Teaching Mario to Play Mario: Reinforcement Learning on *Super Mario Bros.*™

Guanya Shi, Botao Hu, Yan Wu
California Institute of Technology
Pasadena, CA 91125
`{gshi, bhu2, ywu5}@caltech.edu`

Abstract

We present a deep learning model to successfully learn control policies from high-dimensional input data using reinforcement learning. The model is based on the idea of Deep Q-Network (DQN), with convolutional neural network trained by Q-learning algorithm, whose input is tile representation of the screen and output is a value estimation function. Also, replay buffer, target network and double Q-learning are applied to lower data dependency and approximate real gradiant descent. We applied our model to *Super Mario Bros.*™, and get some good preliminary results.

1 Introduction

Reinforcement learning is a set of goal-oriented algorithms, which learn from its experience and optimize the agent control over the environment. However, to apply RL in reality, it has been a known challenge for agents to learn directly from the real-world high-dimensional sensory inputs. The majority of successful RL applications utilize hand-crafted features, while there is an increasing amount of work that have been made in making learning more natural – deriving feature representation from raw high-dimensional inputs, and generalizing past experience to new situations based on them.

Playing computer games has gained popularity in machine learning, where valuable observations have been made and used in developing and improving algorithms. Minh and his group has developed an artificial agent, called deep Q-network. Using raw pixels and game score as inputs, it achieved similar capability as a professional human games tester over 49 Atari 2600 games. *Super Mario Bros.*™is a series of platform video games created by Nintendo. In this project, we study to construct an RL Mario agent, which learns from the game environment. We use a tile-based state representation, simplify reward as the total distance Mario traveled, and use convolutional neural network together with Q-Learning algorithm to evolve decision strategy that aims to maximize the reward.

2 Environment and Interface

2.1 Overview

The environment of our project is managed by **OpenAI Gym**. The *Super Mario Bros.*™game is built into Gym through an open source python linking project on **GitHub**. This python project communicates with the Nintendo Entertainment System (NES) emulator **FCEUX 2.2.3**, which runs the game with a variety of specific settings. We will stick to **Level 1-1** of *Super Mario Bros.*™as the training and testing environment for reinforcement learning. The pipeline of overall control flow in this project's environment and interface is shown below in figure 1.

Caltech CS159 final project report.
Code: `https://github.com/GuanyaShi/Training-DQN-to-play-Super-Mario-Bros.`

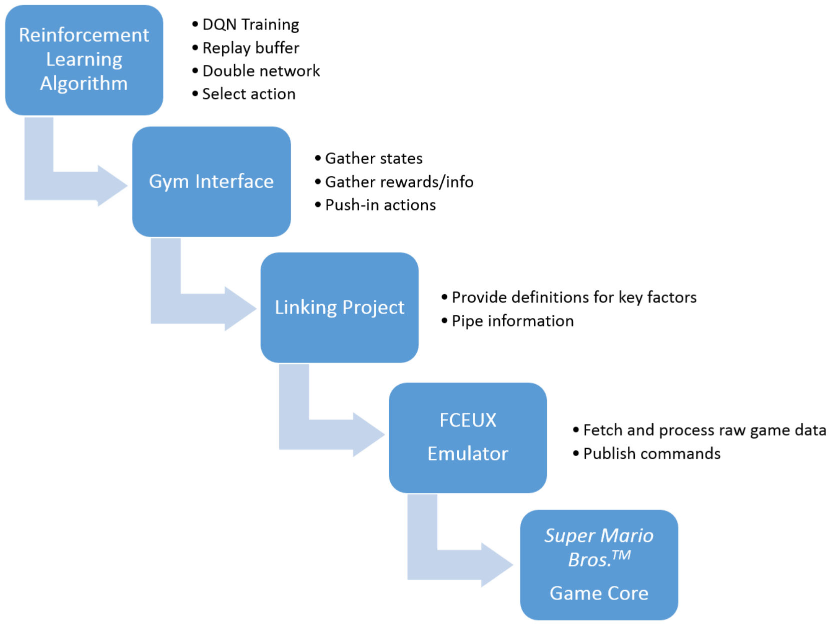


Figure 1: Pipeline of overall control flow

2.2 Emulator

As described above, we need NES emulator to run the game. In this project, we use FCEUX. It is an "all in one" emulator that is accurate and powerful, providing tools for debugging, Lua scripting and other advanced requirements. When starting a new game, FCEUX will be launched first, and a thread will be opened listening to incoming pipe (instructions), the message is processed and written in output file, communicating actions to game. When Mario agent dies, environment will be reset for a new game, and FCEUX is launched first and repeat the process above.

