CS229 Final Report
Reinforcement Learning to Play Mario

Yizheng Liao
Department of Electrical Engineering
Stanford University
Email: yzliao@stanford.edu

Kun Yi
Department of Electrical Engineering
Stanford University
Email: kunyi@stanford.edu

Zhe Yang
Google Inc.
Email: rabbiest@gmail.com

***Abstract*—In this paper, we study applying Reinforcement Learning to design a automatic agent to play the game Super Mario Bros. One of the challenge is how to handle the complex game environment. By abstracting the game environment into a state vector and using Q learning — an algorithm oblivious to transitional probabilities — we achieve tractable computation time and fast convergence. After training 5000 iterations, our agent is able to win about 90% of the time. We also compare and analyze the choice for learning parameters α and γ.**

I. Introduction

Using artificial intelligence (AI) and machine learning (ML) algorithms to play computer games has been widely discussed and investigated, because valuable observations can be made on the ML play pattern vs. that of a human player, and such observations provide knowledge on how to improve the algorithms. Mario AI Competition [1] provides the framework to play the classic title Super Mario Bros, and we are interested in using ML techniques to play this game.

Reinforcement Learning (RL) [2] is one widely-studied and promising ML method for implementing agents that can simulate the behavior of a player [3]. In this project, we study how to construct an RL Mario controller agent, which can learn from the game environment. One of the difficulties of using RL is how to define state, action, and reward. In addition, playing the game within the framework requires real-time response, therefore the state space cannot be too large. We use a state representation similar to [3], that abstracts the whole environment description into several discrete-valued key attributes. We use the Q-Learning algorithm to evolve the decision strategy that aims to maximize the reward. Our controller agent is trained and tested by the 2012 Mario AI Competition evaluation system.

The rest of this report is organized as follows: Section 2 provides a brief overview of the Mario AI interface and the Q-Learning algorithm; Section 3 explains how we define the state, action, reward to be used in RL; Section 4 provides evaluation results; Section 5 concludes and give some possible future work.

II. Background

In this section, we briefly introduce the Mario AI frameword interface and the Q-learning algorithm we used.

A. Game Mechanics and the Mario AI Interface

The goal of the game is to control Mario to pass the finish line, and gain a high score one the way by collecting items and beating enemies. Mario is controlled by six keys: UP (not used in this interface), DOWN, LEFT, RIGHT, JUMP, and SPEED. In the game, Mario has three modes: SMALL, BIG, and FIRE. He can upgrade by eating certain items(mushroom and flower), but he will lose the current mode if attacked by enemies. Other than Mario, the world consists of different monsters (Goomba, Koopa, Spiky, Bullet Bill, and Piranha Plant), platforms (hill, gap, cannon, and pipe) and items (coin, mushroom, bricks, flower). The game is over once SMALL Mario is attacked, or when Mario falls into a gap. For more specific descriptions, please see [3] and [1].

When performing each step, the Mario AI framework interface call could return the complete observation of 22 x 22 grids of the current scene, as shown in Figure 1. That is, an array containing the positions and types of enemies/items/platforms within this range. This is the whole available information for our agent.

The benchmark runs the game in 24 frames per second. The environment checking functions are called every frame. Therefore, while training we have to train and update the Qtable within 42 milliseconds.

B. ϵ-greedy Q-Learning

Q-learning treats the learning environment as a state machine, and performs value iteration to find the optimal policy. It maintains a value of expected total (current and future) reward, denoted by Q, for each pair of (state, action) in a table. For each action in a particular state, a reward will be given and the Q-value is updated by the following rule:

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t)+\alpha_{s,a}(r+\gamma \max(Q(s_{t+1}, a_{t+1}))) \quad (1)$$

In the Q-learning algorithm, there are four main factors: current state, chosen action, reward and future state. In (1), $Q(s_t, a_t)$ denotes the Q-value of current state and $Q(s_{t+1}, a_{t+1})$ denotes the Q-value of future state. $a \in [0, 1]$ is the learning rate, $\gamma\,[0, 1]$ is the discount rate, and r is the reward. (1) shows that for each current state, we update the Q-value as a combination of current value, current rewards and max possible future value.

