Creating a Q-learning Agent to Play Super Mario Bros.

Oma Hameed and **Katie Kowalyshyn** and **Ilina Navani**
{omhameed,ilnavani,kakowalyshyn}@davidson.edu
Davidson College
Davidson, NC 28035
U.S.A.

Abstract

Our goal was to create a Q-learning agent to play Level 1 of Super Mario Bros. using the open-source environment provided by the 2010 Mario AI competition. Between 2009 and 2012, there were four Mario AI competitions, and while each year's competition differed slightly, the 2010 version was the first to include a specific track for learning agents. After implementing an agent that iteratively learns a policy to make decisions in the game environment, we found that our agent performs better than most randomized and rule-based agents, yet does not successfully complete the entire level.

1 Introduction

In 2009, Sergey Karakovskiy and Julian Togelius created the first Mario AI Competition, where participants submitted AI agents to play the video game Super Mario Bros. to the best of their ability. Following its first year, the competition was run thrice, though modified to accommodate additional submissions. The second version of the competition in 2010 consisted of four tracks, including the Gameplay, Learning, Level Generation, and Turing Test tracks. By separating the competition into four tracks, the creators were able to offer specialized environment support for each track and allow agents with similar approaches to compete against one another. Their paper, "The Mario AI Championship 2009-2012" (Togelius et al. 2013) provides a detailed overview of the focus of each of the different tracks. By introducing a Learning track designed to reward controllers that learn to play a single level as well as possible, the 2010 competition facilitated the implementation of agents using reinforcement learning (RL) algorithms. Therefore, our goal was to implement an AI agent using the Q-Learning algorithm that can successfully play a level of Super Mario Bros. in the Learning track of the 2010 Mario AI competition.

RL is a widely studied method in AI that provides a powerful framework to train intelligent agents through interactions with their environment. Specifically, the Q-learning algorithm is a model-free reinforcement learning technique involving the dynamic exploration of state-action pairs to optimize an agent's decision-making. In the context of Super Mario Bros., Q-learning enables an agent to autonomously learn effective strategies by iteratively updating its knowledge of which actions yield successful outcomes. In understanding the process of implementing an agent for Super Mario Bros., we studied agents that had been entered into the competition and analyzed their results through the paper "Mario AI Benchmark and Competitions," written by the creators after the 2010 competition (Karakovskiy and Togelius 2012). We were, therefore, able to gain a detailed understanding of their benchmark and the competition framework and learn how to build an agent capable of navigating the specified environment. The remainder of this paper will outline the game of Super Mario Bros and its representation in the Mario AI competition, as well as the implementation of our RL agent and its performance in the game.

2 Background

Super Mario Bros. Overview

The video game Super Mario Bros. is centered around moving the player-controlled character, Mario, through two-dimensional levels, which are viewed sideways. The target of the game is to reach the goal of each level, which is located on the rightmost side of the levels. Along the way, a higher score can be achieved by collecting items, killing enemies, and clearing the level as fast as possible (Togelius et al. 2013). Below is a figure which shows the interface for Super Mario Bros.

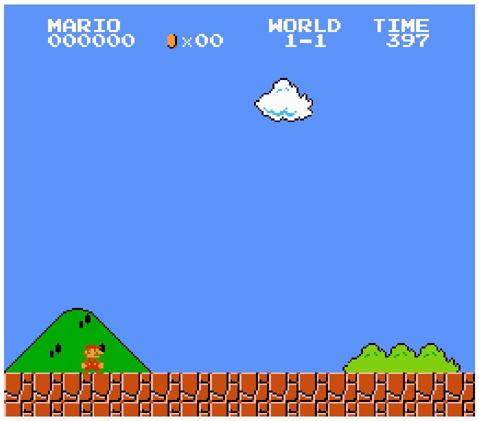

Figure 1: Super Mario Bros. 2009 Graphical Interface (Kauten 2023)

