

# Reinforcement Learning for Super Mario Game

Jiabin Chen, Runtian Zhang, Wenjian Dong

**Abstract**—With the great success of deep learning as well as deep reinforcement learning in the past of years, many Atari games have been solved well. In this paper, we design and train an agent to play the 1985 Nintendo game Super Mario Bros within the OpenAI Gym framework. The agent Mario must learn to navigate itself through the map, kill its enemies, jump over pipes and gaps, and finally reach the destination while maximizing the scores. We develop and test several classical algorithms including traditional rule-based algorithm, evolutionary algorithm, and deep Q-learning algorithm. Our agent can finish the first level with the first two methods, and reach a good position with the last method. We also make comparison among these methods in term of reward achieved, stability, and time consuming.

## I. INTRODUCTION

OpenAI released the full version of Gym Retro, a platform for reinforcement learning research on games. In this project, we choose a classical game, Super Mario Bros, as the environment given by Gym Retro and we focus on using Reinforcement Learning techniques to play this game.

Reinforcement Learning (RL) [3] is one widely-studied and promising ML method for implementing agents that can simulate the behavior of a player [4]. In this project, We focus on how to formulate useful observation space as input, how to design the reward function to shape the behavior of the agent, and how to build an RL Mario controller agent that can learn its strategy based on the game environment. We exploit several algorithms, including traditional rule-based algorithm, evolutionary algorithm, and deep Q-learning algorithm.

In the following sections, we will explain in details about the environment and algorithms applied to agent, present a training curve and give some analysis and discussion.

## II. BACKGROUND AND RELATED WORK

Super Mario Bros is a popular game around the world and there are many AI algorithms on the game. In this section we briefly discuss some typical algorithms.

**Traditional Q Learning and SARSA:** This is a typical approach in reinforcement learning. Gabe Grand and Kevin Loughlin investigate several forms of traditional Q learning algorithms and SARSA algorithm [7].

**Deep Q Learning:** As the deep learning develops recent years, scientists have been exploiting the deep learning method to evaluate Q value. Sean Klein implement this method for example [9]. Furthermore, there are many other variants of deep Q learning which achieve the state-of-art in some tasks, such as double Deep Q Network, Dueling Deep Q Network, Prioritized Experience Replay, etc.

Policy Gradient based methods show sometimes a better performance, such as REINFORCE, A2C, A3C, Proximal Policy Optimization, etc.

**Evolutionary algorithm:** evolution algorithms try to find effective strategies through a family of population-based trial and error problem solvers. It has widely been used in game AI design.

## III. THE ENVIRONMENT

The goal of our agent Mario is relatively simple, begin on the left edge of a level and navigate through the map and reach the flagpole without dying, while trying to maximize the score. For reaching an optimal level of gameplay, the agent learns how to avoid/kill its enemies, jump over pipes, spaces and adjust its speed to improve the score.

To formulate Super Mario Bros as an Reinforcement Learning problem, we specify three components: the state space, the action space and the reward function.

- **State Representation:** the state of the game is represented by a  $13 \times 16$  tile grid with integers 0-3:
  - 0 - empty space
  - 1 - object such as ground, pipe or coin
  - 2 - enemy
  - 3 - Mario
- **Action Space:** the representation on the action space is dependent on the FCEUX emulator aka Nintendo Controller. There are 6 basic actions that can be either pressed (1) or not pressed (0) in the game environment: UP, RIGHT, DOWN, LEFT, JUMP and FIRE. This leads to 64 possible combination of actions. However, there are only 9 actions that make logical sense and have an impact on the game. For example, pressing left and right is an action on the emulator but has no effect within the environment.
- **Reward Function:** its the amount of reward achieved by the agent during the previous action. This is what is used to shape Marios behavior. The Gym Super Mario environment by Philip Paquette provides a default reward function, which changes in respect to Marios distance among the level. Mario gets +1 reward if it moves right, -1 if it moves left and 0 if it stays in place. However, this isnt all, Mario also gets +100 rewards for killing enemies, jumping over pipes and avoiding obstacles while gets -300 rewards in case of die.