2.3 Game Interface

As we can see in figure 1, the game interface consists of two parts: Gym interface and Linking project.

Gym interface basically does two things: (1) Information Gathering. Gym interface periodically gathers four key factors: state, action, reward and information, which are universally needed for reinforcement learning algorithms. These factors take different forms, and are defined and specified by the game itself. (2) Action Publishing. After RL models figure out what action to take as the next step, the Gym interface is in charge of publishing the action to the lower level controller of the game.

As mentioned before, the linking project is an open source python program. This program is essential because it installs *Super Mario Bros.*™into the Gym library so that we can abstract complex control issues and focus on reinforcement learning algorithm itself. The functionality of this program can be summarized as: (1) Pipe information. The program modifies the data format of information, makes it compatible with upper/lower level controller, and pipes them. (2) Provide definition. This key functionality requires the program to explicitly define the format of action, state, reward and information as the foundation of reinforcement learning, which will be discussed in detail in the next section. (3) Control the emulator. Despite publishing command specified by chosen action to the emulator, the program can also perform controls such as close, restart, or interrupt the game.

2.4 Definition of Key Factors

2.4.1 Interaction Frequency

Super Mario Bros.™is a frame-based game, that is, commands and feedbacks are published and listened on a frame basis. The frame-per-second (fps) rate determines the frequency of game interaction.

In this project, actions are selected and published in each frame, and similarly, feedbacks are collected in each frame. Generally, all the following definitions refer to single-frame measurements.

2.4.2 State

The linking program provides two choices of states: Regular state and Tile state.

Regular state simply returns all the information: a 256×224 array representation of the screen, where each element is a red, blue and green (RGB) value.

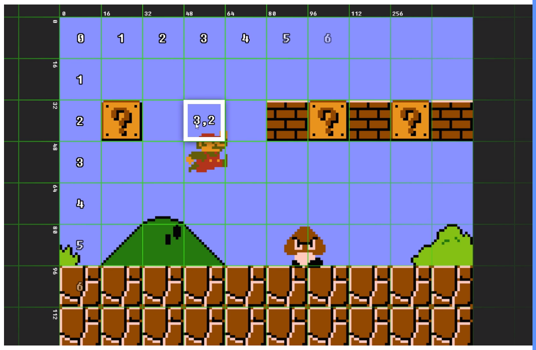
Tile state is a generalized state. In *Super Mario Bros.*™, all the inner representation of character, enemy and obstacles are based on tiles (or say grids). That is, the screen is divided into tiles as the size unit, and objects tend to span one or more tiles. Also, collision detection and physical engine of this game are based on occupancy tiles. An illustrative example is shown in figure 2a below[1].

Tile state is very useful because it extracts information from a complicated screen, while maintaining the correctness (it is the actual representation of the game world). Tile state mode will return a 13×16 array, where each element can have one of the following values:

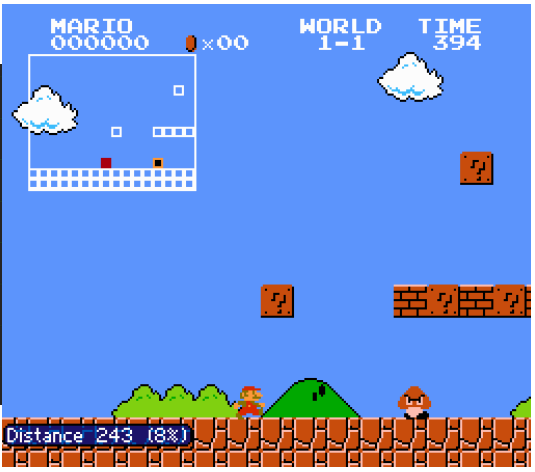
- 0 - Empty space.
- 1 - Object (e.g., platform, pipes, question blocks, etc).
- 2 - Enemy (e.g., Goomba, Koopa Troopa, etc).
- 3 - Mario (including tall Mario and aggressive Mario).

