CITS3001 – Algorithms, Agents and Artificial Intelligence

Super Mario Project Report

Mitchell Otley (23475725)
Jack Blackwood (23326698)

Contents:

Overview – DDQN Agent

To demonstrate Reinforcement Learning we utilised a semi-manually tuned and implemented DDQN Agent (Double Deep-Q Network) using the Python Library PyTorch. The goal of the agent is to play the first level of Super Mario Bros - by interpreting the Game and passing it to an Agent that will:

- Act optimally based on the environment state using Q-optimal.
- Retain past experiences, considering the current state, action, expected reward and subsequent state.

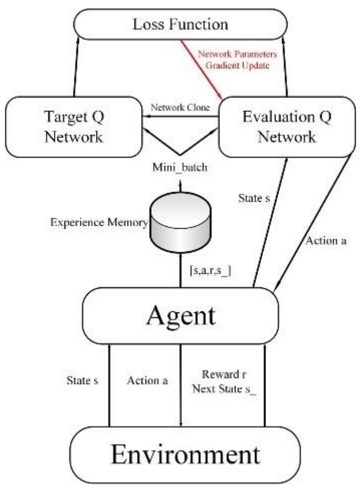
Initially, the agent knows nothing about the environment, so it tries out random actions to learn, as it gathers more knowledge, it reduces the frequency of random actions and starts to rely more on its learned optimal actions. The Agent is able to store these experiences in a Replay Buffer and once sufficient experiences have been gathered the Agent will 'learn' using the DDQN algorithm (Double Deep Q-Learning Network). This exploration-exploitation trade-off is known as Epsilon Greedy.

How it works:

The core of this implementation is two CNN (Convolutional Neural Networks), which are designed to correspond with the observation frames fed to it from the Game.

- The **"online" network** is the primary model that gets updated during training. It processes the game's images and predicts which action Mario should perform by outputting the Q-values for each action the agent can take in the action space. In our implementation, this is either MOVE_RIGHT or JUMP_RIGHT.
- The second **"target" network** is a copy of the online network but doesn't learn directly. It's used to help stabilise training by providing consistent predictions from the experience.

DDQN Architecture



The optimal action to take is derived from both the network's produced Q-value and a series of Temporal Difference (TD) calculations. These calculations use two Q-values: one as an estimate from the current knowledge (online network) and another as a target based on more stable knowledge (target network) to indicate how good a certain action is in a situation and predict how good the action will be in the next situation. Post-action the Agent will check how good the current knowledge of the expected reward was (a reward function defined by the Super Mario Gym primarily influenced by maximum X-Position) compared to the real outcome using a loss function (*SmoothL1Loss*) – it essentially measures how 'far-off' the predictions were. Depending on the action an optimiser is also used in our implementation for adjustments to improve the Agent's predictions for next time. After learning a defined number of steps with updated online knowledge, if the changes are good and favourable to greater reward, the agent copies the online knowledge to the target model for any future predictions, thus improving the model.

Discussion - DDQN:

The main advantage of using the two Online and Target networks is stable Q-value updates (optimal choices) – the prediction, evaluation and syncing should be better for constructing a more precise model, unlike a traditional DQN where the same network predicts the Q-values for both current and next states which can lead to an inherent bias as it leans towards actions it already expects to be beneficial. Although while training we found that this was not easily observed – for a proper comparison we would have to train a DQN also. It may also be highlighted that observed training showed no meritable improvement beyond a certain point, which may indicate that it was too precise in reaching an optimal state that was not associated with the end of the level.

By employing a fixed-size experience replay buffer, DDQN can recycle and learn from past experiences multiple times. This approach can save memory while training, but it ensures that it can make the most out of every collected sample. In theory, this should also let the DDQN agent achieve superior performance, reaching further in the level - with fewer experiences, making it both space and sample-efficient, eliminating recency bias, avoiding repetitive mistakes and better recognising patterns that lead to success. There are some issues we observed with this however, if we go to re-train the model from an existing checkpoint DDQN would always be in a state of 'no-experience' – the buffer is empty, so would require us to save the training buffer and reload it (which was quite large >2-3GB) – or give the model a 'burn-in' period to let it pick up on new experiences before it starts training. Also, due to the way our DDQN uniformly picks random samples, if good experiences are rare, it might not pick up on them to make the changes required – meaning the model doesn't improve as much as it could. One solution to this may be implementing a Priority System, so the agent learns favourably from the experiences it deems more 'valuable'.

While our current implementation is tailored for Super Mario Bros. 1-1, the architecture of the model can be quite adaptable, with minimal modification, or pure extra training time, it can be adapted to play other levels, or even entirely different games and applications. DDQN should have an innate ability to generalise across diverse inputs and environments given proper tuning and ample training purely based on its desire for optimal reward.

Number of Steps Trained: 2202768 Steps

Able to beat (1-1): No

Overview – SB3 PPO Agent

For our 2nd-Reinforcement Learning Algorithm, we implemented a StableBaselines3 PPO (Proximal Policy Optimisation) Agent. The StableBaselines3 library provides a robust platform ensuring that the intricacies of the PPO algorithm are mostly handled 'under-the-hood', while still maintaining an intuitive interface for configuration, monitoring, and training. The PPO agent operates on a different principle from the Q-learning paradigm (as in DDQN) - PPO offers a more balanced approach, blending on-policy-based and value-based techniques.

How it works:

At the heart of PPO is a neural network, like DDQN, this is a CNN. This is trained to output a probability distribution over actions, given an observed input state. PPO calculates the advantage for each action taken, which is essentially a measure of how better or worse an action was compared to the average action for a given state.

An action's advantage reflects its expected cumulative reward (its alignment with the desired outcome of reaching further in the level) while minimising a loss function. During training, PPO updates the policy to increase the probability of good actions (those with positive advantages and greater reward) and decrease the probability of bad actions (those with negative advantages). Of note, PPO limits the extent to which the policy can be changed in a single evaluation, this is to prevent changes that can destabilise learning. Unlike DDQN which uses a 2nd CNN, limitation on policy updates is achieved with a "clipped" objective function consisting of several parameters that we can tune. If the new policy deviates too much from the old policy (beyond a predefined parameter), the objective function is modified to discourage the change. SB3 provides default values that we can modify such as a 'learning_rate' (to signify how the model responds to changes in policy), 'entropy coefficient', 'n_steps', 'batch_size', with more advanced parameters that can drastically shape how the PPO agent will train. As the Model trains it will 'callback' saving the Current Policy to a Model File and self-evaluating to see if the Model performed better across its average reward.

