**Super Mario Bros Agent Analysis**

Rhianna Hepburn: 23340238

     Brittany Carlsson: 22752092

CITS3001: Agents, Algorithms and Artificial Intelligence

***Abstract:***

This paper provides a comprehensive analysis of the performance, strengths and weaknesses of three agents in the context of playing the Nintendo’s 1983 video game, Super Mario Bros. The three agents implemented are a hand-written rule-based agent, and two Stable Baseline implementations; Proximal Policy Optimisation (PPO) and a Deep-Q Network Reinforcement Learning algorithm.  The aim of this report is to determine the suitability of different machine learning algorithms in creating the most optimal game-playing agent.

**Summary of Performance Metrics Used**

The comparative analysis between our three AI implementations will focus on the following metrics:

1. **The percentage of level completed:** This data was collected by dividing the x-position of the agent at the time the episode ends by the x-position of the flag at level 1, stage 1. However, this metric operates under the assumption that the x-position of the game flag is the same for all worlds and levels.
2. **The percentage of valuable actions:** A valuable action has been defined as an action which results in the rewards function of the OpenAI gym being greater than 0. This excludes inefficient actions, such as static jumping, and provides an insight into how quickly and efficiently the agent moves through the environment.
3. **Rewards gained:** Our analysis considers the mean reward, standard deviation of rewards, and the coins collected by our agents.
4. **Performance on unseen levels:** The three algorithms were tested on random stages of Super Mario bros to determine their adaptability to unseen levels. The rule-based agent was tested on the majority of the overworld stages (Super Mario Wiki, 2023). To get a more accurate insight on the success of transfer learning, we trained two PPO and DQN agents only on World 1-level 1, and then tested it on random stages.
5. **Success of agents in large state-action spaces:** This metric compares the performance on random stages of two PPO and DQN agents trained on random stages.

For the PPO and DQN algorithms, these metrics were calculated by running our trained model through 100 episodes, while for rule-based this information was determined over 1 run of each episode, as the agent makes the same decisions every run.

**Rule-Based Implementation**

The rule-based agent employs a set of hand-written conditions that explicitly define what action the Mario agent should take, depending upon the upcoming ten pixels of the game screen. The algorithm scans ten pixels in front of, and below Mario (with the origin being the right most red pixel on Mario’s cap) to determine upcoming obstacles and changes in the environment. This algorithm has only been optimized for World 1, Stage 1 of the Super Mario Bros, although the logic has the potential to be scaled so Mario can complete multiple levels. This would involve increasing the number of colour-defined enemies, objects, and holes that the algorithm recognizes to suitably inform Mario’s actions.

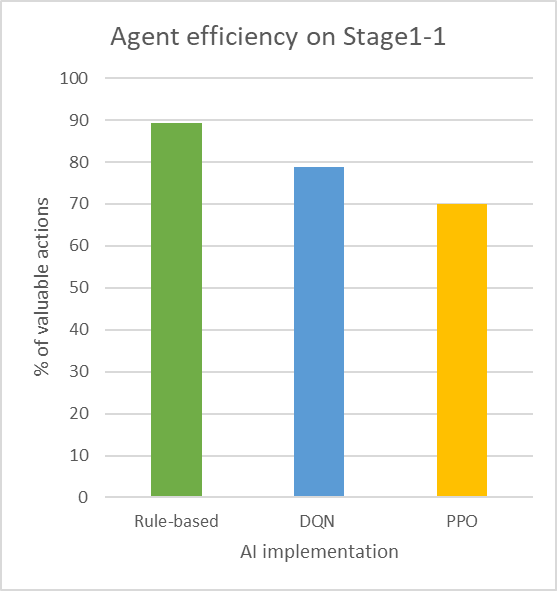
**Optimization, Debugging and Visualization Tools Utilised**

The rule-based agent was visualized, debugged, and optimized using these methods:

1. **Visual Studio Code Python3 debugger**: Through the use of breakpoints, this debugging technique provided clear insight into the stages of execution where Mario would make suboptimal moves. Screenshots would be taken before and after the breakpoints, and then Microsoft Paint would be used. This also allowed us to optimize the number of pixels to be scanned. Initially, 5 pixels were scanned, but through visualizing Mario’s actions at key stages of execution, it was evident that this value led to unnecessary actions and increased deaths. Hence, the optimal value was found to be 10 for object identification.
2. **Print statements:** These were used to provide insight into what the next 10 pixels in front of Mario were, and what Red, Green and Blue (RGB) values each pixel held.
3. **Microsoft Paint:** The screenshots from the agent playing the game were analysed in Microsoft Paint with the grid option. This provided insight and visualisation regarding the pixel distance from the right-most red pixel on Mario’s cap to the nearest obstacle, and informed the creation of rules for different objects.
4. **Super Mario Bros game via the Nintendo Switch Online game pass:** This was utilised to determine how high Mario was required to jump to get over pipes of all sizes, and to learn the optimal distance away from a pipe that was needed to make a successful jump.
5. **Performance functions:** This algorithm utilized the performance matrix provided by OpenAI gym, and analysed the info dictionary to determine how well Mario was performing in terms of total score. We then added further functionalities to provide greater insight. This included monitoring the number of moves, efficiency of moves, percentage of level completed, and a printed message describing the cause of episode termination. This included whether Mario had died, gotten stuck or captured the flag.

**Analysis of the Rule-Based Implementation**

A graph with blue bars

Description automatically generated****A key strength of the rule-based agent is the optimization and efficiency of its actions. Mario’s actions have been defined to ensure no unnecessary jump or move is made, which results in a fast completion time and a high percentage of valuable actions in comparison to the other two agents (Figure 1.1). The rule-based agent took Mario 103 seconds to complete level 1 and had 89.38% of its actions resulting in a positive reward.  Additional strengths of this algorithm include reduced training time, computational strain, and increased ability to optimize Mario’s performance. The rules-based approach allows high control over Mario’s actions, leading to a more successful agent in the levels the rules have been defined for, in comparison to reinforcement learning agents. This is evident in the rule-based agent being the only agent to complete Word 1, level 1 (Figure 1.2 and Figure 1.3).

***Figure 1.2*** *displays the level completion status (%) based on x-position at time of death, or flag capture for all agents*

***Figure 1.1*** *identifies the number of valuable actions taken by each agent as a percentage*

The weaknesses of this agent include its inability to adapt to unseen levels of the game, seen in the reduced completion, smaller average scores and increased deaths on unseen levels (Table 1.1, Table 1.2 and Figure 1.3). The agent requires code that takes into consideration every minor detail of each level to hard-code rules for Mario to progress. For example, for an unseen level such as World 2-1, Mario dies almost instantly from an approaching enemy as the algorithm has not yet analysed the new colour scheme (Table 1.1 and Table 1.2). Thus, creating an agent that could complete multiple worlds and stages of Super Mario Bros would be very time expensive, and with such a large state-action environment, this approach becomes infeasible in terms of space and time complexity. Hence, Reinforcement Learning agents, such as the PPO and DQN agent, are more A graph of different colored bars

Description automatically generatedsuitable algorithms for designing game playing agents in large state spaces.

**Figure 1.3** Displays a column graph representing the percentage of level completion by the Rule-Based agent on the selected overworld levels, where the completion status is based on the x position at the time of death or flag capture for each level based in Table 1.1. The legend is as shows: (World Stage, hence, 1 1 is World 1, Stage 1).

Another trend exemplifying the weakness of this agent in unseen levels is the decreased efficiency of actions. In unseen versions, there are regular instances of episodes with a greater number of moves made, but no resulting increase in the score of the episode. This correlation is evident in table 1.3, where the total moves is similar to that of Mario in world1-1, but the average score is nearly 400 points lower. However, it should be noted that the rule-based agent did have some success on unseen levels, such as World 8-1. The agent almost reached halfway before dying from an unaccounted enemy; this suggests that if the enemy had been considered, the agent had a high probability of completing the level. Further, the poor performance can mainly be attributed to unseen enemies. The majority of episodes terminated due to death, rather than Mario being stuck and not reacting to different sized pipes. Overall, this analysis provided insight into how this agent could be further modified to have the ability to pass non-boss, Overworld levels.

|  |  |  |  |
| --- | --- | --- | --- |
| Time | Score | Completed level | Total Moves |
| 257 | 220 | 23.83% | 2069.3 |

**Table 1.1** Statistics for Overworld Selection Using Version 0

A table with numbers and a number on it

Description automatically generated**Table 1.2** Averages the Overworld Levels Using Version 0

**Stable Baseline Implementations**

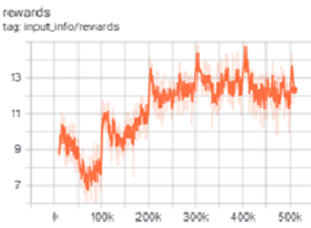
We trained and tested two Stable Baseline algorithms. This was aDQN Stable Baselines 2 implementation *(StableBaselines2, DQN 2018),* with optimization functions sourced from *(dvorjackz, MarioRL/paper.pdf at master · dvorjackz/mariorl - github 2020)* and a PPO stable baseline 3 implementation *(StableBaselines3, PPO 2021).* The individual analysis of the algorithms will focus on the agents trained on random stages, as we wanted to see how well the agents will perform in the full state-action space of the Super Mario Bros game.