For computation efficiency, we choose tile state representation. Also, it is essential to take past data into consideration, otherwise Mario would be confused about his state (for example, with only one image of Mario floating in the air, we do not know whether Mario is jumping up or falling down). So, each time we gather the data of last eight frames (including the current frame), say $\mathcal{F} = \{1, 2, 3, 4, 5, 6, 7, 8\}$, and choose four frames with stride II, which are $\mathcal{F} = \{1, 3, 5, 7\}$, and frame 1 is the current frame. Of course, they are all tile representations.

In this way, our state s is a $4 \times 13 \times 16$ tensor in PyTorch. During the training, we can see a colored tile image on the top left of the GUI, as shown in figure 2b.



(a) Tile representation of *Super Mario Bros.*™ world



(b) Screenshot of running console

Figure 2: Tile state representation illustration

2.4.3 Action

The typical controller of Nintendo Entertainment System (NES) is shown in figure 3. As we can see, apart from SELECT and START buttons which will not be used during the game process, the other

[1]YouTube, Code Super Mario in JS (Ep 5) - Tile Collision

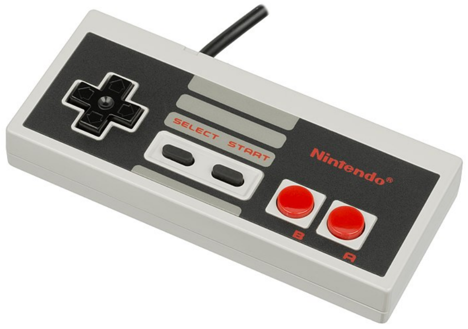


Figure 3: Controller of Nintendo Entertainment System

six buttons, which are referred as UP, DOWN, LEFT, RIGHT, JUMP (B), ATTACK (A), are often used. More importantly, one can input the combination of either JUMP or ATTACK and one of four direction buttons. So, including empty action, there should be $1 + 6 + 2 \times 4 = 15$ legal command inputs.

In *Super Mario Bros.*™, however, UP are only used in ladder climbing which does not appear Level 1-1, DOWN are only used in pipe diving (unnecessary) or avoiding flying obstacles (not in Level 1-1), and ATTACK are used by aggressive Mario which is hard and unnecessary to obtain. For simplicity, we exclude these inputs from our action space. Moreover, since Mario keeps going right, we also exclude LEFT button to avoid confusion (it is possible to clear Level 1-1 without LEFT button; it is a common behavior for most human players). As a result, we now have four things in our action space:

- 1 - EMPTY action.
- 2 - JUMP.
- 3 - RIGHT.
- 4 - RIGHT + JUMP.

The action space turns out to be $\mathcal{A} = \{1, 2, 3, 4\}$.

2.4.4 Reward

The reward of this game is defined as the distance Mario have passed from the last frame to the current frame (i.e., the frame-wise speed of Mario). The distance is measured horizontally from the starting point to Mario, ignoring height. That is, the faster Mario runs to the right, the more reward he can get. Additionally, in some models, when Mario gets stuck (e.g., in front of a tall pipe), instead of getting 0 reward, he will get a small negative reward (e.g., -0.01) for punishment.

2.4.5 Done

Done is a boolean variable indicating the game is ended or not. Done is FALSE in normal game plays; Done will be TRUE when Mario dies or clears the Level. When Done is TRUE, the linking program would force the emulator to restart the environment and go to next training epoch.

2.4.6 Other Information

The Gym interface enables the emulator to feed other informations as a dictionary variable. Useful information includes:

- Distance - this is used by the reward function. Distance is measured in game's level, and provided by the physical engines of the game.
- Life - indicates Mario is alive or not. This provides another way to monitor Done variable.
- Time - how much time left in this Level. This might be a potential factor to determine reward function, since we want Mario to be fast.

Now that we have everything set up, it's time to introduce our algorithms.

3 Algorithm

3.1 Basic Q-learning algorithm

The goal of the agent is to interact with the emulator by selecting actions in a way that maximizes future rewards. We make the standard assumption that future rewards are discounted by a factor of γ per time-step, and define the future discounted return at time t as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where T is the time-step at which the game terminates. We define the optimal action-value function $Q^*(s, a)$ as the maximum expected return achievable by following any strategy, after observing some state s and then taking some action a, $Q^*(s, a) = \max \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$, where π is a policy mapping states to actions.