Discussion – PPO

One large advantage to this Agent was its relative ease of implementation, as mentioned, the SB3 Library handles most of the high-level calculations, loss and value functions – providing a series of hyper-parameters for the tuning. This makes it an extremely approachable Agent, it also meant we could spend more time working with the gym-env tweaking and tuning. Mentioning hyper-parameters, this PPO agent was extremely sensitive to even slight changes - we conducted several experiments changing primarily the 'learning_rate' and 'n_steps' of the Agent to see how this would impact both training time, model size and relative effectiveness. Unlike DDQN, PPO is always training 'on-policy', which means that it learns from the most recent policy and most recent set of experiences. This can lead to faster convergence but requires more computation and sample inefficiency, instead of keeping past experiences stored in a buffer it keeps a record of new experiences as a batch. We found that with a smaller 'n_steps' and 'batch_size', the agent was able to learn more about a particular set of experiences - it took computationally less time individually but was overall slower due to increased frequency. With a smaller 'learning_rate' the changes made to policy about newly explored actions are smaller and more honed, but even slight changes could be observed in training metrics as large variance. To get this Agent to be effective, it is key to ensure that proper hyper-parameters are set – otherwise, it may train sub-optimally.

sb3-ppo-agent\logs_old_models\

Number of Steps Trained: 1300000 Steps

Able to beat (1-1): Yes – but win rate is marginal.

Overview – OpenCV Agent

Our OpenCV agent implements image template matching to follow rule-based logic to play Super Mario Bros. The heuristic used by our agent to choose the next action depends on whether Mario is on the ground or in the air (Figure 1 - Appendix). As opposed to a DDQN and PPO agent, the agent's heuristic uses predetermined logic and does not require any training of a model beforehand.

How it works:

Given a library of template images, OpenCV's template matching function employs 2D convolution to compare the template image against the input image and returns the coordinates of any instances of the template image appearing above a confidence threshold. Every time the agent needs to perform action determination, it uses template matching to gather the on-screen coordinates of the images (e.g. of Mario, of a goomba, etc.). It then uses these coordinates to follow the heuristic and decide on an appropriate action.

Some of the decision-making process requires knowledge of how Mario is moving in relation to the environment, so the algorithm will take an 'early snapshot' of the environment some frames before the next actions are chosen, to calculate the displacement of Mario and the on-screen enemies. We decided not to consider any items (mushrooms, stars, coins) in the heuristic, as the goal of our agent is to reach the end of the level as quickly as possible, without slowing down to collect items/coins.

Discussion - OpenCV

A rule-based agent has the advantage that it is simple to implement and modify the heuristic that it follows. If the agent is struggling to overcome a specific obstacle, the algorithm designer can manually edit the agent's logic to accustom the scenario. Additionally, a rule-based agent is not reliant on any training of a model on many iterations before being able to perform well. If the logic of the agent is generalised, it can be instantly applied to any unseen environment, so long as the environment is constrained within the logical context of the heuristic.

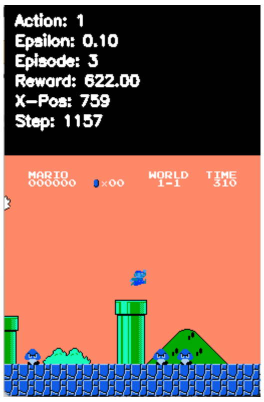
Due to the agent following the exact same action-determination tree diagram in every instance, the agent will always choose the same action in the same situation. This lack of variation means that if the agent is unable to overcome an obstacle beyond the scope of its' heuristic (e.g. a Piranha plant coming out of a pipe), it is restricted to never being able to pass that section of the level until human intervention alters the heuristic accordingly.

The template matching function used for our OpenCV image processing is resource-intensive, slowing down the speed at which Mario can traverse the level. As opposed to a machine learning algorithm, which can perform action determination at a quick pace once a model has been trained, the OpenCV agent needs to perform image processing to gather the relevant environment information before being able to consult its heuristic to make the next move.

Able to beat (1-1): Yes

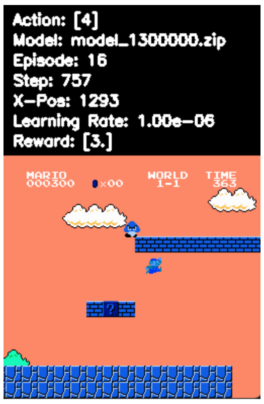
Visualisation/Debugging Techniques – DDQN & PPO

Both DDQN and PPO are a form of 'supervised learning' and as they train their models are constantly changing. We need some way to visualise the training as it is happening and evaluate a Model after its training has been completed.
DDQN has several key metrics that we must visualise during training. A dedicated `metrics` logger takes information from the Environment State and Networks to both store and plot for visual representation dynamically.



```
Action: 1
Epsilon: 0.10
Episode: 3
Reward: 622.00
X-Pos: 759
Step: 1157
```

DDQN Replay Agent Metrics



```
Action: [4]
Model: model_1300000.zip
Episode: 16
Step: 757
X-Pos: 1293
Learning Rate: 1.00e-06
Reward: [3.]
```

PPO Replay Agent Metrics

- **Training the Agent**
  - Creates a Log File (Figure 2 – Appendix), which can generate Graph Plots live:
    - The current episode
    - The current step-count
    - The current exploration-epsilon value
    - The current episode reward and mean Reward over 100 Episodes
    - Total Length of the episode and mean Length over 100 Episodes
    - Associated Mean Q-Value
    - Mean Loss determined by Loss Function
    - Maximum X Value reached in the Logging Interval
    - Time Δ for Logging Interval & Current Time Stamp
  - **reward_plot.jpg** – A plot across the episodes for Reward (Figure 3 – Appendix)
  - **q_plot.jpg** – Charting per-episode Q-Values (Figure 4 – Appendix)
  - **max_x_plot.jpg** – The Maximum X Value reached per episode.
  - **length_plot.jpg** – Total Length of each Episode (in steps)

