COMP 579 Final Project: Super Mario RL

Aaron Mills
260849570
aaron.mills2@mail.mcgill.ca

Abstract

This project is an investigation of various reinforcement learning algorithms with the objective of creating a high-performing agent for the classic 2-D platforming game: Super Mario Bros. In the performed experiments, both 'classical' reinforcement learning and deep reinforcement learning algorithms were employed. The process of defining a reduced state space and establishing the reward function for the classical algorithms is documented, and the process of applying various deep reinforcement learning algorithms is discussed. The classical algorithms struggled to learn effectively and failed to overcome challenging sections of levels, including tall obstacles and long gaps. The deep reinforcement learning algorithms, while inconsistent, learned to overcome the obstacles and managed to complete the first level of Super Mario Bros.

1 Introduction

Games are an important part of society, and in recent times the popularity of video games has skyrocketed worldwide. Many modern games have advanced graphics and complex controls. In this paper, reinforcement learning to play the classic Super Mario Bros (SMB), released by Nintendo for the Nintendo Entertainment System (NES) in 1985, is explored (1).

SMB is a 2-D platforming game where the agent (Mario) moves rightward, jumps over enemies and obstacles, and collects coins and power-ups in order to reach a goal as fast as possible with the largest amount of points. The inspiration for this project arose after stumbling upon two papers that tried variants of Q-Learning to solve this problem (2; 3). These papers alluded to the Mario AI Competition from 2009 to 2012, which involved path-finding algorithms, such as A*, used to create an agent to play SMB in a Java environment (4). The objective of this project is to see if reinforcement learning techniques can compete with the successful path-finding algorithms from this competition held over a decade ago. The primary focus of the project is to create an agent proficient at level completion rather than attaining the highest score.

Algorithms such as Q-Learning, SARSA, Double Q-Learning, and Expected SARSA were first attempted but obtained poor results. Deep reinforcement learning techniques were then explored, including Deep Q-Learning (DQN), Double Deep Q-Learning (Double DQN), Dueling Deep Q-Learning (Dueling DQN), and Proximal Policy Optimization (PPO). After implementing the deep RL algorithms using Deep Learning for Java (DL4J) for operation in the Java environment, it was clear that training these agents in Java would be infeasible. To make training times reasonable, the project pivoted to an OpenAI gym Python environment, and all four of the deep RL agents were re-implemented. Due to time constraints, only the PPO and Dueling DQN agents were extensively trained.

2 Background

Dueling DQN: The Dueling DQN algorithm is very similar to the DQN algorithm learned in the course lectures. Instead of modeling the values of the state-action space, the value of the state and the advantages for each action in that state are modeled separately. They use the same convolutional network but have separate linear networks. From the state values and advantages, the state-action values are computed and then returned (9):

$$Q(s,a;\theta) = V(s;\theta) + (A(s,a;\theta) - \frac{1}{|\mathcal{A}|}\sum_{a' \in \mathcal{A}} A(s,a';\theta)) \tag{1}$$

PPO-Clip: There are two main variants of the PPO algorithm: the Adaptive Kullback-Leibler (KL) Penalty and the Clipped approaches. The PPO-Clip method is a modification of the Trust Region Policy Optimization algorithm, both of which are developed by OpenAI, where the objective that is maximized is changed to prevent large policy updates. The objective is expressed as follows (11):

$$L(\theta) = \mathbb{E}_t\left[min\{\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}A_t, clip(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}, 1-\epsilon, 1+\epsilon)A_t\}\right] \tag{2}$$

In the equation above, A_t is the advantage. The $clip$ function forces the policy ratio to be clamped between values $1-\epsilon$ and $1+\epsilon$ where $\epsilon > 0$ is a chosen parameter. For this project, $\epsilon = 0.2$ was chosen. The objective function takes the minimum between the policy ratio multiplied by the advantage and the clipped policy ratio multiplied by the advantage. This clipping keeps updates to the policy small.

3 Methodology and Results

3.1 'Classical' Reinforcement Learning

The Mario AI Competition provides a Java environment in which agents can interact by providing information about which buttons to press for each frame (5). There are five buttons: LEFT, RIGHT, DOWN, JUMP, and SPEED. Given that certain combinations of inputs result in redundant actions (e.g., both LEFT and RIGHT selected simultaneously), excluding those results in an action space of size 14.