As mentioned in section 2, our state s is a $4 \times 13 \times 16$ tensor, and our action a is selected from the set of legal game actions, $\mathcal{A} = \{1, 2, 3, 4\}$.

The optimal action-value function obeys an important identity known as the Bellman equation. This is based on the following intuition: if the optimal value $Q^*(s', a')$ of the state s' at the next time-step was known for all possible actions a', then the optimal strategy is to select the action a' maximizing the expected value of $r + \gamma Q^*(s', a')$,

$$Q^*(s, a) = \mathbb{E}[r + \gamma \max_{a'} Q^*(s', a') | s, a]. \tag{1}$$

The basic idea behind Q-learning algorithms is to estimate the action-value function, by using the Bellman equation as an iterative update, $Q_{i+1}(s, a) = \mathbb{E}[r + \gamma \max_{a'} Q^*(s', a') | s, a]$. It is common to use a function approximator to estimate the action-value function, $Q(s, a; \phi) \approx Q^*(s, a)$. In the reinforcement learning community this is typically a linear function approximator, but sometimes a non-linear function approximator is used instead, such as a neural network. In this project, we use neural network function approximator (both NN and CNN) with weights ϕ as our Q-network.

Finally, here comes the standard gradient descent algorithm. First, we take some action a_i and observe (s_i, a_i, s_i', r_i) from the emulator; second, we apply the gradient descent algorithm:

$$\phi \leftarrow \phi - \alpha \frac{\mathrm{d}Q(s_i, a_i; \phi)}{\mathrm{d}\phi} (Q(s_i, a_i) - [r(s_i, a_i) + \gamma \max_{a'} Q(s_i', a_i'; \phi)]). \tag{2}$$

Note that this algorithm is model-free and off-policy.

3.2 Replay buffers

One issue of the basic Q-learning algorithm is that it assumes the data samples to be independent, while actually sequential states are strongly correlated in our emulator. Furthermore, in RL the data distribution changes as the algorithm learns new behaviors, which can be problematic for deep learning methods that assume a fixed underlying distribution.

To alleviate the problems of correlated states and non-stationary distributions, we use an experience replay mechanism which randomly samples previous transitions, and thereby smooths the training distribution over many past behaviors. In this project, each time we will sample a batch from our replay buffer $\mathcal{B}$.

3.3 Target networks and double Q-learning

Another issue of the basic Q-learning algorithm is that it is not real gradient descent, because in equation (2), $r(s_i, a_i) + \gamma \max_{a'} Q(s_i', a_i'; \phi)$ is not the true target value, and it also changes every gradient step.

To solve this issue, we use "target network" to represent the target value, and use "policy network" to represent our current policy. Then equation (2) will change to:

$$\phi \leftarrow \phi - \alpha \frac{\mathrm{d}Q(s_i, a_i; \phi)}{\mathrm{d}\phi} (Q(s_i, a_i; \phi) - [r(s_i, a_i) + \gamma \max_{a'} Q(s_i', a_i'; \phi')]), \tag{3}$$

where $Q(s, a; \phi)$ is the policy network and $Q(s, a; \phi')$ is the target network. Every K steps, we will update the target network parameters: $\phi' \leftarrow \phi$.

However, we still have issue even if we use target networks. In the last term of equation (3), we have

$$\max_{a'} Q(s', a'; \phi') = Q(s', \operatorname*{argmax}_{a'} Q(s', a'; \phi'); \phi'), \tag{4}$$

which means the action is selected from ϕ', and the value approximation is also from ϕ'. This will bring overestimation in Q-learning.

To solve this issue, we use double Q-learning in this project, which slightly changes equation (4) to:

$$Q(s', \operatorname*{argmax}_{a'} Q(s', a'; \phi); \phi'), \tag{5}$$

which means we use the policy network to select action and use the target network to approximate the value.

