REINFORCEMENT LEARNING FOR SUPER MARIO BROS

Duong Nguyen Xuan [*1] **Trung Nguyen Tran** [*1 2] **Nga Dao Thi Thu** [*1 2]

Abstract

For our final project, we will make an automatic agent using reinforcement learning algorithms to play game Super Mario Bros (SMB). SMB is a side-scrolling 2D video game with a huge state space. There are 32 levels in SMB. Within a game stage, Mario starts on the left of the map and have to overcome many obstacles and enemies. The goal state is getting the flag on the right map. Collecting the coin and defeating enemies in the map are optional. Our project was based on two sources. The first is OpenAI gym toolkit (**?**), a toolkit for developing and comparing reinforcement learning algorithms. The second is gym-super-mario-bros (**?**), an OpenAI Gym environment for Super Mario Bros on The Nintendo Entertainment System using the Family Computer Emulator Ultra X emulator. Our goal is solving level 1-1 and evaluating the result of these gamestages in detailed. We will focus on implementing and optimizing two reinforcement learning algorithms to SMB. We begin with Q-learning, which learns an action-utility function, or Q-function, giving the expected utility of taking a given action in a state. Another algorithm is approximate Q-learning, this algorithm is a combination of Q-learning and an approximate space function. Moreover, we tried to add more variants of reinforcement learning algorithms such as Deep Double Q-learning to our project.

1. Introduction

Applying Machine Learning algorithms to play videos game has been discussed for a long time. In Machine Learning, there are three ways to approach: Supervised Learning, Unsupervised Learning and Reinforcement Learning. Reinforcement Learning is a type of Machine Learning technique that makes an agent able to learn to decide what to do in a particular environment. Game is an interesting environment for Reinforcement Learning. Using Reinforcement Learning or Artificial Intelligence to play game has been researched since the the first examples of AI was the game of Nim made in 1951 and published in 1952. In 2016, Google Deepmind released AlphaGo to prove the power of AI. This is the first AI program can beat the top player in Go game. Games environment is also an important aspect to be considered. In games, there are players characters and non-players character(NPC). About NPCs, They can interact with environment and players automatically. They are almost enemies of player characters to make the game more difficult or they can be an ally to players characters to help them reach the goal of games. We can understand that is some kind of AI to make the games more interesting. But in almost games, NPC is set up to follow only one plan. It means that the NPC will do the same things in every game. There is still another thing we pay attention to, that is player character. The players characters are these characters controlled by player. t In Reinforcement Learning, We try to make our characters play games automatically by trial and error. They can learn to play game, interact with environment by themselves.

1.1. Game Mechanics

Implementing Reinforcement Learning(RL) in Mario game is also a challenges. This game is a classic platform video released by Nintendo Entertainment System (NES). This was a bestselling games of all time with over 40 millions physical copies. The game divined into several world and stage. In each stage, we will control the Mario Character, overcome many obstacles, defeat enemies.We can also collect coins, and flowers. Mario has 3 forms: small, big and fire. When Mario is small, just one hits by enemy that can take Mario life. If Mario is in Big forms, there is one hit to make Mario become small. And final form gives Mario a new ability shooting enemy and one hit by enemy to become Big form. To implement RL in game, We face with the huge stage space of game. We have 6 buttons to control Mario. So we can count that the action space equal to 2^6.That means with any of 6 buttons we press it or not With that problems we consider to setup some different RL algorithms to solve it.

2. BACKGROUND

There are different ways to approach RL. The most typical model in RL is Markov Decision Process(MDP). MDP model is consist of:

- A sets of States (s)
- A sets of Actions in each state (a)
- A transition model $P(s'|s,a)$
- A reward function $R(s)$

Transition model is a probability function describes the outcome of each action in each state. Here, the outcome is stochastic, so we write $P(s'|s,a)$ to denote the probability of reaching state s if action a is done in state s. The next component is reward function $R(s)$. It is a function that returns the reward received of the agent for taking action a in state s. At any state, the action of the agent should do is a solution. This kind of solution is called a policy. It is traditional to denote a policy by π, $\pi(s)$ is the recommend action for the state s.