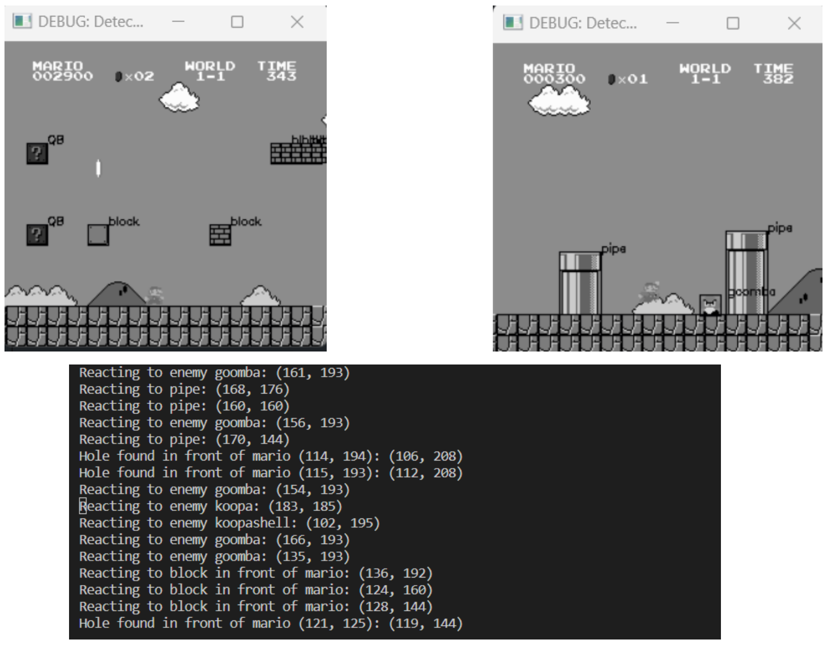
Replay Agent
  - Shares the Metrics Logger and creates the same log file and plots.
  - If executed with the '—render' flag the Game Space will be visible alongside Game Metrics specific to that episode. This is very useful for debugging as it lets us watch the Agent Play.
  - Using the '—checkpoint' flag will let us choose a specific Model to test.
  - Exploration is set to 0.1 – indicating that we are relying on the Model rather than random action.
  - Quick Output to a file containing:
    - Individual Episode #
    - Episode Reward
    - Number of Steps taken.
  - In accordance with our Experiments, the Replay Agent will track the Episode/Steps it was able to traverse past X=1200

PPO is very similar in its Debugging and Visualisation techniques to DDQN, but there are several key distinctions.

- **Training the Agent**
  - Using Stable Baselines, 'Tensorboard' is pre-implemented which is a very useful tool for tracking live data as the Model trains. It also lets us visually explore training data from previous runs and compare it to the current. All Plots are generated dynamically and inside an interface. (Figure 5 – Appendix)
  - This logging is useful to see if the Model is converging on an optimal behaviour. Visual inspection of loss and reward can help pinpoint how the Agent is performing.
  - A VecMonitor Wrapper is used to store Training Rewards to a **monitor.csv**
    - This is useful for SB3's implemented callbacks when determining the average Reward Mean.
- **Replay Agent**
  - Will allow us to specify several Models to evaluate and compare performance based on X-Position.
  - Adjusted Metrics are displayed to better identify the Model and its key parameters.
  - PPO Replay will also output based on the '—render' flag. The Console Output is also identical as DDQN.

Visualisation/Debugging Techniques – OpenCV Agent

The algorithm has multiple visual debugging aids built in to visualise the action-determination process of the agent in real time. The agent can be given a 'detect' debugging flag to present the game in greyscale with rectangles drawn around any detected objects.
The 'console' debugging flag will print to the terminal every time Mario makes a jumping action, including the reasoning behind the decision.



```
Reacting to enemy goomba: (161, 193)
Reacting to pipe: (168, 176)
Reacting to pipe: (160, 160)
Reacting to enemy goomba: (156, 193)
Reacting to pipe: (170, 144)
Hole found in front of mario (114, 194): (106, 208)
Hole found in front of mario (115, 193): (112, 208)
Reacting to enemy goomba: (154, 193)
Reacting to enemy koopa: (183, 185)
Reacting to enemy koopashell: (102, 195)
Reacting to enemy goomba: (166, 193)
Reacting to enemy goomba: (135, 193)
Reacting to block in front of mario: (136, 192)
Reacting to block in front of mario: (124, 160)
Reacting to block in front of mario: (128, 144)
Hole found in front of mario (121, 125): (119, 144)
```

'detect' and 'console' flags for OpenCV Agent

Additionally, the agent can return key metrics from an instance, by specifying the 'metrics=True' flag. When the agent's instance ends, a dictionary is returned containing the reward, run time, number of steps taken, and final global x-position of Mario. This debugging technique assists with the large-scale analytics of the algorithm used to compare it with other algorithms.