Algorithm 1 Deep Double Q-learning with Replay Buffer

```
 1: Initialize replay memory $\mathcal{B}$
 2: Initialize policy network $Q(s, a; \phi)$ with random weights
 3: Initialize target network $Q(s, a; \phi')$ with random weights
 4: while True do
 5:     Save target network parameters: $\phi' \leftarrow \phi$
 6:     for episode=1,$K$ do
 7:         With probability $\epsilon$ select a random action $a$
 8:         Otherwise select $\operatorname*{argmax}_a Q(s, a; \phi)$
 9:         Observe $\{s, a, s', r\}$ from the emulator and add it to $\mathcal{B}$
10:         Sample a batch $\{\mathbf{s}, \mathbf{a}, \mathbf{s}', \mathbf{r}\}$ from $\mathcal{B}$
11:         Batch gradient descent:
```

$$\phi \leftarrow \phi - \alpha \frac{\mathrm{d}Q(\mathbf{s}, \mathbf{a}; \phi)}{\mathrm{d}\phi} (Q(\mathbf{s}, \mathbf{a}; \phi) - [\mathbf{r} + \gamma Q(\mathbf{s}', \operatorname*{argmax}_{\mathbf{a}'} Q(\mathbf{s}', \mathbf{a}'; \phi); \phi')])$$

The algorithm we apply in our project is shown in algorithm 1.

3.4 Q-network structure

In this project, the input to the Q-network is $4 \times 13 \times 16$ tensor to represent the state s, and output is four value corresponding to four different $Q(s, a)$ since our action space is $\mathcal{A} = \{1, 2, 3, 4\}$. So we design two different networks. One is neural networks (NN) and another is convolutional neural network (CNN). The network topologies are shown in figure 4.

4 Result and Discussion

We mainly focus on three criteria to measure a model's performance:

- Total distance, namely, is the total distance Mario have passed in Level 1-1.
- Total duration, which is the total number of frames Mario experienced from the start to the end of the game.
- Average speed, which is total distance divided by total duration.

As the fundamental criterion of performance, the total distance in a typical training process is shown in figure 5. As we can see, total distance keeps going up as more episodes are trained.

Once training is complete, we test the performance of each model by loading the trained target neural network, selecting action with a greedy random probability of 0.05 (i.e., there is 0.05 probability that Mario agent selects actions randomly, in order to avoid caveats), and compute total duration, total distance and average speed averaged over 100 testing epochs. Results are listed in table 1 below. Our