We chose Q-learning for two reasons:

1) Although we model the Mario game as approximately Markov Model, the specific transitional probabilities between the states is not known. Had we used the normal reinforcement learning value iteration, we will have to train the state transitional probabilities as well. On the other hand, Q-learning can converge without using state transitional probabilities ("model free"). Therefore Q-learning suits our need well.
2) When updating value, normal value iteration needs to calculate the expected future state value, which requires reading the *entire* state table. In comparison, Q learning only needs fetching two rows (values for s_t and s_{t+1}) in the Q table. With the dimension of the Q table in thousands, Q learning update is a lot faster, which also means given the computation time and memory constraint, using Q table allows a larger state space design.

The learning rate α affects how fast learning converges. We use a decreasing learning rate $\alpha_{s,a}$ [**?**] different for different (s, a) pairs. Specifically,

$$\alpha_{s_t,a_t} = \frac{\alpha_0}{\text{\# of times action } a_t \text{ performed in } s_t} \quad (2)$$

the equation is chosen based the criteria of proposed by Watkins' original Q-learning paper. He shows the following properties of α is sufficient for the Q values to converge.

1) $\alpha(s_t, a_t) \to 0$ as $t \to 0$.
2) $\alpha(s_t, a_t)$ monotonically decreases with t.
3) $\sum_{t=1}^{\infty} \alpha(s_t, a_t) = \infty$.

One can easily verify the series 2 satisfy all the properties.

The discount factor γ denotes how much future state is taken into account during optimization. We evaluate under several γ values and chooses 0.6 as the final value. We will show that non-optimal learning parameters lead to highly degenerated performance in the evaluation section.

When training our agent, we actually used ϵ-greedy Q-learning to explore more states. The algorithm is a small variation of Q-learning: each step the algorithm chooses random action with probability ϵ, or the best action according to the Q table with probability $1 - \epsilon$. After performing the action, the Q table is updated as in 1.

III. Mario Controller Design Using Q-Learning

In this section, we describe how we design the state, action, and reward to be used in the Q-learning algorithm. We also briefly comment on why we design the state as it is.

A. Mario State

We discretize the environment description into the following integer valued attributes:

- Mario Mode: 0 - small, 1 - big, 2-fire
- Direction: The direction of Mario's velocity in current frame. 8 directions + stay. Total of 9 possible values;
- If stuck: 0 or 1. Set true if Mario doesn't make movement over several frames.
- If on ground: 0 or 1.
- If can jump: 0 or 1.
- If collided with creatures in current frame: 0 or 1.
- Nearby enemies: denoting whether there is an enemy in 8 certain directions in 3x3 window (or 4x3 window in large/fire Mario mode)
- Midrange enemies: denoting whether there is an enemy in 8 certain directions in 7x7 window (or 8x7 window in large/fire Mario mode).
- Far enemies: denoting whether there is an enemy in 8 certain directions in 11x11 window (or 12x11 window in large/fire Mario mode). Note the nearby, midrange, and far enemies attributes are exclusive.
- If enemies killed by stomp: 0 or 1. Set true if enemy killed by stop in current frame.
- If enemies killed by fire: 0 or 1. Set true if enemy killed by fire in current frame.
- Obstacles: 4-bit boolean indicating whether there exist obstacles in front of Mario. See figure 1

A Mario state thus needs 33 bits to encode, meaning the number of possible states is 2^33. However, in actual scenes no every state will appear. As we use a hashtable to store the q values, only states visited will be stored and tracked. At the end of training, we found there are only about 2000 states in the table, meaning the state space is very sparse.

The "stuck" state variable is added to tackle the situations where the action taken doesn't work as expected due to the "position rounding problem". For example, we notice when Mario is close to a tube, the JUMP+RIGHT action is chosen in order to jump up the tube, which is correct. However, when Mario is too close to the tube, JUMP key will not work and Mario won't move. As a result Mario will be performing the same action over and over (stucked). By adding a "stuck" state variable, we observe Mario is able to go into a new state and choose different actions in order to break the loop.