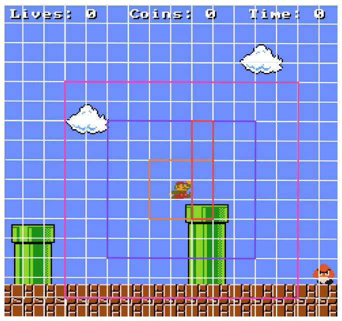


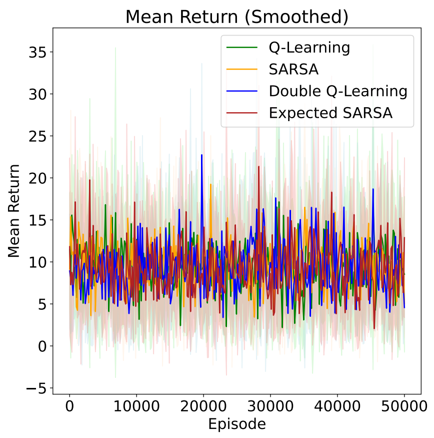
Figure 1: Mario jumping over a pipe in the original Super Mario Bros game with a representation of the grid provided by the Java environment. Shown in orange, purple, and pink are the areas around Mario considered near, mid-range, and far respectively. The red boundary is the area in which the tiles are checked for obstacles.

The environment provides information about the system's current state, including position, velocity, and how Mario is interacting with the environment. Most information comes from two 16×16 integer grids, which indicate where enemies and objects are located. This yields a massive, infeasible state space. Therefore, some subset of important information needs to be extracted. 41 bits were extracted, leaving 2^{41} different possible state combinations. This is still incredibly large, but only a small fraction of these states can possibly occur in an episode of SMB. Therefore, only state-action values of visited states are maintained in memory, greatly reducing the number of stored entries.

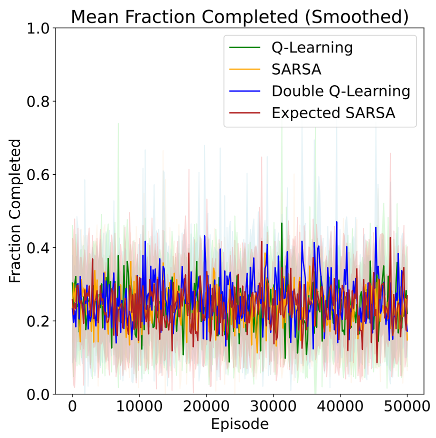
Originally, the reward scheme was very simple. If Mario had made progress and gone rightward, then he is rewarded. If he goes left, then he is penalized. This makes sense since the goal state is always to the right. This got Mario moving rightward and eliminating enemies, but he struggled to jump over tall obstacles, particularly the green pipes with a height of 4 tiles. To solve this problem, a reward was added for Mario leaping from shorter platforms and landing on taller ones. It was only awarded if Mario progressed further rightward to prevent the agent from continuously jumping onto the same platform.

With this setup, four classical reinforcement learning algorithms were investigated: Q-Learning, SARSA, Double Q-Learning, and Expected SARSA. Unfortunately, these algorithms struggled to learn past the halfway point of the first stage. In particular, they could not handle gaps where failure to jump over them results in losing. After trying many different adjustments to the state, reward, and learning parameters, it was time to try more powerful methods using deep reinforcement learning.

3.2 'Classical' RL Results



(a) Return obtained during episodes averaged over 5 runs. The standard deviation is plotted in a lighter colour. The data is smoothed for clarity.



(b) The fraction of the level completed averaged over 5 runs. The standard deviation is plotted in a lighter colour. The data is smoothed for clarity.

Figure 2: Mean return and mean fraction of the level completed by the four 'classical' RL agents: Q-Learning, SARSA, Double Q-Learning, and Expected SARSA.

Figure 2 demonstrates the performance of Q-Learning, SARSA, Double Q-Learning, and Expected SARSA in the Java SMB environment. 5 runs of 50000 episodes were performed, with reward and progress information extracted from the environment and stored. Hyperparameters were initially chosen based on other papers with a similar setup and tweaked based on empirical performance (2; 3). Values of $\gamma = 0.6$ and $\alpha = 0.3$ were selected. ϵ was chosen to be a value that decreased over the course of a run based on the expression $\epsilon = 0.3(1 - i/n)$, where i is the current episode number, and n is the total number of episodes per run. Figure 2 shows the data obtained from these algorithms. The algorithms all performed very similarly, with very little improvement over the course of 50000 episodes.