There are a few key parameters of the OpenCV algorithm that can be modified to change how and when the agent performs action-determination. The key values are:

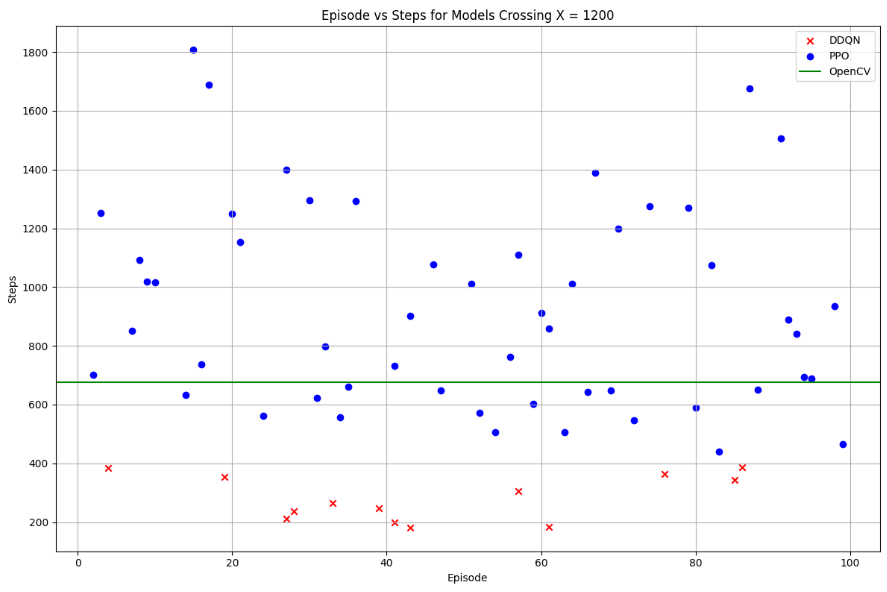
- STEPS_PER_ACTION – Integer value that determines how many steps are taken before another action is chosen. A lower value results in a more reactive agent, but also makes the game run slower. Conversely, a higher value results in a less reactive agent, but the game runs quicker, as object detection occurs at longer intervals.
- GOOMBA_RANGE – Integer value that determines how close (in pixels) Mario should be to a goomba before taking action to jump over/onto it.
- KOOPA_RANGE – Integer value that determines how close (in pixels) Mario should be to a koopa before taking action to jump over/onto it.

To find the best combination of parameters, we ran 847 different simulations, each with a different combination of STEPS_PER_ACTION, GOOMBA_RANGE and KOOPA_RANGE (Figure 6 - Appendix). We found that an OpenCV agent with the parameters of 8 steps per action, 45-pixel range for goombas and 75-pixel range for koopas, performed the best (by beating level 1-1 with the quickest time).

Experiment 1 – Steps Taken to Reach an *x_pos* of 1200 (Level 1-1)

It is difficult to compare the performance of our deterministic OpenCV agent, with no variation between iterations with the same base parameters, and our non-deterministic DQNN/PPO agents, with potential variation between iterations running on a trained model. As such, we used the OpenCV agent to act as a benchmark to compare how quickly a reinforcement learning agent can reach a specific checkpoint in level (1-1).

We chose the global x-position of 1200, as this point is after the first hole in level (1-1). The optimal OpenCV agent took 676 steps to reach the threshold, PPO on average 925, and the DDQN agent consistently outperformed this benchmark with an average of 281 steps:

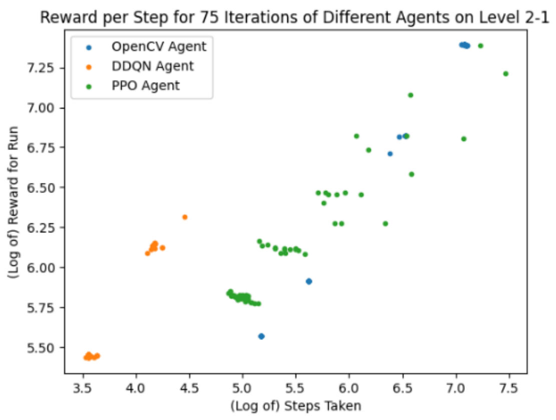


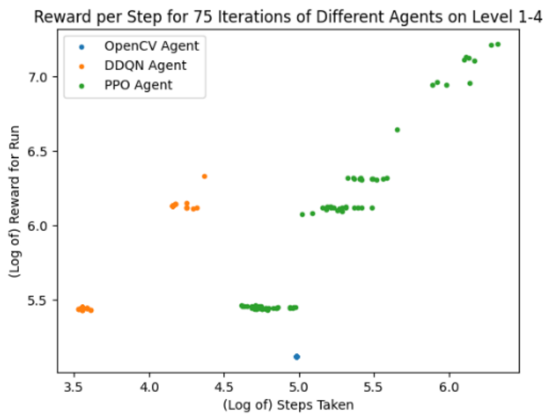
The PPO agent experienced more variation in the number of steps required to reach the threshold but was able to reach the 1200 checkpoint more often than DDQN. On average, the PPO agent took more steps than the OpenCV agent to reach *x=1200*. Despite this, both reinforcement agents showed the ability to be more efficient than a rule-based agent in a trained environment. Whilst the machine learning algorithms can explore and discover quicker paths to traverse the level, the rule-based agent is unable to improve upon its performance without human intervention. We speculate that with further training time, PPO would consistently outperform OpenCV as it settles on an optimal policy.

This experiment highlights a core strength of a rule-based agent - consistency. Despite being trained on thousands of iterations of the first level, the PPO and DDQN agents are still unable to match the consistency of the OpenCV agent, which reaches the goal of 1200 in 676 steps every time (assuming the same pre-set parameters). An agent that can predictably perform well may be more desirable than an agent with no guarantee that it will be able to complete its goal, let alone complete the goal efficiently.

Experiment 2 – Reward/Step on Unseen Level (Level 2-1 & 1-4)

By comparing how a DQNN and PPO agent perform on an unseen level, after being trained to complete level (1-1), with our OpenCV agent employing a general heuristic, we can evaluate whether an algorithm that applies what it has learned from past experiences is stronger in an unseen environment. We chose (2-1) (Figure 7 – Appendix), which shares many of the assets with the first level and (1-4) (Figure 8 – Appendix) which does not share any assets with the first level.





We took the 75 OpenCV agents who were able to complete level (1-1) and evaluated their score. This provided some variation to compare to the non-deterministic agents. To account for the different processing speeds of the agents, as the OpenCV agent performs significantly slower in runtime, we compared the average reward earned per step of 75 iterations of every agent.

The OpenCV agent consistently outperformed the reinforcement-learning algorithms in (2-1) – this is likely due to the pre-defined heuristic recognising the common aspects of the level (the pipes, blocks, enemies etc) meaning it could play relatively well. But it struggled in (1-4) where it did not recognise the environment, it wasn't able to detect what it had not been told to recognise. This highlights the primary weakness of this agent – it will not be able to pass that section of the level until human intervention alters the heuristic.

