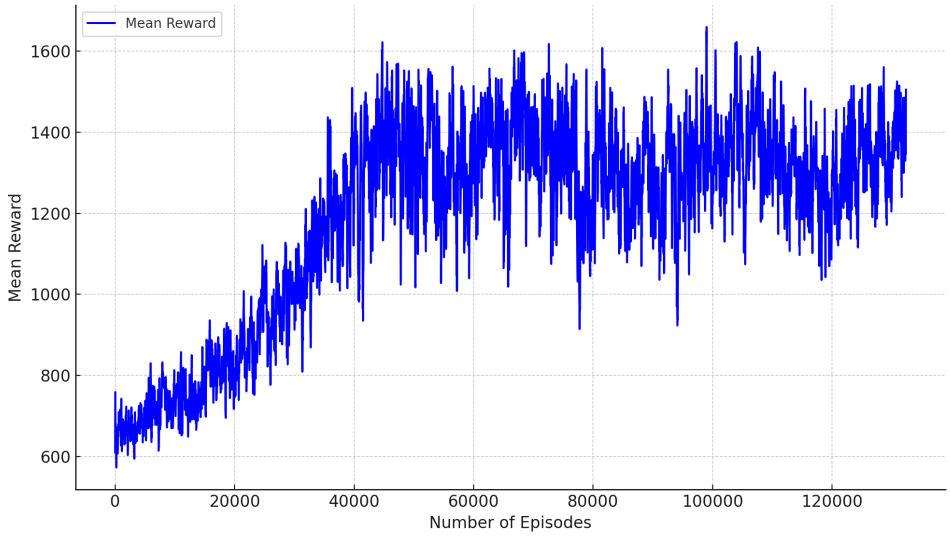
4.1 Performance on Trained Levels

For our initial comparisons we simply compared the rewards the model obtained over the episodes. Within the gym-super-mario-bros package [2], the agent is rewarded for moving to the right as fast as possible, and is penalized for dying. Essentially the faster the agent successfully navigates through the levels, the higher the reward.

We first compare the average reward every ten episodes with both PPO and DDQN, over 5000 episodes.



Given the extended training duration of DDQN, the following graph showcases the mean reward across 133,250 episodes, showing the slower training rate. Despite prolonged training in hopes of mirroring PPO's performance, DDQN appeared to stabilize around the scores of 1300-1400. This stagnation likely results from a combination of insufficient hyperparameter tuning and the limitations of DDQN for this particular application.



4.1.1 Limitations of DDQN

As can be seen in the graphs, DDQN did not perform as expected, this is likely due to poor management of hyperparameters. While I started with the same hyperparameters laid out in the Pytorch tutorial [7], the changes we made must not have been optimal, causing undesired results. However the hyperparameters presented in OpenAI's PPO paper [1] had sufficient values that needed very little tuning.

4.2 Model Scaling

We then compared how a model performed when it had been trained a certain number of times. To do this we took the saved model from n episodes and sampled it 100 times, calculating the mean reward for each. For PPO we used the model trained every 1,000 episodes, however due to the slower training rate of DDQN, we used a model trained every 10,000 steps.