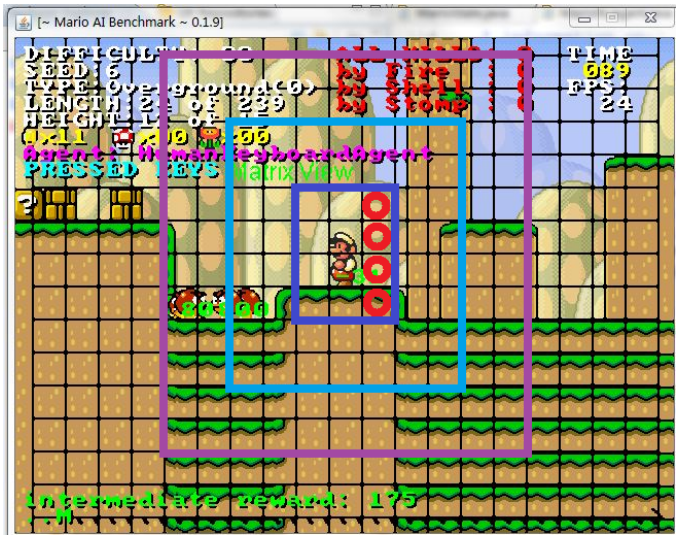

Fig. 1. Mario Scene

The figure 1 shows a typical scene of the Mario environment consisting of the platform and enemies. The area within dark blue, lighter blue, purple boxes indicates the range of nearby, midrange, and far enemies respectively. For instance, in the figure we have both mid-range and far enemies. In the easy level we coped, we don't distinguish different typed enemies. The four red circles denote the obstacles array: if there is obstacle the corresponding bit is set to 1.

B. Actions

The Mario agent performs one of 12 actions from the key combination {LEFT, RIGHT, STAY} x {JUMP, NOTJUMP} x {SPEED(FIRE), NOSPEED}.

C. Rewards

Our reward function is a combination of weighted state attribute values and the delta distance/elevation it performs from the last frame. Basically, moving forward, jumping onto higher platforms and killing enemies will be positively rewarded, whereas moving backward, colliding with enemies and being stuck will be negatively rewarded. We also let the reward of moving forward decrease when nearby enemies are present. Note we carefully design our state such that the reward $R(s)$ is only a function of the state s, not based on what actions are taken to reach s.

IV. Evaluation Results

For training the Q-learning algorithm, we firstly initialize the Q-table entries with a uniform distribution, i.e. $Q \sim \mathcal{U}(-0.1, 0.1)$. Then we trained the agent by 15000 iterations on a fixed level for the three Mario modes. The order is: train the level/episode 20 times for small Mario, 20 times for large Mario, 20 times for fire Mario, repeat, etc. We believe in this order the fire Mario will be able to use the Q table information from previous runs even when he is attacked and downgraded. During training the learning parameters are

$$\begin{aligned} \alpha(s_t, a_t) &= \frac{\alpha_0}{\text{\# of times } a_t \text{ performed in } s_t} \\ \alpha_0 &= 0.8 \\ \gamma &= 0.6 \\ \epsilon &= 0.3 \end{aligned}$$

For every 20 training episodes of each mode, we evaluate the performance by running 100 episodes on the current Q table and use the average metrics as the performance indicators. The evaluation episodes are run with $\alpha = 0$, $\epsilon = 0.01$, so the learning is turned off and random exploration is minimal. It is purely a test of how good the policy is.

We use 4 metrics to evaluate performance

1) A composite "score" combining weighted win status, kills total, distance passed and time spent.
2) The probability the agent beats the level.
3) The percentage of monster killed.
4) The time spent on the level.

Fig. 2. Evaluation Score

Figure. 2 shows the learning curves of evaluation score using the previously described optimized parameters. The discount factor, γ, is 0.6, meaning that the Q-learning algorithm tried to maximize the long-term reward. Obviously, our algorithm demonstrates a learning curve and quickly converges to the optimal solution after about 3000 training iterations. At the end of training cycles, our average evaluation score is around 8500. For some evaluation cycles, we can even achieve 9000 points, which is nearly the highest score human can achieve in one episode.

In order to show the generalization, we also tested the trained Q-learning algorithm with random episode seeds. The results show that for most random seeds, our trained algorithm performs reasonably good. The reason that one test performs bad is that there always be some unknown situations, to which Mario is not trained.

In addition, we plot the learning curves with fixed learning rate ($\alpha = 0$ throughout training) and low discount factor ($\gamma = 0.2$). As discussed above, in our training algorithm, we keep decreasing the learning rate. The figure indicates that with a fixed learning rate, when it converges, the converged solution is not optimal and the variance in scores is larger. A low discount factor means that the learning algorithm maximizes the short-term reward. In our learning algorithm, we gave positive reward for right movement and negative reward for left movement. If the algorithm try to maximize short-term reward, Mario will always move the right. However, in some situations,

Mario should stay or go left to avoid monsters. In this case, the short-term reward maximization is not the optimal solution.