The DDQN Agent particularly struggled to generate a longer run through the levels, despite being able to produce a proportionately higher reward-to-steps ratio. It did however beat OpenCV in (1-4) which indicates to some degree the strength of the generalisation.

PPO was able to compete more in (2-1) and was able to consistently get quite far in (1-4) – we believe this is due to PPO establishing a Policy for 'holes' that was more useful for adapting to other levels. While not being told to look for a certain obstacle in (1-4) it was able to consistently navigate based on the ability to recognise a favourable jump.

Conclusion

We explored the benefits and drawbacks of DDQN and PPO as reinforcement learning agents, and a rule based OpenCV image processing agent, by comparing their performance when playing an emulation of Super Mario Bros for the NES. Each agent showed promising results, and the following conclusions were drawn:

- DDQN agents occasionally outperformed the PPO and OpenCV agents within the trained environment but was typically unable to progress far through the environment. The training parameters needed to be altered to improve overall performance and ensure extrapolation to unseen levels was more successful. Given the correct training environment, we believe DDQN could be a feasible method of playing Super Mario Bros, given the simple state- and action- spaces that Q-learning thrives upon. However, we were unable to produce a strong model to reflect this belief.
- PPO agents performed well when generalising the models to unseen environments, as they were able to develop strong policies for dealing with specific obstacles (e.g. avoiding holes). PPO is a suitable reinforcement learning technique for training an agent to play Super Mario Bros, however, a large amount of training on multiple different levels would be required before a model is able to play the game effectively.
- Rule-based agents, specifically image processing agents, perform consistently well within the context of the heuristic and can generalise well to unseen environments within the scope of the heuristic. If the environment goes beyond that scope, the agent struggles to perform well at all. An image-processing rule-based agent such as our OpenCV agent is suitable for playing Super Mario Bros, where there is a finite number of obstacles to consider in the heuristic.

If we were to conduct this project again, we would make the following changes:

1. Train the PPO agent with a smaller action space, to increase efficacy and provide more comparable results to the DDQN agent.
2. Expand the OpenCV agents' heuristic to deal with a wider range of environments, to allow for stronger performance beyond Level (1-1).
3. Time constraints restricted the amount of testing with training parameters of the DDQN algorithm. We would've liked to be able to allocate more time to finding efficient training conditions so our agent could meet the expectations that a DDQN can generalise to unseen environments strongly.

Appendix

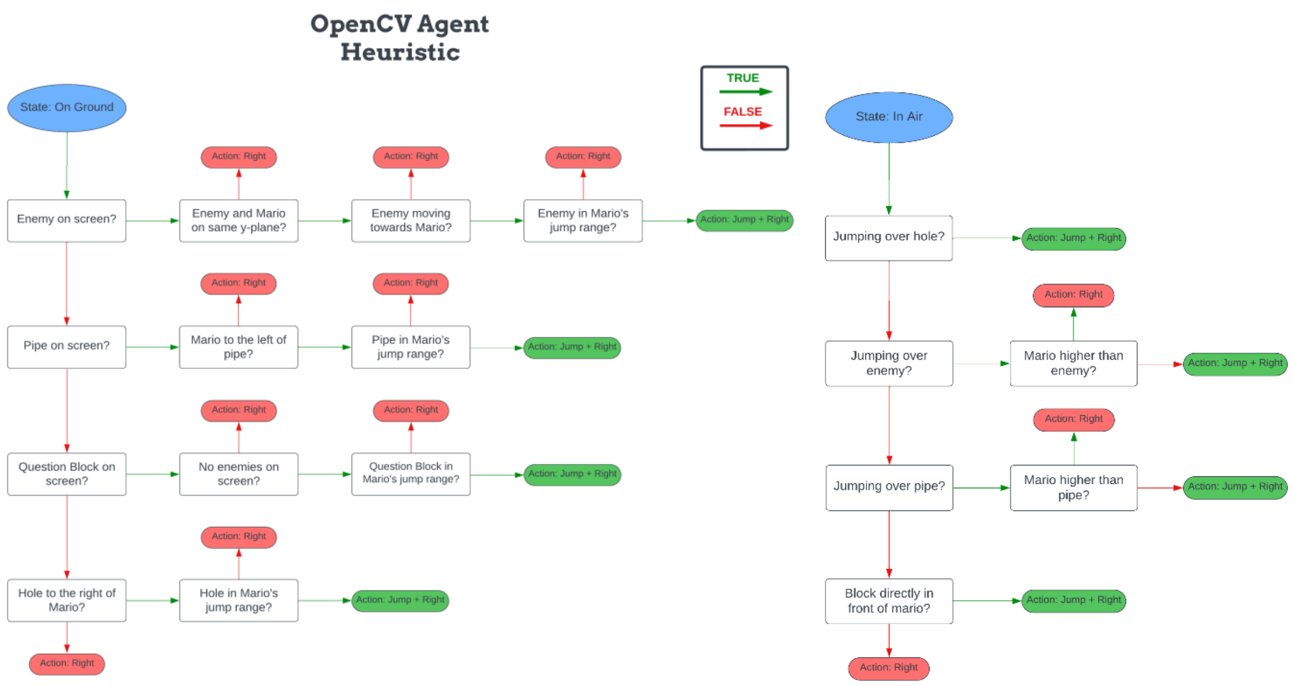
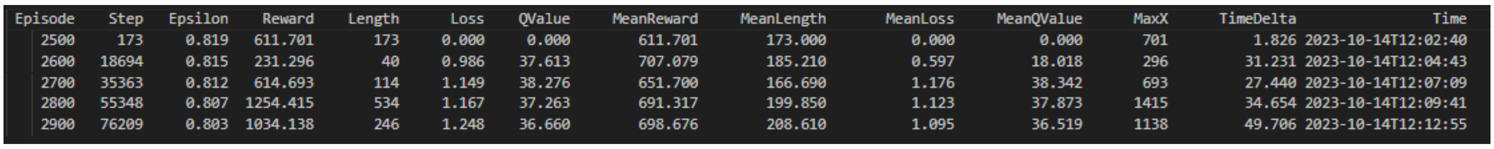

OpenCV Agent Heuristic
TRUE
FALSE
State: On Ground
Enemy on screen?
Enemy and Mario on same y-plane?
Enemy moving towards Mario?
Enemy in Mario's jump range?
Pipe on screen?
Mario to the left of pipe?
Pipe in Mario's jump range?
Question Block on screen?
No enemies on screen?
Question Block in Mario's jump range?
Hole to the right of Mario?
Hole in Mario's jump range?
Action: Right
Action: Jump + Right
State: In Air
Jumping over hole?
Jumping over enemy?
Mario higher than enemy?
Jumping over pipe?
Mario higher than pipe?
Block directly in front of mario?