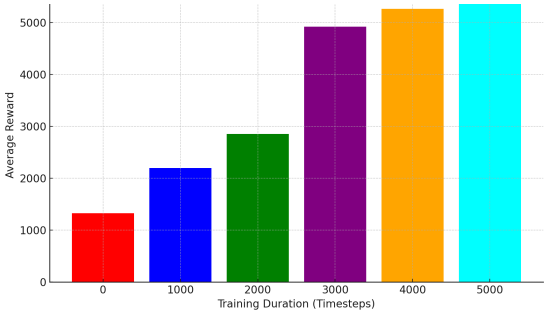


Figure 1: PPO

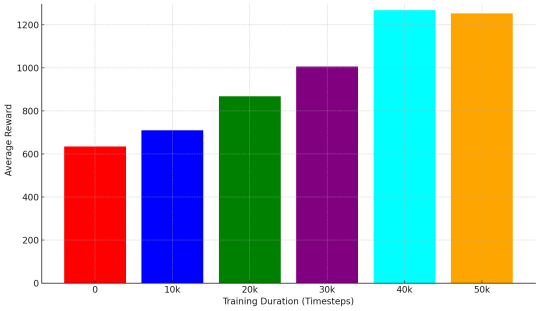


Figure 2: DDQN

4.3 Exploration on Unseen Levels

When we subjected our models to unseen levels, their performance revealed certain inherent challenges of reinforcement learning. Given that both our algorithms, PPO and DDQN, are anchored in reinforcement learning principles, they inevitably struggle with unseen environments, leading to substantially lower mean rewards. This highlights a fundamental drawback of reinforcement learning: its vulnerability to unfamiliar environments. A Rule-Based agent would perform significantly better in such scenarios. Owing to its design, a Rule-Based agent generalizes its reactions based on observations. In the context of PPO, the agent suggests actions contingent on specific states. When encountering previously unseen states, PPO struggles to determine the most fitting action. Similarly, DDQN evaluates scenarios using state-action pairs. The unfamiliarity with new states hampers its ability to pinpoint the most effective action.

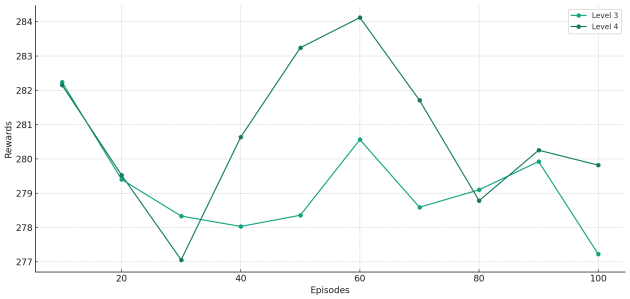


Figure 3: PPO

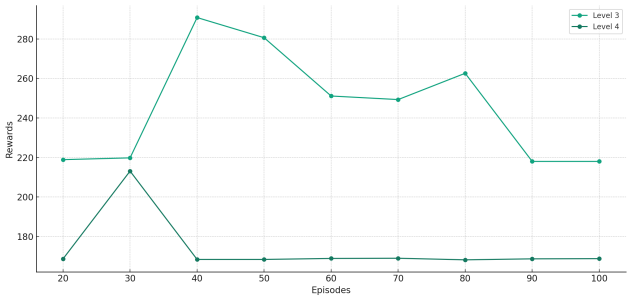


Figure 4: DDQN

4.4 Performance with Score as Metric

We then decided to train our agents using the score achieved as the reward mechanism, instead of progression throughout the level. With this metric the agent is rewarded for breaking bricks, collecting coins and defeating enemies, as well as a reward mechanism for if the player finished the level. However, this presents distinct challenges compared to simple rightward progression including:

- **Sparse and Delayed Rewards:** Actions and their positive outcomes are often distant, making it harder to associate them.
- **Expanded State Space:** Recognizing and interacting with coins, blocks, and various enemies increases the learning complexity.
- **Complexity in Achieving Rewards:** The agent might need a series of actions to achieve a reward, complicating the learning of which actions are most valuable.

These complexities challenge the learning capabilities of algorithms like DDQN and PPO when optimizing for score. This is showcased in the graph below, where the average rewards appear to be extremely randomized.

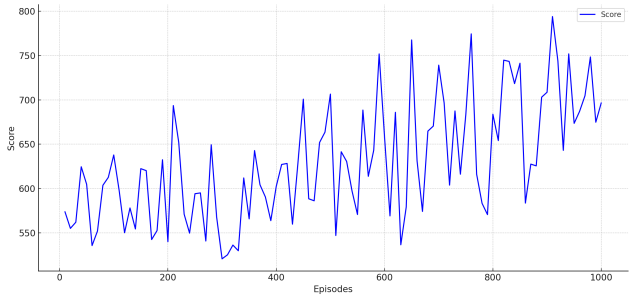


Figure 5: PPO

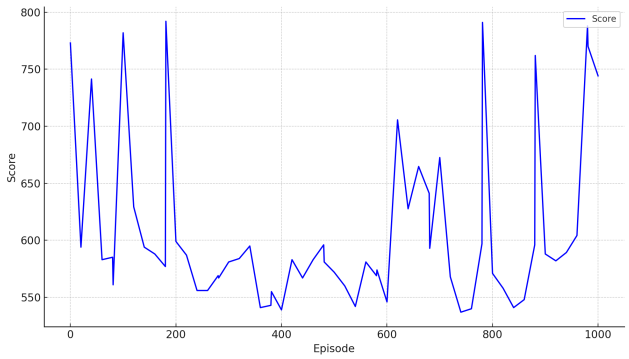


Figure 6: DDQN

4.5 Other Limitations

When employing DDQN and PPO to play Super Mario Bros. there are a number of limitations that we were not able to address for this project:

- **Limited Action Space:** We decided to limit our agents action space to 2 actions: *right* and *jump right*. This was done to increase efficiency whilst training (as our machines were not optimal for training on (macbook)). This limited the agent's capability to navigate intricate terrains or evade complex enemy patterns, especially in later levels.
- **Generalization:** Both DDQN and PPO may overfit to specific strategies suitable for early levels, making them less adaptable to the increasing game complexity.

- **Training Time:** Due to time and hardware constraints, we were not able to train our models to progress any further than the first few levels. This meant we were not able to test how the agent performed in later stages of the game.

In essence, while DDQN and PPO can achieve reasonable performance in initial levels, there are a number of contraints that can hinder progress throughout the game.

5 Visualization and Debugging

5.1 PPO Implementation

During the training of the PPO algorithm, two vital utility functions were employed to ensure effective monitoring and progression of the agent. These utility functions are elaborated upon below:

5.1.1 `save_checkpoint()`

This utility serves the primary purpose of persisting the state of the model at different checkpoints during training. The reasons for implementing this function are:

- **Recovery:** To restore the agent to a previously successful state in case of issues or failures in subsequent training sessions.
- **Evaluation:** To analyze the model's performance at different stages and understand its progression over time.
- **Transfer Learning:** The saved models can be used as starting points for related tasks, leveraging pre-existing knowledge.

5.1.2 `write_to_csv()`

The second utility is focused on recording the average rewards the agent achieves over specified intervals. The significance of this utility includes:

- **Performance Tracking:** By logging average rewards, it's feasible to monitor the agent's performance across training epochs.
- **Hyperparameter Tuning:** Observing reward trends aids in identifying whether certain hyperparameters need adjustments to enhance the agent's efficiency.
- **Convergence Analysis:** A steady increase in average rewards can indicate the algorithm's convergence towards an optimal policy.

Both utilities played a pivotal role in guiding the training process, evaluating the model's development, and ensuring the consistent improvement of the PPO agent.

5.2 DDQN Implementation

The training process integrates several components to efficiently train, monitor, and persist the agent's state. These components are detailed below:

5.2.1 MetricLogger Class

The `MetricLogger` class serves as the primary monitoring tool during the training phase. Its functionalities and purposes include:

- **Recording Metrics:** It is designed to track various metrics, ensuring comprehensive evaluation of the agent's behavior.
  - *Episode Rewards:* Monitors the rewards obtained by the agent in each episode.
  - *Episode Lengths:* Keeps track of the duration or number of steps taken in each episode.
  - *Losses:* Provides insights into how the model's predictions deviate from the actual results.
  - *Q-values:* Represents the expected future rewards for actions taken by the agent.
- **Visualization:** Enables real-time and post-training visualization to understand the agent's progression and areas of improvement.

5.2.2 Optimization with torch.optim

The `torch.optim` package facilitates the optimization process. The key roles of this package are:

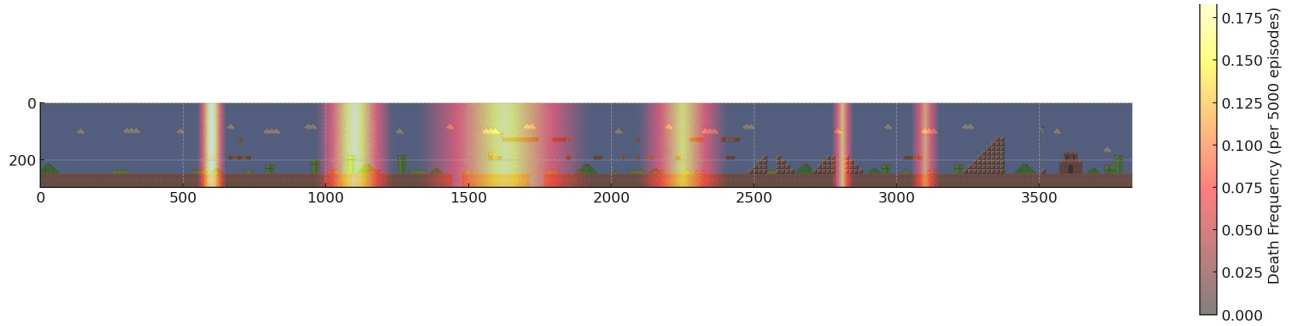
- **Stochastic Gradient Descent (SGD):** Employs SGD to adjust and refine the model's parameters based on the gradient of the loss function.
- **Parameter Updates:** Ensures consistent and effective update of the neural network's parameters, driving the agent towards optimal performance.

5.2.3 Model Persistence with torch.save

During the training loop, model persistence is crucial for various reasons:

- **Checkpointing:** The `torch.save` function is invoked to store the model's state at different episode lengths, allowing for recovery and evaluation at various stages.
- **Transfer Learning:** Saved models can serve as foundational states for training on related tasks in the future.
- **Evaluation:** Enables post-training analysis by loading models from specific episodes to analyze their performance.

All these components, in synergy, ensure an effective, monitorable, and resilient training process.



5.3 Death Heatmap

We created a death heatmap to visualize challenges faced by the Deep RL agents during training. We logged the agent's position each time it reaches a terminal state. Using the Python library `NumPy` for data aggregation and normalization we were able to figure out the most common and location and could then plot these using `Matplotlib`. A heatmap provides a graphical representation of where an agent most frequently fails or encounters obstacles. By analyzing these areas of high concentration, we can identify specific challenges the agent faces. We were able to use these insights to guide hyperparameter adjustments, such as modifying the learning rate and adjusting the exploration-exploitation trade-off. We found that as the agent died early in the level when using DDQN, we could adjust the ϵ-greedy policy to increase progression. In the case of PPO, the clipping range could also be adjusted based on the frequency of suboptimal policy updates in certain regions, as indicated by the heatmap. If we found the agent was frequently dying in certain locations, increasing the clipping range allowed the policy to explore further. We ended up excluding the code that creates the heatmap from our final project as we found it was computationally expensive and did not remain useful in later stages of training.

5.4 Experimentation

For each of the experiments discussed earlier, a variety of visualization and debugging tools were employed to elucidate the agent's decision-making process and to understand how optimization enhanced performance. These tools and observations include:

1. **CSV Logs:** Logs were maintained for each training session, simplifying the process of generating and analyzing graphs.

2. **Model Preservation:** The models were stored, ensuring data availability for future graphing exercises and evaluation.

3. **Observation during Training:** While training or running the agent on specific levels, its decisions were closely monitored. This provided a clearer perspective on which hyperparameters required tuning. For instance, if the agent struggled with a tall pipe and lacked the required jump height, boosting the exploration rate could prompt it to experiment with diverse actions and potentially find a path over the obstacle.

4. **System Resource Check:** At the inception of training, system resources were evaluated to preempt potential bottlenecks.

6 Conclusion

Our exploration of Proximal Policy Optimization (PPO) and Double Deep Q-Network (DDQN) in the context of playing Super Mario Bros has provided valuable insights into the strengths and limitations of these reinforcement learning algorithms. While both algorithms demonstrated reasonable performance, PPO emerged as the superior choice due to its policy optimization, sample efficiency, stability, and adaptability to dynamic environments. However, we also identified several challenges and limitations, including the struggle of both algorithms to generalize to unseen levels and their sensitivity to changes in the reward metric.

References

[1] Open AI. Proximal policy optimization. https://openai.com/research/openai-baselines-ppo, 2023.

[2] Kauten Christian. gym-super-mario-bros. https://github.com/Kautenja/gym-super-mario-bros/tree/master, 2023.

[3] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, et al. Deep q-learning from demonstrations. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

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

[5] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2018.

[6] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[7] Howard Wang Steven Guo Yuansong Feng, Suraj Subramanian. Train a mario-playing rl agent. 2020.