In MDP, our goal is to find the optimal policy which is called π^*. The quantity of a policy is measured by that expected utility. When we try to maximize the expected utility of the current state that means we are finding the optimal policy. Optimal policy defined by:

$$\pi* = argmax_a \quad (\sum_{s'} P(s'|s,a)U(s')$$

The performance of the agent was measured by a sum of rewards for states visited. There are two different ways to compute the utility of sequences:

- Adaptive rewards:
  $U_h(s_0, s_1, ..., s_n) = R(s_0) + R(s_1) + R(s_2) + ...$
- Discounted reward:
  $U_h(s_0, s_1, ..., s_n) = \gamma R(s_0) + \gamma^2 R(s_1) + \gamma^3 R(s_2) + \dots$

γ is a number between 0 and 1. γ informs the preference of an agent for current reward and future reward. The closer to zero of γ, the more insignificant of the future reward to be. When γ equal to 1, it is the special case of discounted reward. It is equivalent to adaptive rewards. To calculate the optimal policy, we use value iterations algorithms. This algorithm follow the basic idea that calculate the utility of each state and choose the optimal action based on the state utility. Before we go to value iterations, we look through the core of that algorithms – Bellman equations. The Bellman equation show the relations between current state utility and the next state utility. The utility of the current state is equal to the immediate reward plus to the expected discounted utility of the next state when choosing the optimal action:

$$U(s) = R(s) + \gamma max_{a \in A(s)} (\sum_s' P(s'|s,a)U(s')$$

The number of equations is equals to the number of possibles state. To solve this equations, linear algebra is impossible. In equations, there is max operations to maximize the utility of the next state when choosing action a. But max operator is not a linear algebra. To approach this equations, there is a different ways that is using iterations. The iteration step is called Bellman update:

$$U_{i+1}(s) = R(s) + \gamma max_{a \in A(s)} (\sum_s' P(s'|s,a)U(s')$$

The $U_i(s)$ is the utility value for the state s at the i_{th} iterations

Value iterations pseudocode:

Algorithm 1 Value Iterations

VALUE-ITERATION (mdp, ϵ) mdp, an MDP with states S, actions $A(s)$, transitions model $P(s'|s,a)$, reward $R(s)$, discount γ, ϵ, the maximum error allowed in the utility
Local variable U, U' vectors of utilities for states S, initially zero. δ the maximum change in the utility of any state in an iteration
$\delta < \epsilon(1-\gamma)/\gamma$ $U \leftarrow U'; \delta \leftarrow 0$
each state s in S $U'[s] \leftarrow R(s) + \gamma max_{a \in A(s)} \sum P(s'|s,a)U[s']$ $U'[s] - U[s] > \delta\delta \leftarrow U'[s] - U[s]$
U

3. RELATED WORK

For the past many years, Mario was an interesting topic in AI community . In 2009, there was a Mario AI competition (**?**) between AI researchers. The competition was held in association with the IEEE Consumer Electronics Society Games Innovation Conference 2009 and with the IEEE Symposium on Computational Intelligence and Games. This competition lasted 4 years till 2013. While the competitors try to implement variety of Machine Learning techniques, Reinforcement Learning, the top team uses the A* algorithms to solve the game This competition has no longer held, but many useful sources was explored. Our project is based on this sources of the competition and developed on that.

4. ALGORITHMS

4.1. Q-learning

Q-learning algorithms (**?**) was proposed by CHRISTOPHER J.C.H. WATKINS in 1989. It is a form of model-free Reinforcement Learning. Q–learning initialize with a Q–table which is contain pair of $(state, action)$. In this algorithms, the agent will not learn the utility of the policy instead it will learn the utility the action chosen at the state. For each action in a particular state, Q-value is updated the utility of action - Q*(s,a) based on the temporal difference following the rule:

In update rule, Q(s,a) is defined the Q-value of the action a

at the state s in the k_{th} iteration. α is defined as the learning rate. γ is discount factor, it control the short term and long term reward.

To make the Q-learning work faster, there is the trade-off between the exploitation and exploration of the agent. At every iterations, the agent can choose only one action. But in the update rule, the agent will lose the exploration if it always chooses the $maxQ(s, a)$. To balance this term, Adding the $\epsilon - greedy$ is a solution.