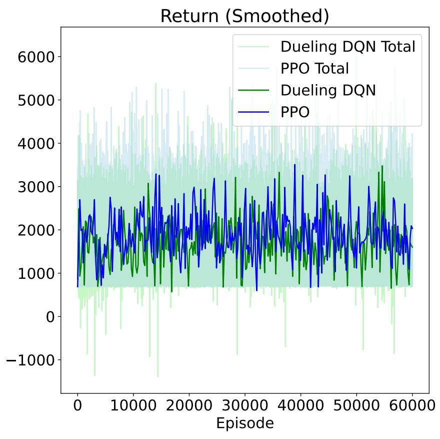
3.3 Deep Reinforcement Learning

After obtaining poor results with classical RL techniques, using function approximators was the next natural course of action. Originally, the plan was to use the same Java environment provided for the Mario AI Competition. DQN, DDQN, and PPO were implemented in Java with pixel images extracted from the environment, then processed with resizing and grayscale for input. However, after attempting to train these algorithms, it quickly became clear that they were far too slow to perform substantial training. To be able to train deep RL algorithms, the environment and programming language needed to change.

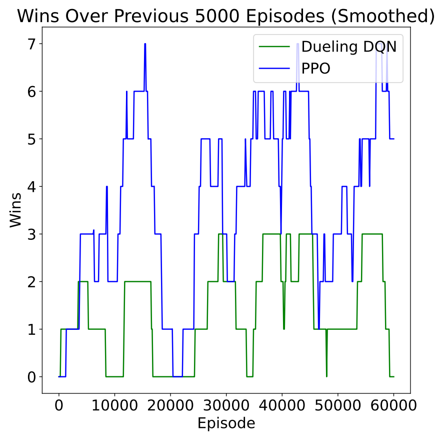
The project shifted to an Open AI gymnasium environment for SMB in Python, where an RGB image of size 256×240 is provided for the state and a simple reward scheme that rewards rightward movement and penalizes leftward movement. This is perfectly fine for the purposes of implementing deep RL since the algorithms investigated all use convolutional neural networks. To improve efficiency and performance, the frames are processed prior to input to the models. First, the frames are resized to size 84×84, and made grayscale. Instead of each frame being passed to the agent, successive frames are collected into batches of size 4 and then forwarded to the convolutional neural networks.

The algorithms developed in the Python environment include DQN, DDQN, PPO, and one new one: Dueling DQN. Due to time constraints, only the Dueling DQN and PPO algorithms were trained. This decision was made after training all of the algorithms for 3000 episodes and selecting the two best performers out of the four candidates. For the three DQN algorithms, a replay buffer maintains state, action, reward, and next-state pairs. Similarly, for the PPO algorithm, a rollout buffer saves trajectories, state values, and log probabilities of actions in those states. It also computes the return and advantages over the saved trajectories. Periodically a batch of data is sampled from these memory structures for training.

3.4 Deep RL Results



(a) Return obtained during the episodes. The data is smoothed for clarity.



(b) The number of wins achieved by the agent over the last 5000 episodes.

Figure 3: The return and the number of wins obtained by the Dueling DQN and PPO deep RL algorithms.

Figure 3 demonstrates the total reward (return) obtained by the agents for each episode trained and the number of wins achieved over the previous 5000 episodes. The hyperparameters include: $\gamma = 0.99$, $\alpha = 0.0001$ for both agents, and $\epsilon = 0.3$ for the Dueling DQN agent. The agents were trained in batches of 5000 episodes for practical reasons. Every 5000 episodes the models, their optimizers, and all the data collected were saved. In this environment, Mario has 3 lives per episode, and checkpoints in the level are enabled. This may have contributed to the uninteresting curves in Figure 3 even though the progression of the agent is clear when observing it run. Mario is able to obtain more total reward in an episode where he fails 3 times before the checkpoint rather than passing the checkpoint on the first life and dying two times quickly after respawning.

Throughout this entire section of the project, Mario's performance slowly improved (he would reach further locations in the level more often), but at a decreasing rate. In the first level, there are 4 tough obstacles in addition to the enemies that need to be overcome: the tall pipes, a small gap, a large gap, and a gap between two sets of stairs. After training, Mario could consistently cross the tall pipes and small gap. He was reasonably successful at overcoming the large gap and occasionally could clear the stair gap. Since Mario needs to successfully progress through the earlier parts of the stage in order to learn a later part of the stage, learning later obstacles takes considerably longer.

4 Conclusion

Overall, while the results of this project in terms of data are not as exciting as expected, a lot was learned during this project, and the results revealed there is room for improvement. With the Java environment, the state and reward functions were not given and needed to be defined, and deep RL algorithms not learned in the lectures, including Dueling DQN and PPO-Clip were thoroughly explored. The biggest issue was that later obstacles in the level received too little exposure during training, with the memory buffers being mostly full of the earlier parts of the level. One potential approach to overcome this issue is to split the level into smaller chunks, train the model on these chunks separately, and then train the model on the entirety of the level, effectively 'stitching' the chunks together. Since the ability of the agents to learn new obstacles decreased as they progressed further, perhaps with enough training, the implemented algorithms would be able to beat the first level of SMB consistently.