The figure Fig. 1 shows a screenshot of a game and the table TABLE. I shows the corresponding numerical state at the same moment.

## IV. THE AGENT

### A. Rule-based agent

The first method we try is to code the behavior of Mario directly according to the current state. We analyze



Fig. 1. An example of the window.

State representation of the current window															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

TABLE I

THE STATE CORRESPONDING TO THE FIG.1 , WHICH IS A  $13 \times 16$  MATRIX

- the gap in front of Mario
- the enemy's position and number in front of Mario
- the obstacle in front of Mario

and we make decisions on

- when Mario should jump
- how long the jump button is pressed
- the direction of Mario's movement.

By this method, we develop a agent that could successful pass the first level of the game. However, in the following level, Mario can't deal with new situations.

As a traditional method, the advantage of this approach is fast. It doesn't require much computing resource to train the agent. The limitation of this method is that for every new situation we need to handle it manually, and to deal with complicated situation could be difficult.

### B. NeuroEvolution of Augmenting Topologies (NEAT)


NeuroEvolution of Augmenting Topologies (NEAT) is a genetic algorithm (GA) for the generation of evolving artificial neural networks. It alters both the weighting parameters and structures of networks, attempting to find a balance between the fitness of evolved solutions and their diversity. We use the NEAT-Python package since it has already included everything we need.

This algorithm will initially create multiple species, each individual of a specie is a neural network with its proper structure and weights. During each epoch, every individual of all species will be run one time to evaluate its performance. The individuals which had a good score would be conserved for the next epoch. The weights and structure of the network may mutate during the process to introduce the possibility of amelioration of the performance.

We mainly tried 3 different neural network input:

1) *Input: whole image of the game:* At first, we chose the whole image output of the game as the input of our network. But this data is too difficult to use since the dimension is too big (screen resolution:  $256 \times 240$ ), so we tried to compress the screen data by OpenCV to  $32 \times 30$ . Still, the different colors in the image create lots of difficulties for the agent to learn which block is solid empty and where are the enemies.

2) *Input: Map/block information extracted from the system ram:* Then we looked for data directly from the computer memory to get information like block placement of the current screen, mario's position, monsters' position etc. Then, using these information, we generated a  $5 \times 5$  of block information just ahead of mario's position because we have already limited the possible actions of mario into only jumping and moving forward so that the blocks behind mario is less important than the block in front of mario. We can say that we used what mario 'sees' as input.



0	0	0	0	0
0	1	1	1	0
0	0	0	-1	0
1	1	1	1	1
1	1	1	1	1

Fig. 2. What Mario sees: 0=empty; 1=solid; -1=enemy.

3) *Input: Heuristic information transformed from the map information:* In order to reduce the input dimension again, we introduced many heuristic summary of the map information as inputs: if a enemy is within 4 blocks in front of Mario; if a block is within 4 blocks in the front of Mario; if Mario is in the air; if a pit is in front of Mario etc. All the information that we, as human, think are useful.

When each individual finish their run, a score of their current x position will be noted, which will be used to rank the individuals. Finally, we hope that one among all these individuals may reach the end of the level.

### C. Deep Q-learning agent

We recall that Q-learning[7] performs value iteration to find the optimal policy. Using the Q-table for pair of state and action, it maintains a value of expected total reward. For each

action in a particular state, a reward will be given and the Q-value is updated by the following rule:

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

where  $Q(s_t, a_t)$  denotes the Q-value of current state and  $Q(s_{t+1}, a_{t+1})$  denotes the Q-value of next state.  $\alpha \in [0, 1]$  is the learning rate,  $\gamma \in [0, 1]$  is the discount rate, and  $r$  is the reward. But storing and updating a Q-table can become ineffective in big state space environments, e.g, in our Super Mario case. Instead of using a Q-table, Deep Q-Learning use a Neural Network that takes a state and approximates Q-values for each action based on that state. There are two main tasks when training our agent: take an action based on current state and optimize the neural networks. To take an action, we actually used  $\epsilon$ -greedy policy, in which  $\epsilon$  will start at EPS\_START and will decay exponentially towards EPS\_END. EPS\_DECAY controls the rate of the decay. These are three hyper-parameters to be tuned. Within this policy, each step the agent chooses random action with probability  $\epsilon$ , or with probability  $1 - \epsilon$ , our Neural Network takes a stack of four pre processing frames as an input, pass it to three convolution layers and then two fully connected layers, and output a vector of Q-values for each action possible in the given state so that we can take action who has the biggest Q-value of this vector. To train our neural networks, like the original paper[1], we use experience replay memory who stores the transitions that the agent observes, allowing us to reuse this data later. By sampling from it randomly, the transitions that build up a batch are decorrelated. We add a pair (state, action, next\_state, reward) into replay memory after each step. For our training update rule, with the fact that every  $Q$  function for some policy obeys the Bellman equation:

$$Q^\pi(s, a) = r + \gamma Q^\pi(s', \pi(s')) \quad (2)$$

we can define the temporal difference error,  $\delta$ :

$$\delta = Q(s, a) - (r + \gamma \max_a Q(s', a)) \quad (3)$$

We minimize the mean squared loss over a batch of transitions sampled from the replay memory to update our neural networks.

#### D. DQN with Prioritized Experience Replay

Since we sample the batch of experiences uniformly when we train our agent, the experiences that occur rarely but poor action decision have practically no large chance to be selected. To address in this issue, Prioritized Experience Replay (PER) was introduced in 2015 [11]. The basic idea is to change the sampling distribution by using a criterion to define the priority to each experience inserted in memory. We want to take in priority experience where there is a big difference between our prediction and the TD target, since intuitively it means that we have a lot to learn about it. Obviously, we cant just use a priority queue to choose greedily the experiences having high priority, because it will lead to always training the same experiences then lead to over-fitting. So we should formulate a probability distribution to these experiences based

on their priorities. Concretely, the probability of  $i$ th experience is expressed as:

$$p(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \quad (4)$$

where  $p_i = |\delta_i| + \epsilon$  is the priority of the  $i$ th experience and the exponent  $\alpha$  determines how much prioritization is used, with  $\alpha = 0$  corresponding to the uniform case. Notice that  $\epsilon$  is a small positive constant that prevents the edge-case of transitions not being revisited once their error is zero. There is a another issue that prioritized replay introduces bias since it changes the original distribution. We can correct this bias by using importance-sampling (IS) weights:

$$\omega_i = \left(\frac{1}{N} \frac{1}{P(i)}\right)^\beta \quad (5)$$

The role of  $\beta$  is to control how much these importance sampling weights affect learning. In practice, the  $\beta$  is annealed up to 1 over the duration of training, as these weights are more important in the end of learning when our Q values begin to converge. Now when we can update the network weights by using  $\omega_i \delta_i$  instead of  $\delta_i$ .

To implement this in practice, we need to use SumTree as data structure instead of sorting an array. In fact, a SumTree is a Binary Tree(a tree with only a maximum of two children for each node) and the leaves contain the priority values as well as a data array that points to leaves contains the experiences. To sample a mini batch of size  $k$ , the leaves will be divided into  $k$  ranges and then an example will be uniformly sampled from each range. Updating the tree and sampling from tree will be really efficient  $\mathcal{O}(\log(n))$ .

## V. RESULTS AND DISCUSSION

### A. Rule-based algorithm

As the environment is fixed, the playing process of rule-based algorithm is stable. It does not need to train in order to pass the level, and the passing rate is nearly 100%. A video demonstration of the process can be found on the Youtube through link: <https://youtu.be/6ULYBMAo1cM>.

The disadvantage of this approach is that new manual coding is needed for new kind of situation. For example, in the level 2, there is an enemy come from the rear side with high speed. Mario has never encountered such a situation, and it needs new dealing strategy coded by human.

### B. NeuroEvolution of Augmenting Topologies (NEAT)

1) *Input: whole image of the game:* The training was too slow even with the help of OpenCV to reduce the resolution.