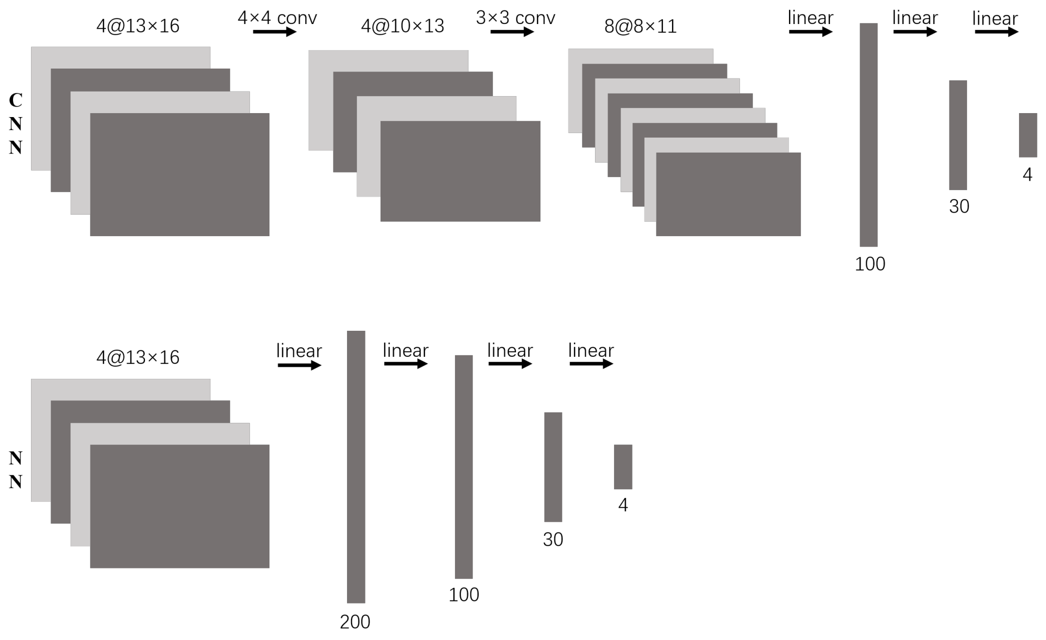


Figure 4: CNN and NN network topologies

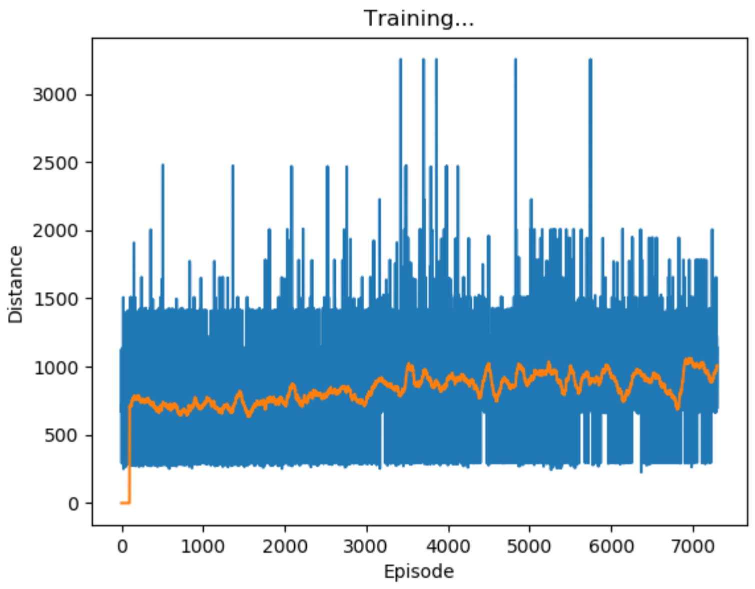


Figure 5: Total Distance Trend in Training Process

baseline for comparison is pure random strategy, that is, each time this model picks an action from our action space $\mathcal{A}$ uniformly at random.

As we can see in the table, Convolution Neural Networks (CNN) generally outperform linear Neural Networks (NN) with shorter duration, longer distance, and higher average speed, and it is natural since CNN extracts more details to help training. By comparing training epochs and learning rates among models, we also observe that the model who learns most performs the best. Additionally, the last model acts particularly bad, which infers that it may stuck in a local minimum, where Mario gets trapped in front of a high pipe until time is up, and he dies. This particular situation will be discussed

Table 1: Performance Comparison among Models

Net	Epochs	Learning rate	Total Duration	Total Distance	Average Speed
Pure Random Strategy			273.58	744.61	4.01
CNN	5800	0.0005	215.06	931.35	4.77
CNN	7300	0.0001	368.44	858.61	4.18
NN	5300	0.0001	463.93	793.15	4.27
NN	8400	0.0002	608.37	696.52	1.38