References

[1] “The official home of Super Mario™ – history,” The official home of Super Mario™ – History. [Online]. Available: https://mario.nintendo.com/history/.

[2] J. -J. Tsay, C. -C. Chen and J. -J. Hsu, ”Evolving Intelligent Mario Controller by Reinforcement Learning,” 2011 International Conference on Technologies and Applications of Artificial Intelligence, Chung Li, Taiwan, 2011, pp. 266-272, doi: 10.1109/TAAI.2011.54.

[3] Y. Liao, K. Yi, and Z. Yang, “CS229 final report reinforcement learning to play mario.” [Online]. Available: http://cs229.stanford.edu/proj2012/LiaoYiYang-RLtoPlayMario.pdf.

[4] J. Togelius, S. Karakovskiy and R. Baumgarten, ”The 2009 Mario AI Competition,” IEEE Congress on Evolutionary Computation, Barcelona, Spain, 2010, pp. 1-8, doi: 10.1109/CEC.2010.5586133.

[5] A. Khalifa, “AMIDOS2006/mario-ai-framework: 10th anniversary edition (not endorsed by Nintendo),” GitHub. [Online]. Available: https://github.com/amidos2006/Mario-AI-Framework.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing Atari with deep reinforcement learning - department of computer ...” [Online]. Available: https://www.cs.toronto.edu/ vmnih/docs/dqn.pdf.

[7] K. Doshi, “Reinforcement learning Deep Q Networks, step-by-step,” Medium, 14-Feb-2021. [Online]. Available: https://towardsdatascience.com/reinforcement-learning-explained-visually-part-5-deep-q-networks-step-by-step-5a5317197f4b.

[8] H. van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double Q-learning,” arXiv.org, 08-Dec-2015. [Online]. Available: https://arxiv.org/abs/1509.06461.

[9] Z. Wang , T. Schaul , M. Hessel , H. van Hasselt, M. Lanctot , and N. de Freitas, “Dueling Network Architectures for Deep Reinforcement Learning.” [Online]. Available: https://arxiv.org/pdf/1511.06581.pdf.

[10] D. Rodrigo, “A graphic guide to implementing PPO for Atari Games,” Medium, 11-Feb-2022. [Online]. Available: https://towardsdatascience.com/a-graphic-guide-to-implementing-ppo-for-atari-games-5740ccbe3fbc.

[11] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” arXiv.org, 28-Aug-2017. [Online]. Available: https://arxiv.org/abs/1707.06347.

5 Appendix

State Formuation

Mode: Mario himself can be in three different modes, Small Mario, Large Mario, or Fire Mario. This can be described in just 2 bits.

Velocity: There are 8 directions in which Mario can be moving, and the alternative case in which Mario is stationary. This leaves us with 9 different cases that can be described in 4 bits.

Stuck: Since there are multiple places where Mario can get stuck in a stage, there is an indicator to encapsulate the idea that Mario has not moved in a certain number of frames. This is described in just 1 bit.

Ground: Since Mario has different abilities if he is on the ground rather than in the air, the state indicates whether Mario is on the ground. This is described in 1 bit.

Jump: There are situations where Mario can and cannot jump. A single bit indicates whether Mario is capable of jumping.

Collision: If Mario has collided with an enemy in this frame, this indicator bit is set to 1.

Nearby Enemies: If there are nearby enemies in any of the 8 directions surrounding Mario, then a bit corresponding to a particular direction is set to 1. An enemy being 'nearby' is defined as an enemy within the 3×3 grid (or 4×3 is Mario is Large or Fire) surrounding Mario.

Midrange Enemies: Similarly, if there are midrange enemies in any of the 8 directions surrounding Mario, then a bit corresponding to a particular direction is set to 1. An enemy being 'midrange' is defined as an enemy within the 7×7 grid (or 8×7 is Mario is Large or Fire) surrounding Mario but outside of the 'nearby' grid.