2) *Input: Map/block information extracted from the system ram:* The training was fast enough, but still hard for our agent to find a solution to the game. We can see in the figure below, our agent has reached at the distance about 2000 and 2400 and it can not surpass this point because in the actual game, these two points correspond to two well where Mario might fall to death. This is hard for the agent to learn since they are in the final part of this level so that Mario won't have many chance to experience this part and learn.

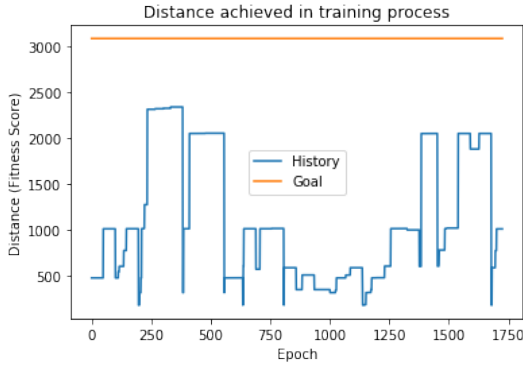


Fig. 3. Distance reached by NEAT using map information extracted from ram.

3) *Input: Heuristic information transformed from the map information:* We have conducted several times of experiments of the approach. Usually within about 500 epoch of training, approximately 2 hours, one agent could reach the end of the level. A link of the demonstration video that we made is on youtube: <https://youtu.be/E0zNcqcgG8g>

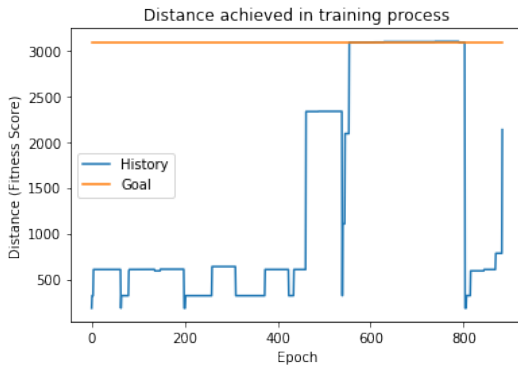


Fig. 4. Distance reached by NEAT using heuristic information.

Then we also tried other levels, where our agent didn't work too well since these levels are usually much more complex than the first one with more traps and situation that the agent has never seen.

### C. Deep Q-Learning Approach

We implement deep Q-learning method in two different versions.

- **Jump-and-reward version**

Before the prioritized-replay deep Q-learning, we first implement a simpler version of deep Q-learning algorithm – a jump-and-reward version. In this version, besides the reward when Mario moving right, we give the agent a small additional reward whenever it chooses to jump. We do this to naively encourage Mario to jump more.

In the Fig.5 we see that Mario could generally reach or pass 900-distance after 40 episodes of training. 900-distance means Mario pass 4 enemies and 2 high obstacles.

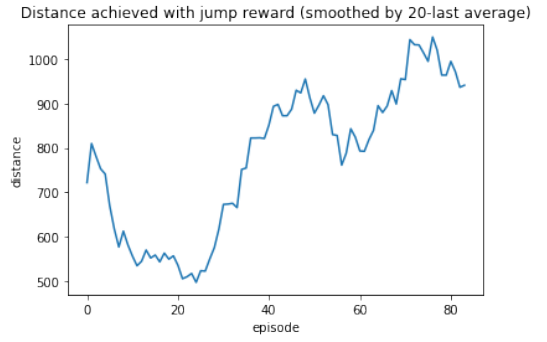


Fig. 5. Distance achieved in training process for jump-reward deep Q-learning

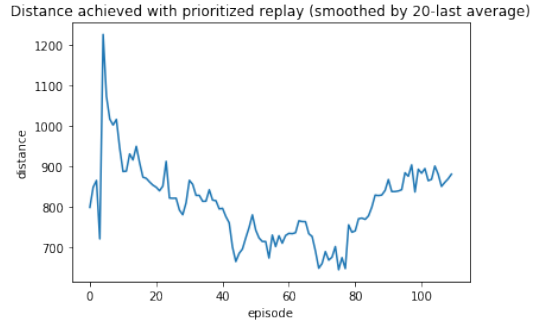


Fig. 6. Distance achieved in training process for prioritized-replay deep Q-learning

- **Prioritized-replay version**

In the Fig.6 we see that Mario's performance staggers around 60th episode and reaches 900-distance at around 100th episode.