This will make our agent quite faster to converge when updating the Q-table. Q-learning algorithms pseudocode:

Algorithm 2 Q-learning

Q-LEARNING (mdp, ϵ)
$percept$, a percept indicating the current state s' and reward signal r' an action **persistent** Q, a table of action values indexed by state and action, initially zero. N_{sa}, a table of frequencies for state-action pair, initially zero. s, a, r the previous state, action, and reward, initially null
TERMINAL?(s)$Q[s, None] \leftarrow r'$ s is not nullincrement $N_{sa}[s, a]$
$Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])(r + \gamma max_{a'} Q[s', a'] - Q[s, a])$
$s, a, r \leftarrow s', argmax_{a'} f(Q[s', a', N_{sa}]s', a'), r'$
a

4.2. Approximate Q-Learning

Due with the large state space of Mario game, Q–learning seems to be impossible. We need to find another way to reduce the number state space of game so we want to make sure that the algorithm always converge. We will try to implement another variant RL algorithm that is Q-learning with approximate linear function. We engineering a sets of features to approximate each states. These features represent the linear Q–function:

$$Q(s, a) = w_1 f_1(s, a) + \ldots + w_n f_n(s, a)$$

Approximate Q-learning algorithm pseudocode (**?**):

4.3. DOUBLE Q-LEARNING

In Q – learning, The max operator in update rule makes this algorithms becomes overestimate. We can see that in this example below:

Assuming that we at state X, there are 6 actions to take and the number of them is the value of these actions. The probability of choosing these action is equal, so we can calculate the expected value of the state X equals to $1/6 * (1+2+3+4+5+6) = 3.5$. But When we run many time with max operators in update rule, suppose we choose the action with reward 6. So that action will be chosen every time but it is higher than the actual value of that state action.

Algorithm 3 Approximate Q-learning

APPROXIMATE Q-LEARNING (mdp, ϵ) $percept$, a percept indicating the current state s' and reward signal r' an action **persistent** g, exploration function.
ϵ, the probability of a random move
F, a set of n feature functions that return a feature value for a state action pair
W', a set of n weights for each feature function initially all zero N_{sa}, a table of frequencies for state-action pairs, initially zero
s, a, rW, the previous state, action, reward and weights, initially null
TERMINAL?(s)$max_{a'} W.F(s', a') \leftarrow r'$ s is not nullincrement $N_{sa}[s, a]$
i = 1 to $nW'[i] \leftarrow W'[i] + \alpha(r + \gamma max_{a'} W.F(s', a') - W.F(s, a))F[i](s, a)$
$s, a, r, W \leftarrow s', g(\epsilon_i, argmax_{a'} Q[s', a'] + 10/(N_{sa}[s', a'] + 1), r', W'$
a

This makes algorithms is biased and more slow to learn. To solve this, Hado Van Hassle proposed a algorithms called Double Q-learning (**?**) to void this overestimate. Double Q – learning works with two Q functions. Each Q functions is updated Q-values based on the other Q – functions for the next state.

Algorithm 4 Double Q-Learning

InputInput OutputOutput
DOUBLE Q-LEARNING **Initialize**Q^A, Q^B, s

end Choose a, based on $Q^A(s,)$ and $Q^B(s,)$, observer r, s'

Choose (e.g. random) either UPDATE(A) or UPDATE(B)

$UPDATE(A)$ Define $a^* = argmax_a Q^A(s', a)$

$Q^A(s, a) \leftarrow Q^A(s, a) + \alpha(s, a)(r + \gamma Q^B(s', a*) - Q^A(s, a)$ Define $b^* = argmax_a Q^B(s', a)$

$Q^B(s, a) \leftarrow Q^B(s, a) + \alpha(s, a)(r + \gamma Q^A(s', a*) - Q^B(s, a)$

4.4. Deep Double Q-learning

Begin with the Q-learning algorithm, by using Bellman equation, we have value-action function:

$$Q_{i+1}(s, a) = [r + \gamma a' max Q^*(s', a'|s, a)]$$

We add variable ϕ to estimate the value-action function, $Q(s, a, \phi) \approx Q^*(s, a)$. In our implement, we use neural network function (NN) with weight ϕ.