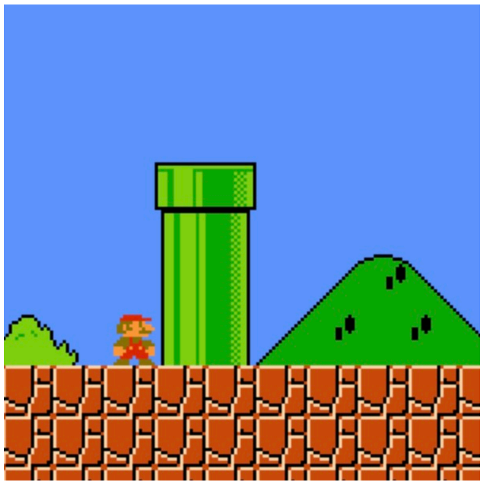
Far Enemies: Similarly, if there are far enemies in any of the 8 directions surrounding Mario, then a bit corresponding to a particular direction is set to 1. An enemy being 'far' is defined as an enemy within the 11×11 grid (or 12×11 is Mario is Large or Fire) surrounding Mario but outside of the 'midrange' and 'nearby' grids.

By Stomp: If Mario eliminated an enemy by stomping on top of it, this indicator bit is set to 1.

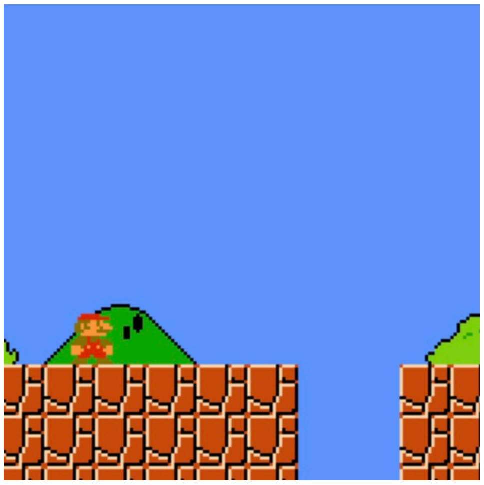
By Fire: If Mario eliminated an enemy by shooting fire at it, this indicator bit is set to 1.

Obstacles: 5 bits are dedicated to 5 grid locations in front of Mario as indicated by the red rectangle in Figure 1. These indicate whether an obstacle is in front and/or a pit may be below.

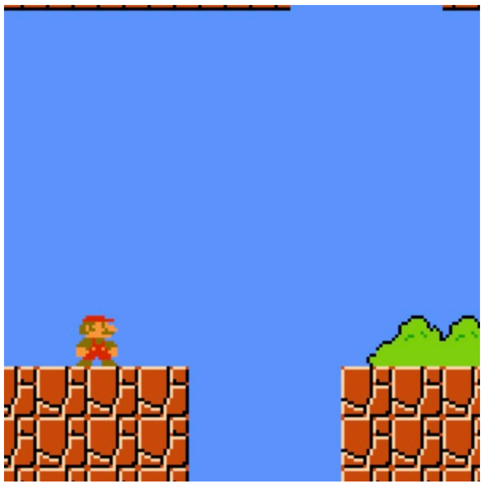
Major Obstacles of Level 1-1 Throughout the first level, there are many obstacles that Mario needs to learn how to pass in order to reach the goal. These include enemies, barriers, platforms, and gaps that must be overcome by jumping at the appropriate time. Below are the four most difficult obstacles that Mario needs to learn to clear.



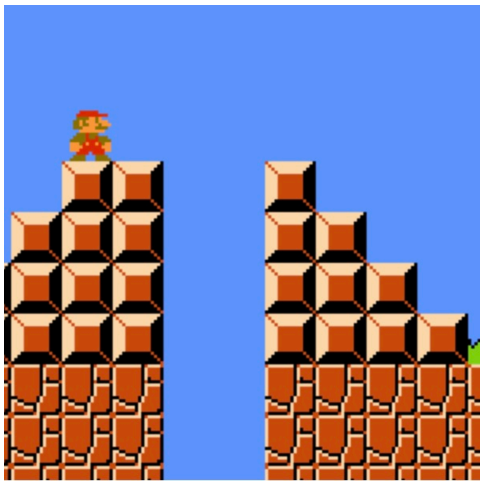
(a) Tall pipe obstacle. This is the first major hurdle that Mario needs to learn to overcome. It takes many successive ‘jump’ actions to reach the height in which the pipe can be passed.



(b) Small gap obstacle. This is the second major hurdle that Mario needs to learn to overcome. Unlike the tall pipe, there are dire consequences for failing to make this jump successfully.



(c) Large gap obstacle. This is the third major hurdle that Mario needs to learn to overcome. The same consequences as the small gap apply, but this time Mario needs to make many successive ‘jump’ actions to clear the gap. Mario learns to sprint beforehand to more easily pass this gap.



(d) Stair gap obstacle. This is the fourth major hurdle that Mario needs to learn to overcome. While seemingly not much harder than the small gap obstacle, Mario does not have the clearance to sprint before reaching the ledge giving him less horizontal velocity in his jump.

Figure 4: The four major obstacles of SMB world 1 level 1.