Figure 1: Action-Determination Heuristic for OpenCV Agent



```
Episode    Step  Epsilon    Reward   Length     Loss   QValue   MeanReward   MeanLength    MeanLoss   MeanQValue    MaxX    TimeDelta               Time
   2500     173    0.819   611.701      173    0.000    0.000      611.701      173.000       0.000        0.000     701        1.826 2023-10-14T12:02:40
   2600   18694    0.815   231.296       40    0.986   37.613      707.079      185.210       0.597       18.018     296       31.231 2023-10-14T12:04:43
   2700   35363    0.812   614.693      114    1.149   38.276      651.700      166.690       1.176       38.342     693       27.440 2023-10-14T12:07:09
   2800   55348    0.807  1254.415      534    1.167   37.263      691.317      199.850       1.123       37.873    1415       34.654 2023-10-14T12:09:41
   2900   76209    0.803  1034.138      246    1.248   36.660      698.676      208.610       1.095       36.519    1138       49.706 2023-10-14T12:12:55
```

Figure 2: DDQN Log File

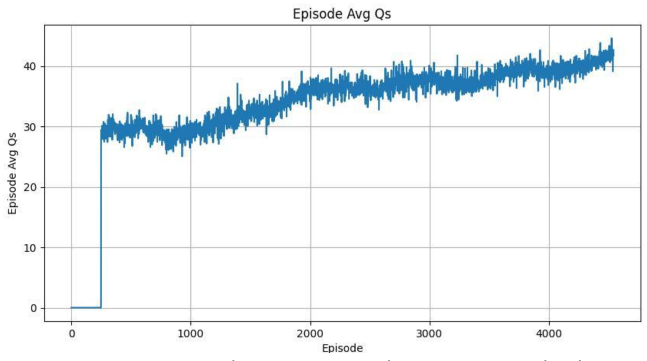


Figure 3: DDQN Episode Average Q-Values – an example plot produced by DDQN Logging.

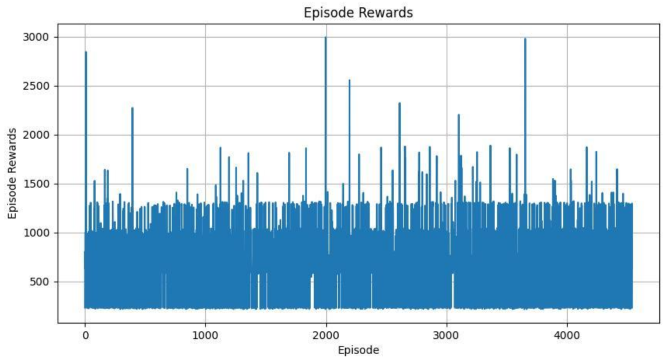
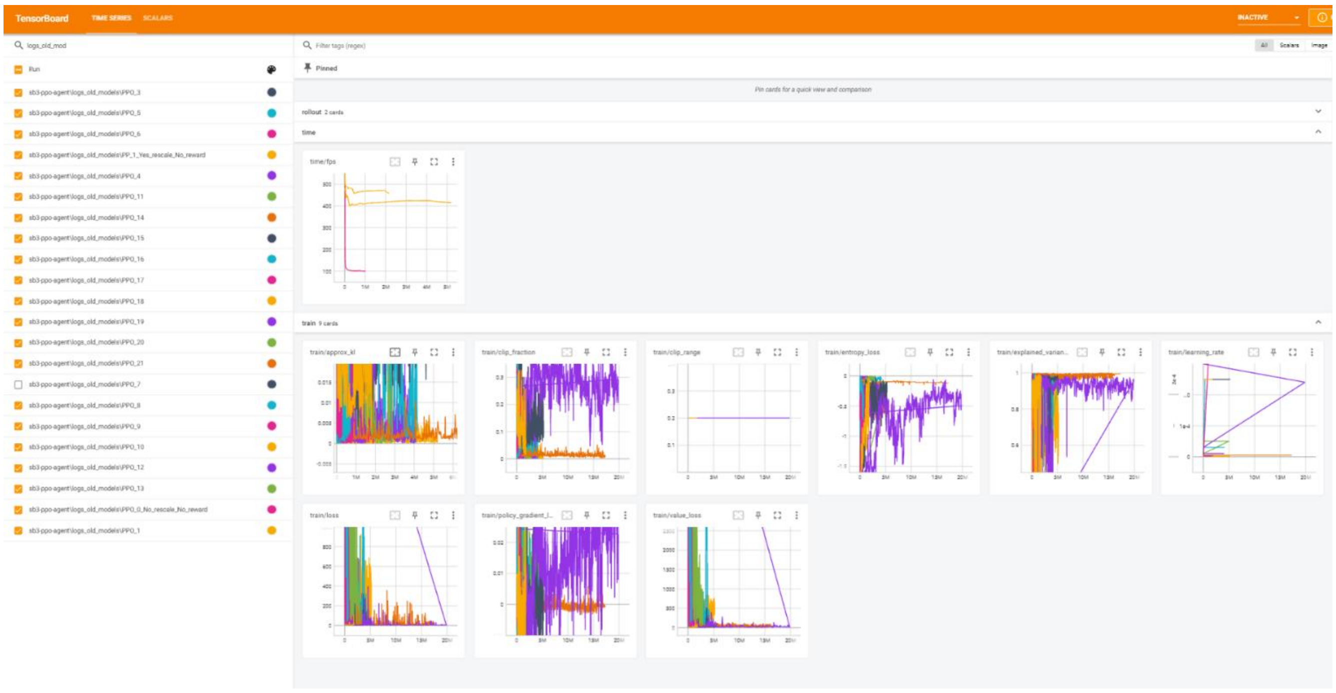


Figure 4: DDQN Episode Rewards– an example plot produced by our Agent as it trains.