When playing a level, Mario can walk or run to the right and left, jump, and shoot fire, all while being in one of 3 modes: Small, Big, or Fire (Tsay, Chen, and Hsu 2011). In Fire mode, he can shoot fireballs to kill enemies. Furthermore, all of Mario's modes can kill enemies by stomping on their heads or using a shell to collide with them. Mario produces a shell by stomping on the turtle-like enemy, however, if he touches enemies instead of stomping on them, he will get hurt. Getting hurt by an enemy either changes Mario to a less powerful mode or kills him if in Small mode. Fortunately, Mario can restore his health by collecting power-up items such as Mushrooms and Fire Flowers along the way. Other objects in Mario's surroundings are also important, specifically, the four types of bricks. Mario can stand on the bricks, and if Mario is not in fire mode, he can gain either power-up items or coins by breaking bricks. Bad landforms and obstacles such as hills, gaps, cannons, and pipes easily injure Mario or die immediately and lose the game. The game ends when Mario loses a life by falling into a gap or getting hurt when in Small mode, or if he has not reached the goal when the level's time limit of 200 seconds runs out (Tsay, Chen, and Hsu 2011). The reader is referred to the Nintendo website for a more in-depth description of the game (Nintendo 2020).

Competition Benchmark

The 2010 competition provided benchmark software to all competitors (Karakovskiy and Togelius 2012). Hence, we built our agent on its benchmark and estimated its performance by using the given evaluation system. Each game in the benchmark is run in 24 frames per second, making each time step correspond to 42 milliseconds, during which the agent receives a description of the environment and outputs an action. The environment is represented as a two-dimensional array that describes the world around Mario with block resolution and with Mario himself in the center (see Figure 2 below). Specifically, the array contains the positions and types of enemies, items, or platforms around Mario at a given time step. Additional information about what state Mario is in (Small, Big, or Fire), or whether he is currently on the ground or can jump, is provided via separate binary/discrete variables.

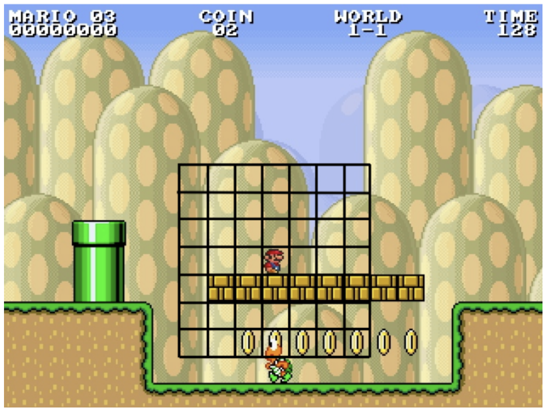

Figure 2: Environment Representation around Mario (Karakovskiy and Togelius 2012)

Mario AI Agent

Our agent employs Q-learning to identify the best moves for winning Level 1 of Super Mario Bros. Q-learning performs value iteration to find the optimal policy which is the strategy that, for each state, prescribes the action that maximizes the expected cumulative reward. This is achieved by maintaining a value of current and future rewards for each (state, action) pair in a table. Our implementation of states, actions, and rewards was guided by Tsay's, Chen's, and Hsu's paper "Evolving Intelligent Mario Controller by Reinforcement Learning" (Tsay, Chen, and Hsu 2011), and our design choices for each are broken down below.

States. The different environments that Mario encounters serve as the states, which consist of the information provided by the benchmark to the agent at every time step. In our implementation, each of the states is composed of attributes in the environment listed below and is ultimately represented as a unique number in the Q-table.

- Mario Mode: Size of Mario where 0 represents Small, 1 represents Big, 2 represents Fire
- If Mario is on the ground, or not: 0 or 1
- If Mario can jump, or not: 0 or 1
- If Mario is stuck, or not: 0 or 1
- If Mario collided with an enemy, or not: 0 or 1
- If Mario killed an enemy, or not: 0 or 1
- Enemy information: If there are enemies in front of, to the back of, above, or below Mario, or not (True/False for each)
- Obstacle information: If there is an obstacle, such as a pipe or block, in front of Mario, or not (True/False)

Since Super Mario Bros. has a complex environment with multiple enemies, items, landforms, and obstacles occurring in many different positions, the state space is too large to initialize to a Q-table. This poses a challenge when playing within a framework that requires real-time responses. To reduce the number of states to a practical level that makes RL feasible, we used a hash table to store $(state, action)$ pairs. Since not all states will likely be visited during the learning process of Level 1, using a hash table allows us to dynamically store and track only the states that have been encountered by the agent.

Actions. In every state, Mario can perform one of 12 actions, determined by a set of keys that can be employed individually or in combination with one another. The 12 actions are: {stay, left, right, jump, fire, right+jump, left+jump, right+fire, left+fire, jump+fire, left+jump+fire, right+jump+fire}.