**Visualization and debugging:**

Both algorithms utilized callbacks and tensorboard logs. The tensorboard logs were used to visualize the performance of the agent as it trained, the convergence of policies, and to provide insight into the correlation between the values of the parameters used, and how that affected the agent’s performance. Further, callbacks, sourced from Stable Baslines3, saved the current model every 10,000 time-steps, and were used to load the agent at different points in training to visualize its actions and determine the causes for varying performance.

**DQN Implementation and training analysis:**

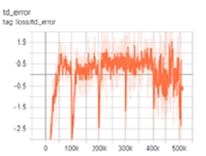
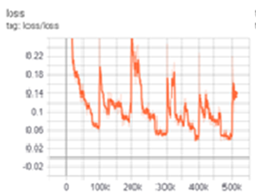
A screenshot of a graph

Description automatically generatedOur DQN agent (trained on random stages) showed clear signs of successful training, evident in the tensorboard logs. The rewards function showed a clear positive increase as training continued, although the trends in episode rewards were more ambiguous due to a large amount of noise (Figure 1.4, Figure 1.5).

***Figure 1.5*** *Identifies the reward per episode from training the DQN agent for 500,000 time-steps.*

***Figure 1.4*** *Identifies the rewards function from training the DQN agent for 500,000 time-steps.*

Further indicators of successful training is the decreasing loss, and convergence of the TD error towards 0, especially within the last 100,00 training iterations (Figure 1.6, Figure 1.7). This shows the algorithm’s ability to predict the true value of each state improved throughout training. This data also indicates the potential for improved performance of the agent with increased training, assuming the trends in the graphs continue.



***Figure 1.6*** *Identifies the loss per episode from training the DQN agent for 500,000 time-steps.*

***Figure 1.7*** *Identifies the TD error per episode from training the DQN agent for 500,000 time-steps.*

Our algorithm was optimized through the following functionalities:

* **Grayscale observations:** This was done to boost training speed.
* **Frame stacking:** This allows the algorithm to better capture movement and velocity in the environment and improves training speed.
* **Prioritized experience replay**: This has been found to improve performance and training efficiency by replaying important transitions (*dvorjackz, 2020*).
* **Double Q learning:** This was used in preference to standard DQN, as it can assist in preventing the overestimation of agent’s action values, and encourages the search for new highest values.
* **Dueling extensions:** This assists in improving policy evaluation for actions with similar values and can lead to increased stability and robustness of Q-value approximation.

**PPO Implementation and Training Analysis:**

Our PPO agent trained on random stages unexpectedly showed signs of decreased performance as training iterations increased (Figure 1.8, 1.9), seen in the mean episode rewards decreasing while the time taken for each episode increases. This correlates to the trend of the PPO agent getting “stuck” when moving through the level, seen in our experimental data.

A graph with red line

Description automatically generatedA graph with orange line

Description automatically generated

***Figure 1.9*** *Identifies the mean reward per 100 episodes from training the PPO agent for 500,000 time-steps.*

***Figure 1.8*** *Identifies the mean length of every 100 episodes from training the PPO agent for 500,000 time-steps.*

A graph with orange lines

Description automatically generatedThe weakness of this agent at this point in training is also seen in the ambiguity of the explained variance chart (Figure 1.10). This highlights the potential weakness of the algorithm’s value determination. The majority of the values are less than zero, showing the value function is worse than just predicting zero *(Oxbowerce, 2022)*, *(Scikit, 2023).* Further, the positive values are still far off 1, which would be a perfect prediction.

***Figure******1.10*** *Identifies the Explained\_Variance obtained from training the PPO agent for 500,000 time-steps.*

A graph with red line

Description automatically generatedInitially, we thought the drop in episode rewards could be due to the learning rate decreasing at a gradient that was too high. The rate of annealing could be experimented with further, but when we trained the same agent with a constant learning rate of 0.0001, the same drop in episode rewards occurred during training (Figure 1.11).