Run
- [✓] sb3-ppo-agent\logs_old_models\PPO_3
- [✓] sb3-ppo-agent\logs_old_models\PPO_5
- [✓] sb3-ppo-agent\logs_old_models\PPO_6
- [✓] sb3-ppo-agent\logs_old_models\PP_1_Yes_rescale_No_reward
- [✓] sb3-ppo-agent\logs_old_models\PPO_4
- [✓] sb3-ppo-agent\logs_old_models\PPO_11
- [✓] sb3-ppo-agent\logs_old_models\PPO_14
- [✓] sb3-ppo-agent\logs_old_models\PPO_15
- [✓] sb3-ppo-agent\logs_old_models\PPO_16
- [✓] sb3-ppo-agent\logs_old_models\PPO_17
- [✓] sb3-ppo-agent\logs_old_models\PPO_18
- [✓] sb3-ppo-agent\logs_old_models\PPO_19
- [✓] sb3-ppo-agent\logs_old_models\PPO_20
- [✓] sb3-ppo-agent\logs_old_models\PPO_21
- [ ] sb3-ppo-agent\logs_old_models\PPO_7
- [✓] sb3-ppo-agent\logs_old_models\PPO_8
- [✓] sb3-ppo-agent\logs_old_models\PPO_9
- [✓] sb3-ppo-agent\logs_old_models\PPO_10
- [✓] sb3-ppo-agent\logs_old_models\PPO_12
- [✓] sb3-ppo-agent\logs_old_models\PPO_13
- [✓] sb3-ppo-agent\logs_old_models\PPO_0_No_rescale_No_reward
- [✓] sb3-ppo-agent\logs_old_models\PPO_1

Figure 5: Tensorboard Overview – PPO Training Metrics

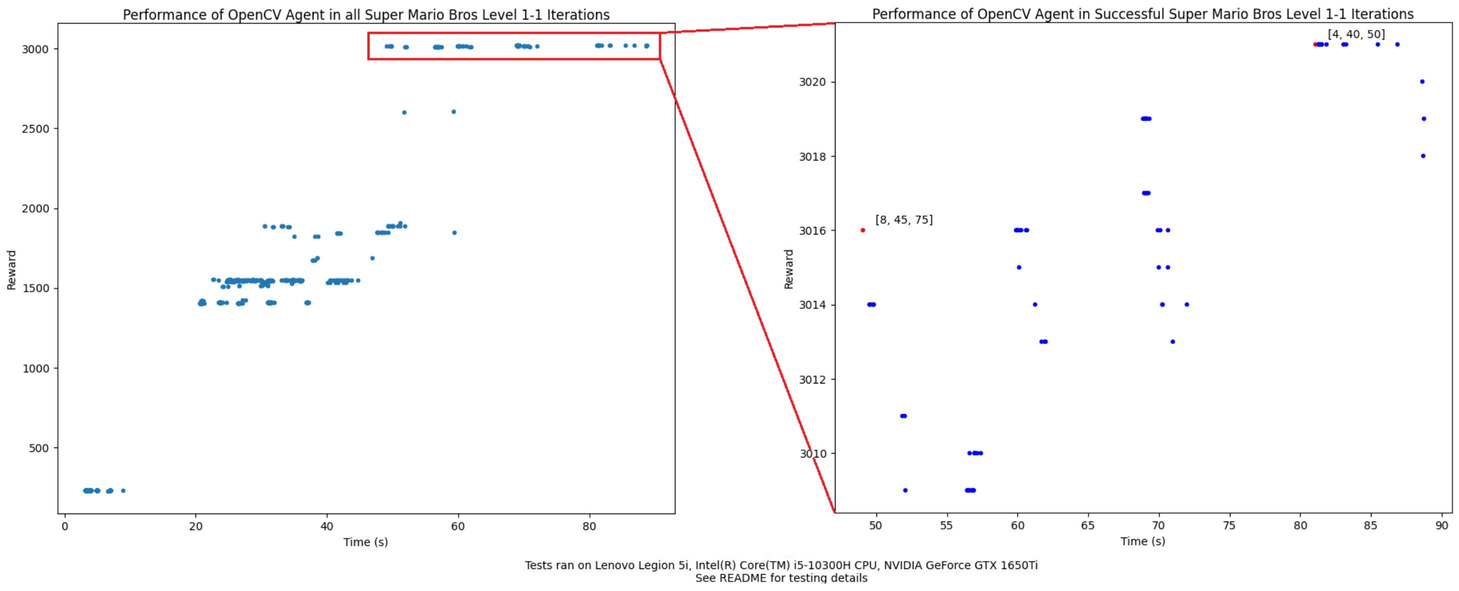


Figure 6: (Left) Performance of 847 Instances of OpenCV Agent with Different Parameter Combinations (Right) Performance of the 75 Successful Instances