Rewards. Each state has a reward associated with it, either positive or negative, depending on its impact on Mario's survival. States that enable moving forward, jumping onto platforms, and killing/avoiding enemies are positively rewarded in increasing order. On the other hand, states that result in Mario being stuck or moving backward are negatively rewarded, and states that result in a collision with obstacles or enemies are given especially high negative rewards.

Q-learning. When playing Super Mario Bros, our agent decides on its next move by using the policy learned during training. In the learning process, when the agent changes states and receives a reward, the Q-learning algorithm updates the Q-value using the formula:

$$Q(s,a) = Q(s,a) + \alpha \cdot \left(R + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)\right)$$

where $Q(s,a)$ is the current state's Q-value, $Q(s',a')$ is the future state's Q-value, α is the learning rate, R is the reward, and γ is the discount factor.

The learning rate determines how new information affects accumulated information from previous instances. For instance, if α is 1, then the new information completely overrides any previous information. Additionally, the discount factor determines the importance of future Q-values. Specifically, if γ is 0, then the agent will only consider the immediate rewards of the current state, whereas if γ is near 1, the agent will try to maximize the long-term reward even if it is many moves away.

Finally, when taking an action during learning, we implemented an exploration-exploitation strategy, to balance exploring new actions and exploiting known actions based on Q-values. Exploration selects an action at random, therefore allowing the agent to discover potentially better actions and learn more about the environment. Exploitation, on the other hand, leverages the agent's current knowledge to choose actions with high Q-values. To ensure effective learning, we use a mostly greedy exploitation strategy with only occasional exploration.

3 Experiments

Learning and Evaluation

As per the Learning Track of the 2010 competition, our agent played a level of Super Mario Bros 10,000 times intending to converge toward an optimal policy. Following learning, its performance was evaluated on the same level by using the scoring method of the competition, where the score is calculated by a weighted combination of the total distance traveled on the level, total number of kills, and time spent. For more specific information about scoring techniques, readers are referred to Togelious and Karakovskiy's paper from 2012 (Karakovskiy and Togelius 2012). We ultimately ran 10 independent rounds of evaluation after learning and took the average among their scores to determine our agent's final score.

As discussed earlier, hyperparameters such as exploration chance, learning rate, and discount factor are known to impact the efficiency of Q-learning significantly. Hence, we repeated the process of learning and evaluation with different combinations of hyperparameters, particularly comparing low (0.15), medium (0.4), and high (0.7) learning rates and discount factors. Exploration chance, that is the probability of choosing a random action, was set to 0.15 during learning to allow for occasional exploration while encouraging the agent to choose actions with high Q-values a majority of the time.

Performance Against Other Agents

After evaluating our agent's performance using the above methods, we compared its average scores to that of other agents playing at the same level. Specifically, we tested against 4 agents that were designed by the competition creators. Each of the agents is briefly described below.

Random Agent. This agent generates random boolean values for each possible action, with a bias towards moving right and making long jumps, resulting in a stochastic and unpredictable approach to the game.

Scared Agent. This agent primarily focuses on initiating jumps when encountering potential obstacles. The agent resets its state by defaulting to rightward movement without using speed, contributing to a strategy that aims to navigate the game with a cautious approach.

Speedy Scared Agent. This agent prioritizes both speed and caution when playing a level. By utilizing a combination of jumping and speed actions, it moves right while attempting to maintain an optimal speed, thus strategically balancing agility and risk aversion.

Forward Agent. This agent is designed to move forward while dynamically adapting to the environment. While prioritizing continuous rightward movement, the agent assesses potential dangers, such as gaps and enemies, and adjusts its actions accordingly. Hence, its approach is more focused on navigating obstacles and maintaining forward momentum.

4 Results

Our evaluation results, after performing Q-learning with different combinations of hyperparameters, are shown in Table 1 below. We see that the highest score on the level is achieved with a learning rate of 0.4 and a discount factor of 0.7. An optimal learning rate of 0.4 suggests that the agent is making substantial updates to its Q-values based on new experiences, which is suitable for an environment like Super Mario Bros where it needs to adapt quickly to changing conditions. Further, an optimal discount factor of 0.7 suggests that the Q-learning algorithm aims to maximize the long-term reward, allowing the agent to plan and consider the cumulative impact of its actions.