***Figure 1.11*** *Identifies the episode reward mean score for the PPO agent trained with a constant learning rate.*

A graph with red line

Description automatically generatedWhile this suggests the learning rate isn’t the main cause for this performance drop, a more gradual annealing process would allow for greater exploration and inject more randomness into the agent's movements; this may help it break out of the loop of getting stuck. Another way to overcome this issue is to modify the entropy coefficient. The entropy loss steeply approaching zero between 100,000-200,000 training iterations coincides with the drop in episode reward (Figure 1.12). This suggests that the level of randomness in the model needs to decrease at a slower rate.

***Figure 1.12*** *Identifies the entropy loss for the PPO agent trained with a linear annealing learning rate.*

**Performance Comparison of the Two RL Agents**

**Success on unseen levels**

**A graph of a number of people

Description automatically generated with medium confidence**Over 100 episodes, our DQN agent performs better than PPO on unseen stages. This is evident in the higher average rewards, the larger percentage of valuable actions, and greater percentage of the level being completed (Figure 1.13).

***Figure 1.13***  *Identifies the evaluation statistics for both RL agents over 100 episodes.*

The biggest performance contrast is seen in the average number of steps taken, and as a result, the percentage of valuable actions. Through watching the two agents play the game, it was clear this difference results from the PPO agent getting “stuck” in the environment, unable to jump over a pole or object. This is a consistent trend in all our experimental data, and resulted in the PPO agent being unable to move through the environment and score points, stuck in a loop of inefficient static jumping. The key difference between the implementation of our DQN and PPO algorithms that we believe contributed to this problem was the absence of frame stacking in our PPO algorithm. As noted in *(yumouwei, Super-mario-bros-reinforcement-learning 2023),* an increase in frame stacking correlates to an improved ability of Mario to jump over objects blocking his path, and can assist in capturing the velocity of objects. Our PPO algorithm had no frame stacking, while the DQN algorithm used a 4-frame stack to train with. As a result, the DQN agent had greater success at learning Mario’s movements in motion, and jumping over pipes before he hit them. The lack of frame stacking in our PPO implementation likely caused a reduced ability in the algorithm to learn Mario’s movements in the environment, and to recognize the correct time to jump to clear a pipe. Using frame stacking would have also improved the training time for our PPO agent and should be implemented in the future. However, both agents performed poorly, with no more than 15% of the course being completed on average. In comparison, the DQN agent trained on level 1 completed 33% of the course on average. This shows that the PPO and DQN agents' ability to respond to unseen environments is poor. However, this poor performance may also be arising from insufficient training time, as the agents do show signs of random actions rather than learnt skills, when playing level 1. This is exemplified in the agents’ inconsistency in jumping over holes and enemies, indicating these actions may be arising due to randomness rather than learnt actions at this stage in training.

**Performance comparison on trained levels**

A graph of a performance of a trained agent

Description automatically generatedFor DQN and PPO agents trained on World 1-1, DQN clearly outperforms PPO, completing more of the level with greater efficiency and rewards (Figure 1.14, 1.15). A likely cause of the DQN completing more of the level is the implementation of prioritized replay implemented only in our DQN algorithm, which has been found to improve training efficiency and improve performance (*dvorjackz, 2020*). Prioritized replay is especially suitable for Super Mario Bros, where large negative reward actions, such as dying, occur sparsely over the course of training. Having this optimization would have taught the DQN agent to place greater emphasis on avoiding death, through replaying this interaction with the environment, thus explaining its higher level of completion in comparison to PPO, who did not have this implemented. Lastly, DQN is considered more suited for discrete action spaces, such as the simple movement action space used in this game, while PPO is more suited for continuous action spaces. This could also have contributed to the observed performance difference in the two agents. Again, the problem of the PPO agent getting stuck is very clear in this experimental data, and heavily contributes to PPO’s poor results.

***Figure 1.14*** *Identifies the number of valuable actions the RL agents made over 100 episodes, as well as their completion status for the level, this is based on how far they were able to get (X-position) in the level*

The PPO agent took 24062 mean steps, while the DQN agent only took 49.67 mean steps and made it 14.7% further in the levels. In addition to differences in frame stacking, the DQN algorithm also commences training with an initial epsilon of 1, which decays to 0.1 This injects a high amount of randomness into the algorithm initially, and is something that might need to be reflected in the PPO implementation to avoid the drop in performance during training, and tendency to get stuck. This could be implemented by reducing the clipping range from 0.2 to 0.1, to enforce slower updates of policy.