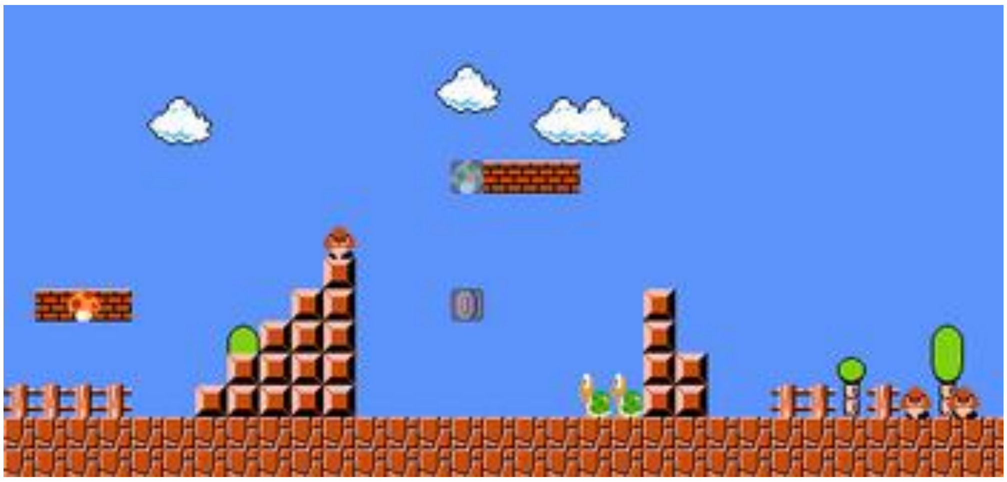


Figure 7: Level 2-1 in Super Mario Bros. Features a Similar Environment to Level 1-1, Which the Agents were Trained on

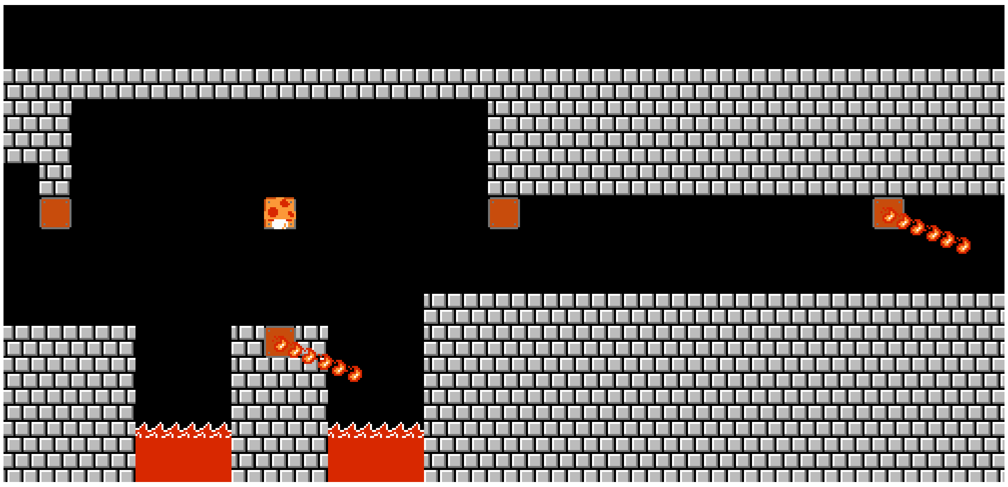


Figure 8: Level 1-4 in Super Mario Bros. Predominantly Features Holes as Obstacles

References

PyTorch: Train a Mario-playing RL Agent. 13/06/2023. Accessed 15/09/2023. https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html

Vmoens. Mario RL Tutorial. 13/06/2023. Accessed 16/09/2023. https://github.com/pytorch/tutorials/blob/main/intermediate_source/mario_rl_tutorial.py

Christian Kauten. Super Mario Bros for OpenAI Gym. GitHub, 2019. 21/06/2022. Accessed 23/08/2023.

https://github.com/Kautenja/gym-super-mario-bros

Yuansong Feng. MadMario agent. 27/10/2020. Accessed 25/09/2023. https://github.com/yfeng997/MadMario/blob/master/agent.py

Andrew Grebenisan. Play Super Mario Bros with a Double Deep Q-Network. 09/03/2020. Accessed 07/10/2023. https://blog.paperspace.com/building-double-deep-q-network-super-mario-bros/

Sebastian Heinz. Using Reinforcement Learning to play Super Mario Bros on NES using TensorFlow. 29/05/2019. Accessed 20/09/2023. https://www.statworx.com/en/content-hub/blog/using-reinforcement-learning-to-play-super-mario-bros-on-nes-using-tensorflow/

Nicknochnack. MarioRL. 22/12/2021. Accessed 21/09/2023. https://github.com/nicknochnack/MarioRL/blob/main/Mario%20Tutorial.ipynb

Stable Baselines Documentation: PPO. 08/05/2023. Accessed 15/09/2023. https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html

Stable Baselines Documentation: Examples. 08/05/2023. Accessed 15/09/2023. https://stable-baselines3.readthedocs.io/en/master/guide/examples.html

OpenAi: Proximal Policy Optimization. 20/07/2017. Accessed 21/09/2023. https://openai.com/research/openai-baselines-ppo

Yuan, Siyu & Zhang, Yong & Qie, Wenbo & Ma, Tengteng & Li, Sisi. (2020). Deep reinforcement learning for resource allocation with network slicing in cognitive radio network. Computer Science and Information Systems. 18. 55-55. 10.2298/CSIS200710055Y.

PyTorchL SmoothL1Loss. Accessed 07/10/2023. https://pytorch.org/docs/stable/generated/torch.nn.SmoothL1Loss.html

PyTorch: Adam. Accessed 07/10/2023. https://pytorch.org/docs/stable/generated/torch.optim.Adam.html

Jason Brownlee. Machine Learning Mastery: Gentle Introduction to the Adam Optimization Algorithm for Deep Learning. 13/01/2021. Accessed 09/10/2023.https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

MarioWiki Fandom: World 1-4 (Super Mario Bros.). Accessed 16/10/2023. https://mario.fandom.com/wiki/World_1-4_(Super_Mario_Bros.)

MarioWiki Fandom: World 2-1 (Super Mario Bros.). Accessed 15/10/2023. https://mario.fandom.com/wiki/World_2-1_(Super_Mario_Bros.)

Open Source Computer Vision Template Matching Documentation. Accessed 28/09/2023. https://docs.opencv.org/4.x/d4/dc6/tutorial_py_template_matching.html.

Vmorarji. Object-Detection-in-Mario. 28/08/2020. Accessed 12/10/2023.https://github.com/vmorarji/Object-Detection-in-Mario/blob/master/detect.py.

Matplotlib Scatter Plot Documentation. Accessed 14/10/2023. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.scatter.html