Fig. 3. Winning Probability

Figure. 3 shows the winning probability learning curve. As the early stage of learning process, the winning probability is as low as 0.3. With the increase of training cycles, the winning probability increases and converges to around 0.9. For the low γ learning, even the average probability is increasing but the variance is very high. For the fixed α learning, the learning curve converges but the converged value is not optimal. The curve demonstrates that our trained agent is consistently good on beating the level!

Figure. 4 shows the percentage of monster killed. Since we generated the training episode by the same seed and setup, the total number of monsters within a episode is the same over training. At the beginning, the Q-values are generated randomly. Therefore, the killing percentage is very low. The learning curve shows fast convergence of the killing probability within a few training episodes, due to the high reward we gave. There are two reasons that we give high reward for killing. Firstly, the killing action is given high score in evaluation and therefore, we can achieve a high score in evaluation. Secondly, the Mario is safer when he kills more monsters.

In this figure, the learning curve with fixed α shows a similar performance to our optimal learning curve. The learning curve with low γ has very low mean and high variance.

Figure. 5 shows the time spent for the Mario to pass a episodes successfully. The optimal learning curve shows a fast

Fig. 4. Percentage of Monster Killed

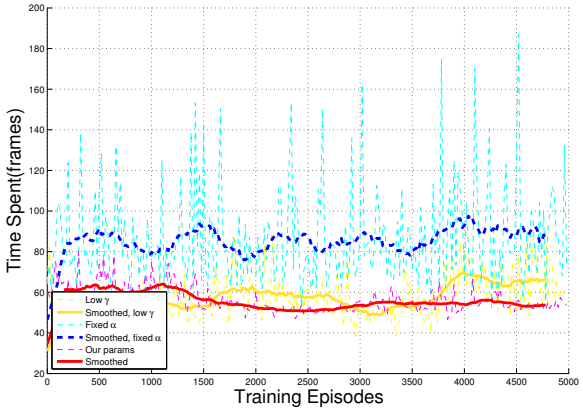

Fig. 5. Time Spent in Frames

convergence within 250 iterations. The learning curve with low γ has a similar performance as the optimal learning curve. The reason is that the short-term reward maximization forces the Mario to keep moving rightward. However, as we discussed above, it is not optimal since the Mario will collide creatures with high probability. The learning algorithm with fixed α needs more time to win because the converged policy is not the best.

V. Conclusion

In this project, we designed an automatic agent using Q-learning to play the Mario game. Our learning algorithm demonstrates fast convergence to the optimal Q-value with high successful rate. The optimal policy has high killing rate and consistently beat the level. In addition, our results show that the state description is general enough, that our optimal policy can tackle different, random environments. Further, we observe that long term reward maximization overperforms short term reward maximization. Finally, we show that using decaying learning rate converges to a better policy than using fixed learning rate.

Our work on our RL approach could be extended in a number of ways. For example, in our learning algorithm, we did not design the state for the Mario to grab mushroom and flowers. In addition, our algorithm focuses on optimizing the successful rate. Possible future work may include, but is not limited to:

- Extend our state to allow grabbing coins and upgrading.
- Vary the reward function to realize different objectives (e.g. killer-type Mario)
- Make the state design more precise to cope with the position rounding problem.
- Explore other RL approaches such as SARSA[**?**] and fuzzy-SARSA[**?**] to reduce state space and increase robustness.

We believe that our work provides a good introduction to this problem and will benefit the people with interests in using reinforcement learning to play computer games.

References

[1] J. Togelius, S. Karakovskiy, and R. Baumgarten, "The 2009 mario ai competition," in *Proceedings of the IEEE Congress on Evolutionary Computation*, Citeseer, 2010.

[2] R. Sutton and A. Barto, *Reinforcement learning: An introduction*, vol. 1. Cambridge Univ Press, 1998.

[3] J. Tsay, C. Chen, and J. Hsu, "Evolving intelligent mario controller by reinforcement learning," in *Technologies and Applications of Artificial Intelligence (TAAI), 2011 International Conference on*, pp. 266–272, IEEE, 2011.