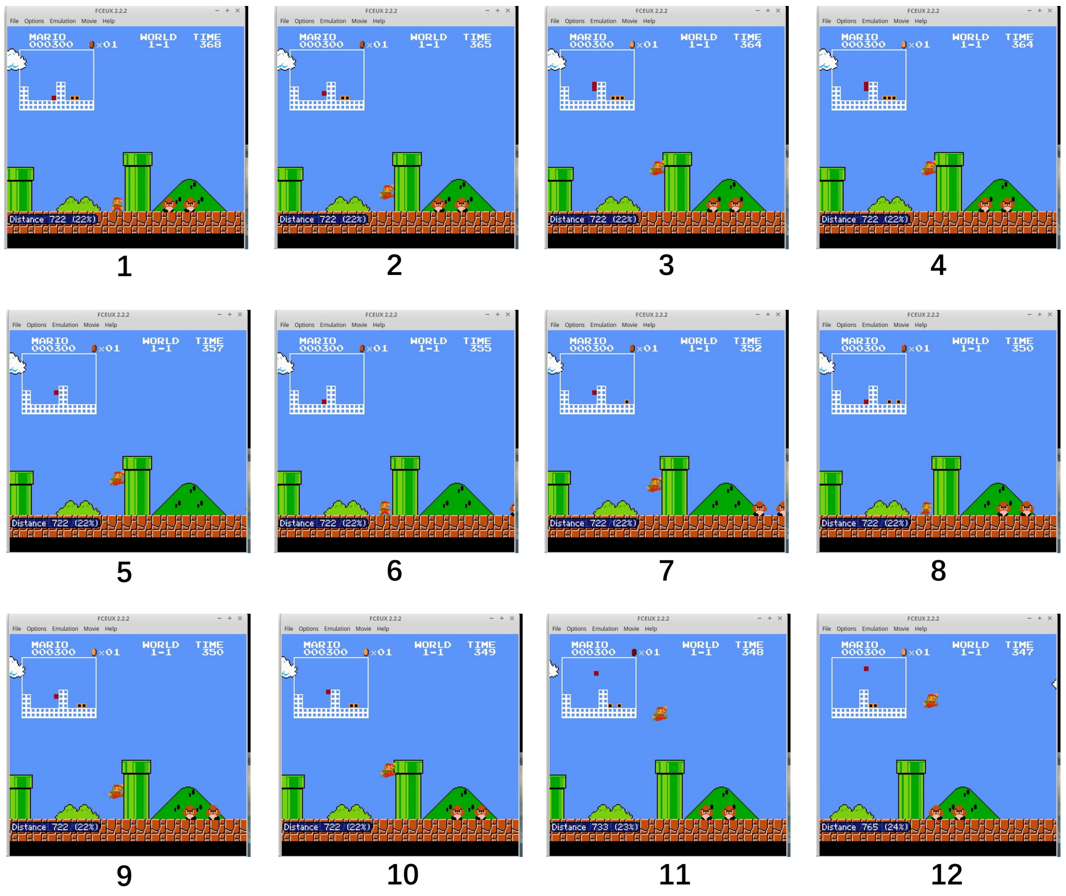


Figure 6: Without training

in the next paragraph. Moreover, we observe a fairly good performance from pure random strategy, and we speculate that since we only have a very small action space $\mathcal{A} = \{1, 2, 3, 4\}$, it is possible that simply acting randomly would result in good performance, since the overall goal is to go right and jump to avoid obstacles. In the future, it might be good to have a larger action space (e.g., add going left) for reinforcement learning.

One interesting phenomena noticed during training is that, on average Mario performed better on jumping over obstacles than avoiding or attacking enemies. Without much learning, Mario always gets stuck at obstacles. Especially at high obstacles, Mario spends a lot of time doing vertical samll-scale jumps, and fall down to the ground again. That is because at the start of training, the model has a high probability of selecting action randomly in order to explore a more variety of states, and going over a high obstacle needs long consecutive JUMP inputs, which is nearly impossible in random strategies. After several thousand epochs, Mario agent is able to take a small jump at small obstacles and take big ones at high obstacles, and those actions are made in a relative short time. However when coming across enemies, Mario performs in a more or less random way, not