A graph of a number of people

Description automatically generated

***Figure 1.15*** *Identifies the evaluation statistics for both RL agents over 100 episodes.*

**Success of agents in large state-action spaces**

A graph of a number of people

Description automatically generated with medium confidenceThe poor performance of both agents trained on random stages and tested on random stages indicates the need for greater training time to deal with the larger state-action space the agents are required to learn. The DQN agent only completed 16.83% of the level, while PPO completed 8.78%. The inefficiency of the PPO agent is highlighted most clearly in this experiment, with DQN having 84.67% of its actions leading to a positive reward, while the PPO only had 9.88%.

***Figure 1.16*** *Identifies the evaluation statistics for both RL agents over 100 episodes on Random Levels.*

**Conclusions**

Overall, the Rule Based agent proved to be the fastest, most efficient, and least error-prone agent of the three when completing world-1, stage-1, in relation to the current training levels that both Reinforcement Learning agents are at. However, a rules-based approach is infeasible when creating a game-playing agent suitable for the worlds and stages of Super Mario Bros. Thus, the DQN agent was the most successful algorithm in the context of this game, as it consistently outperformed the PPO agent on both seen and unseen levels. However, to provide a more certain conclusion, both the PPO and DQN agent need to undertake more training iterations. Further, the PPO algorithm could be further optimized with frame stacking, prioritized replay and adjusted parameters to improve performance.

**​​References​**

Stable Baselines. (2018). Welcome to stable baselines docs! - RL baselines made easy¶. Welcome to Stable Baselines docs! - RL Baselines Made Easy - Stable Baselines 2.10.3a0 documentation. <https://stable-baselines.readthedocs.io/en/master/>

Super Mario Wiki. (2023, September 13). *Super Mario Bros. List of levels*. <https://www.mariowiki.com/Super_Mario_Bros.#List_of_levels>

yumouwei. (2023, April 21). *Super-mario-bros-reinforcement-learning*. GitHub. <https://github.com/yumouwei/super-mario-bros-reinforcement-learning/blob/main/README.md>

dvorjackz. (2020, August 25). *MarioRL/paper.pdf at master · dvorjackz/mariorl - github*. GitHub. <https://github.com/dvorjackz/MarioRL/blob/master/paper.pdf>

StableBaselines3. (2021). *PPOℑ*. PPO - Stable Baselines3 2.2.0a7 documentation. <https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html>

StableBaselines2. (2018). *DQN¶*. DQN - Stable Baselines 2.10.3a0 documentation. <https://stable-baselines.readthedocs.io/en/master/modules/dqn.html>

Scikit. (2023). *3.3. metrics and scoring: Quantifying the quality of predictions*. scikit. <https://scikit-learn.org/stable/modules/model_evaluation.html#explained-variance-score>

7, O. (2022, October 15). *Understanding the tensorboard plots on a stable-baseline3’s PPO*. Data Science Stack Exchange. <https://datascience.stackexchange.com/questions/115243/understanding-the-tensorboard-plots-on-a-stable-baseline3s-ppo>

**APPENDIX**

**DQN (Larger Images from Figure 1.4)**

A graph with orange lines

Description automatically generated

***Figure 2.1*** *Identifies the Episode Reward, obtained from training the DQN agent for 500,000 time-steps.*

A graph with orange lines

Description automatically generated

***Figure 2.2*** *Identifies the Loss obtained from training the DQN agent for 500,000 time-steps.*

A graph with orange lines

Description automatically generated

***Figure 2.3*** *Identifies the Td\_Error, obtained from training the DQN agent for 500,000 time-steps.*

A graph with red lines

Description automatically generated

***Figure 2.4*** *Identifies the Rewards obtained from training the DQN agent for 500,000 time-steps.*

**PPO (Larger Images from Figure 1.5)**

A graph with orange lines

Description automatically generated

***Figure 2.5*** *Identifies the Value\_loss obtained from training the PPO agent for 500,000 time-steps.*

A graph with orange line

Description automatically generated

***Figure 2.6*** *Identifies the Ep\_Len\_Mean obtained from training the PPO agent for 500,000 time-steps.*

A graph with red line

Description automatically generated

***Figure 2.7*** *Identifies the Ep\_Rew\_Mean obtained from training the PPO agent for 500,000 time-steps.*

A graph with orange lines

Description automatically generated

***Figure 2.7*** *Identifies the Explained\_Variance obtained from training the PPO agent for 500,000 time-steps.*

Pixel pixelated image of a monkey and a green object

Description automatically generated

***Figure 2.8*** *Displays the debugging process of determining what colours, pixels or obstacles are 10 pixels ahead of Mario’s right most red cap pixel. This was done in conjunction with printing out the 10-pixel RGB values with a 10-pixel offset. The offset was used to ensure that Mario had enough time to react to the upcoming obstacle (especially enemies).*