In general, as the environment of Super Mario Bros is complex and precise, our current implementation still stagger to learn to handle situations precisely. For example, often, to kill an enemy or to be killed by an enemy is very close in term of the position where Mario falls. Our algorithm still can not learn to handle this efficiently. As the result, the distance Mario achieves is still quite noisy. Another limiting factor is the computation power. Currently our deep Q-learning algorithm requires a very large computation power and we do not have enough time to train it sufficiently.

## VI. CONCLUSION AND FUTURE WORK

In this project, rule-based algorithm shows that with prior human knowledge Mario can reach the end of the special level, but fails to generalize this behavior to other environment, even if the similar one. But this method can give us some intuition about the environment and the design of reward function. NeuroEvolution of Augmenting Topologies (NEAT) performs relatively well in the first level of this game, so we will try to generalize this performance to other more complicated level. Deep Q-learning approach seems poor if we just train 100 episodes for few hours. We think that exploring the policy gradient based methods, like Advantage Actor Critic (A2C), Asynchronous Advantage Actor Critic (A3C), Proximal Policy Optimization, etc, is actually essential for Mario agent to learn

a much better policy in this pretty complicated game environment. The full code is available here: [https://drive.google.com/open?id=1ADRzmoNBA4HYupiNTqb\\_t6Kp205Yy-GN](https://drive.google.com/open?id=1ADRzmoNBA4HYupiNTqb_t6Kp205Yy-GN)

#### REFERENCES

- [1] Human-level control through deep reinforcement learning, *Nature*, 2015
- [2] D. Barber. Bayesian Reasoning and Machine Learning, *Cambridge University Press*, 2012.
- [3] J. Read. Lecture III - Search and Optimization. *INF581 Advanced Topics in Artificial Intelligence*, 2019.
- [4] D. Mena et al. A family of admissible heuristics for A\* to perform inference in probabilistic classifier chains. *Machine Learning*, vol. 106, no. 1, pp 143-169, 2017.
- [5] O. Vinyals et al. StarCraft II: A New Challenge for Reinforcement Learning. <https://arxiv.org/abs/1708.04782>, 2017.
- [6] Sergey Karakovskiy and Julian Togelius, The Mario AI Benchmark and Competitions, [julian.togelius.com/Karakovskiy2012The.pdf](http://julian.togelius.com/Karakovskiy2012The.pdf), 2009
- [7] Gabe Grand and Kevin Loughlin, Reinforcement Learning in Super Mario Bros - CS 182 Final Project Report, [gabegrand.com/files/super\\_mario\\_rl.pdf](http://gabegrand.com/files/super_mario_rl.pdf), 2018
- [8] Yizheng Liao et al, CS229 Final Report Reinforcement Learning to Play Mario, <http://cs229.stanford.edu/proj2012/LiaoYiYang-RLtoPlayMario.pdf>
- [9] Sean Klein, CS229 Final Report, Deep Q-Learning to Play Mario, <http://cs229.stanford.edu/proj2016/report/klein-autonomousmariowithdeeppreinfreinforcementlearning-report.pdf>
- [10] Alejandro Baldominos et al, Learning Levels of Mario AI Using Genetic Algorithms, [https://www.researchgate.net/publication/299778811\\_Learning\\_Levels\\_of\\_Mario\\_AI\\_Using\\_Genetic\\_Algorithms](https://www.researchgate.net/publication/299778811_Learning_Levels_of_Mario_AI_Using_Genetic_Algorithms)
- [11] Tom Schaul, John Quan, Ioannis Antonoglou and David Silver, Prioritized Experience Replay, <http://arxiv.org/abs/1511.05952>