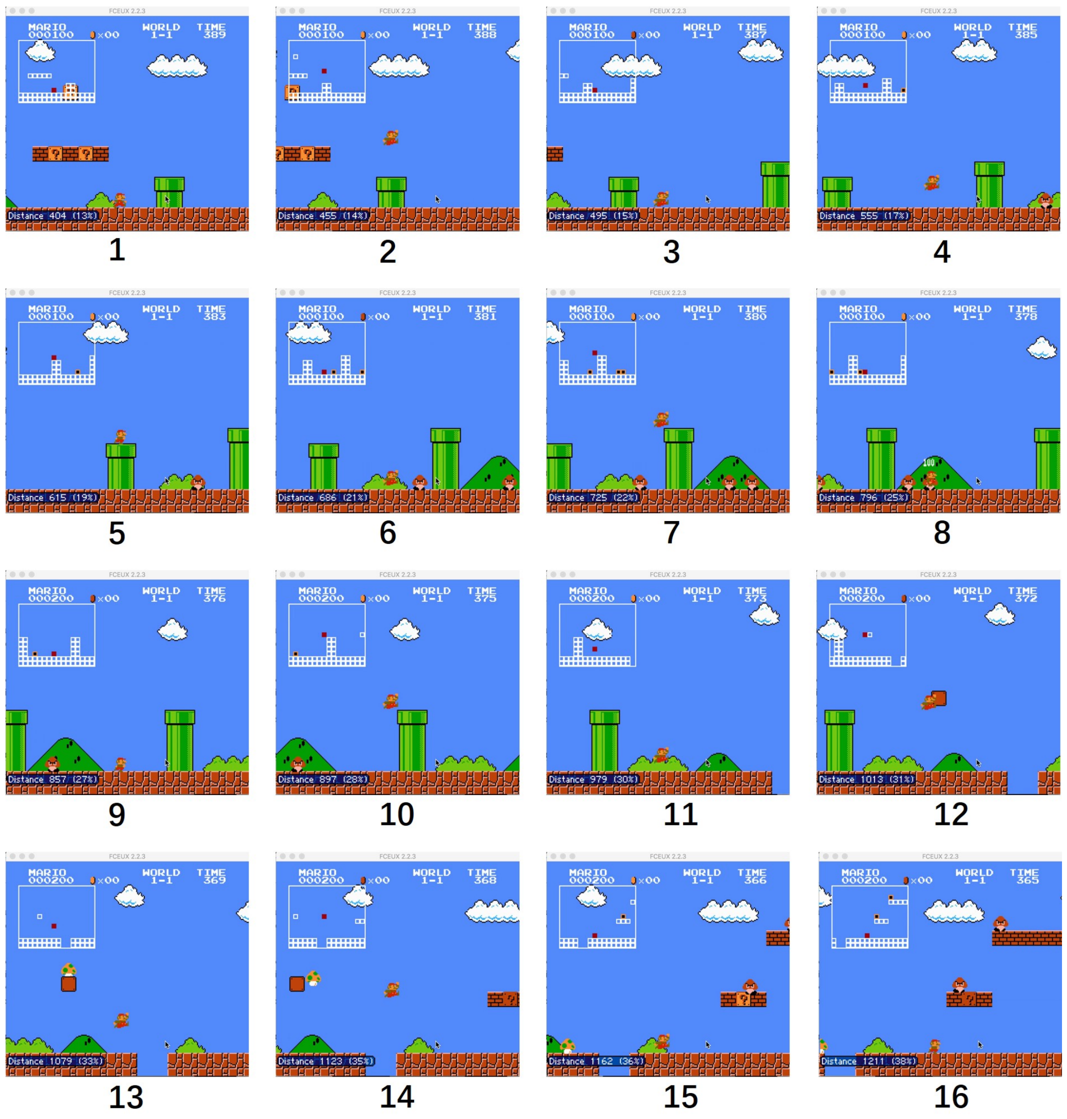


Figure 7: After training

necessarily gaining any knowledge. It is reasonable to suspect that such difference comes from the way we defined reward function.

Figure 6 and figure 7 show typical processes for Mario to avoid obstacles without training (pure random strategy) and after training, respectively. As we can see, before training, Mario stays very close to the obstacle. and keeps doing ineffective small vertical jumps; after training, Mario starts jumping very early and draws a beautiful jump curve over the obstacles. The complete training video of figure 7 is uploaded on YouTube through this link.

5 Challenges and Future Work

The biggest challenge presented in this project is the lack of computation resource. Yue lab generously offered us AWS account. As explained before, our program plays Mario through FCEUX emulator, which requires GUI to operate. AWS doesn't support GUI, and the only solution is to set up a virtual GUI(xQuartz). It turns out to be 10 times slower than local machine, however, possibly caused by the indirect access. Right now on MacPro with 2.6 GHz Intel Core i7, we are able to train around

4000 epochs per day. We would continue training and improving the algorithms, and update for any progress.

In this project, we use a Tile state representation, extracted from the Regular RGB version. It is out of the consideration of computation efficiency. While this 13×16 array is able to correctly extract the majority of useful information, such level of abstraction is not feasible in reality. The next step is to go back to RGB state representation, and teach Mario to play Mario in a real world, comparable to human players.

Also, we can enforce some potential improvements on model design, including (1) more complex network structure (e.g., more convolution or linear layers, padding, etc) for detail extraction; (2) larger past data input, say, instead of feeding the last eight frames, we can do sixteen frames, and so on; (3) better-designed reward functions, which are a byproduct of comprehensive hyper-parameter testing if time permits.

References

[1] Volodymyr Mnih & Demis Hassabis (2015) Human-level control through deep reinforcement learning, *Nature 518*, 529–533.

[2] Volodymyr Mnih & Martin Riedmiller (2013) Playing Atari with Deep Reinforcement Learning, *NIPS Deep Learning Workshop*.

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

[4] Hado van Hasselt & David Silver (2015) Deep Reinforcement Learning with Double Q-learning: a very effective trick to improve performance of deep Q-learning, *AAAI 2016*.

[5] Timothy P. Lillicrap & Daan Wierstra (2016) Continuous control with deep reinforcement learning, *ICLR*.

[6] ppaquette_gym_super_mario,`https://github.com/ppaquette/gym-super-mario`

[7] Playing Mario with Deep Reinforcement Learning, `https://github.com/aleju/mario-ai`