Discount Factor	**Learning Rate**		
	0.15	0.4	0.7
0.15	2153	2720	2720
0.4	2449	2736	2651
0.7	2557	**3110**	2430

Table 1: Average Scores while Tuning Hyperparameters

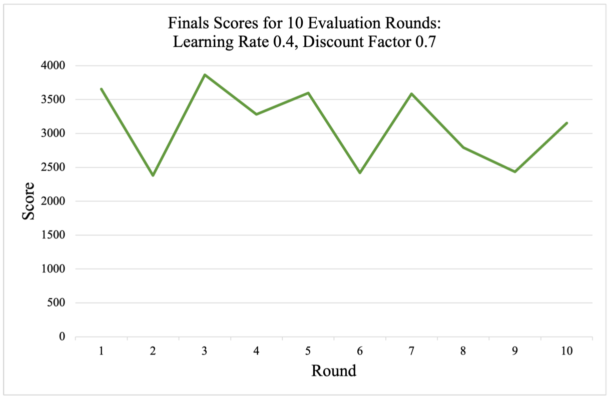

Figure 3: Finals Scores over 10 Evaluation Rounds (Learning Rate 0.4, Discount Factor 0.7)

Overall, we conclude that the average highest score achieved by our agent after using Q-learning is 3110. The scores obtained on each of the 10 rounds of evaluation are shown in Figure 3. As seen in the figure, scores were consistently within the range of approximately 2500 to 3800. Hence, although our agent was not able to successfully complete Level 1, it made significant progress through the level to obtain decent scores. To evaluate performance and further understand the significance of our scores, we compared our results to those of randomized and rule-based agents provided by the competition creators.

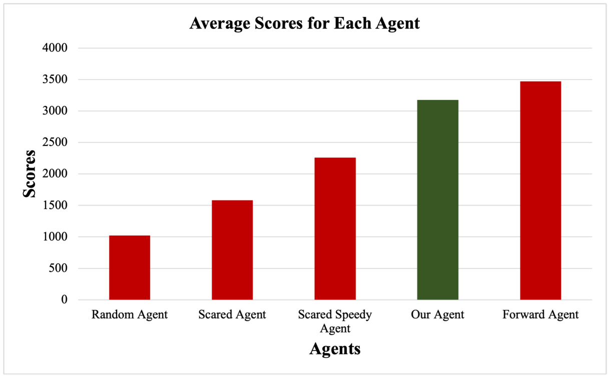

Figure 4: Average Scores for Each Agent

Figure 4 compares the average score of our agent and four other agents after 10 rounds of evaluation on the same level. We see that our agent outperforms three of the four agents. As expected, the Random agent has the lowest scores, followed by the Scared agent and Speedy Scared agent, both of which prioritize jumping throughout a level and are therefore susceptible to gaps in the environment. The Forward agent achieves an overall better score than our agent, and a reason for this could be its ability to precisely detect obstacles and enemies in its environment. While our states in Q-learning also consider the position of enemies relative to Mario, we don't store their exact positions within the frame. The Forward agent, however, always looks ahead and focuses on the dangers that may lie ahead of it, and is therefore able to skillfully avoid or kill them.

5 Conclusions

Having set the objective to create a reinforcement learning agent using the Q-learning algorithm, we were ultimately successful. Though not performing better than all agents that we tested against, we were able to achieve a higher average score than three of the agents provided to us and underperformed against a rule-based agent that prioritized moving forward and avoiding obstacles. Ultimately, our agent did not consider all the attributes in the environment when learning, and we believe that extending our states to do so could significantly improve performance. Namely, our agent did not consider power-ups or collecting coins as a factor in the direction that it moved, although these moves would drastically improve a player's score. Moreover, our state design could be made more precise to record accurate positions of enemies. Further avenues of work could also include implementing different learning algorithms. Nonetheless, our agent was successful in our attempt to play the game using Q-learning and beat a majority of simple random and rule-based agents.

6 Contributions

We were all able to offer input on every aspect of the assignment and complete the code as well as the paper during our in-person meetings. We each proof-read the entire document as well.

References

Karakovskiy, S., and Togelius, J. 2012. The mario ai benchmark and competitions. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):55–67.

Kauten, C. 2023. gym-super-mario-bros: Super mario bros. for openai gym.

Nintendo. 2020. The official home for mario - history.

Togelius, J.; Shaker, N.; Karakovskiy, S.; and Yannakakis, G. N. 2013. The mario ai championship 2009-2012. *AI Magazine* 34(3):89.

Tsay, J.-J.; Chen, C.-C.; and Hsu, J.-J. 2011. Evolving intelligent mario controller by reinforcement learning.