Finally, we apply the gradient descent algorithm:

$\phi \leftarrow \phi - \alpha \frac{\delta Q(s_i, a_i, \phi)}{\delta \phi}(Q(s_i, a_i) - [r(s_i, a_i) + \gamma a' maxQ(s'_i, a'_i, \phi)])$

Algorithm 5 Deep Double Q-learning

Initialize policy network $Q(s, a; \phi)$ with random weights
Initialize policy network $Q(s, a; \phi')$ with random weights
True $\phi' \leftarrow \phi$
episode = 1 ... K
Depend on ϵ select a random action or select $aargmaxQ(s, a; \phi)$
Observe s, a, s', r
Gradient descent: $\phi \leftarrow \phi - \alpha \frac{\delta Q(s,a,\phi)}{\delta \phi}(Q(s, a_i) - [r(s, a) + \gamma Q(s', a' argmaxQ(s', a', \phi); \phi')]$

5. Experiment

5.1. Environment

To run SMB, we use Family Computer Emulator Ultra X (FCEUX) and Gym-Super-Mario-Bros. FCEUX is emulator of the original (8bits) NES. Before FCEUX was released, there had developed many version based on FCE Ultra. Based on that development, FCEUX is combined of all the FCE previous forks like FCE Ultra, FCEU-MM, FCEU Rerecording and released on August 2,2008. FCEUX is "all in one" emulator for both playing game normally and a lot of advanced emulator functions because of that combination above.

Gym-Super-Mario-Bros is an OpenAI environment. It gives us 3 attemps to get through 32 levels in SMB. In each step, we generalize the state of the step as below:

5.1.1. State

There are two kinds of states in SMB: Regular state and tile state:

- Regular state returns a color image with 256 x 224 pixels, where each pixel has 3 channels: red, blue and green (RGB) value. This generalization returns all information what we see on the screen.

- Tile state is a generalized state. The game screen is divided in to a grid. We can represent character, enemies, and obstacles in on that grids. Tile state is return a 13x16 array. In each element of array we consider the values of it: 0 represents an Empty; 1 represents an Object; 2 represents an Enemy; 3 represents Mario. By this generalization, we can get the most important information from a complicated image.

5.1.2. Reward

The reward function is the consideration how far the Mario can move (increase the agent's horizontal value from the last frame to the current frame), as fast as possible, without losing life.

5.1.3. Done

Done is a Boolean value. It indicates that the game is ended or not. Done is default set to false. If the Mario dies or overcome the level, Done will be set to True.

5.1.4. Other information

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

Key	Type	$\log_2 fraction1, p_i$
coins	int	The number of collected coins
flag_get	bool	True if Mario reached a flag or ax
life	int	The number of lives left, i.e., 3, 2, 1
score	int	The cumulative in-game score
stage	int	The current stage, i.e., 1, ..., 4
status	str	Mario's status, i.e.,
time	int	The time left on the clock
world	int	The current world, i.e., 1, ..., 8
x_pos	int	Mario's x position (from the left)
y_pos	int	Mario's y position (from the bottom)

5.2. Modeling

5.2.1. Action

The action space is based on the Nintendo controller, which has 6 buttons (Up, Down, Left, Right, Jump, Attack) that either pressed (1) or not (0). This would result in $2^6 = 64$ possible actions. Nevertheless, only 14 of these combinations make logical sense, and only 9 of the 14 have an effect on the game. For example, the combination left and right does nothing, as Mario simply remains in place. But the number of statements very large and we just want to achieve some simple worlds such as World 1-1 , therefore we use only a set of 4 actions:

actions = { Empty action, Right, Jump, Right + Jump }

5.2.2. State

For simple, we decided to use tile state generalization. For Deep Q-Learning, it is simple to create a fundamental neural network with a few layers. But we have had big problems when we implement Q-Learning algorithm. A back-of-the-envelope calculation reveals that, with 4 possible values for each tile (0-3), 13×16 tiles, and 4 possible actions, the number of state-action pairs is extremely high: $4^{13 \times 16} \times$

$4 \approx 6.77 \times 10^{125}$. In practice, we found empirically that Mario only deals with about $100000 - 300000$ unique states. It seem be possible to store state-action. For reduce the memory, we hash the state-action states before store it in a dict in $Python$. But it will increase the complexity time. The training time each epsilon will be slow because of that.

5.3. Approach

5.3.1. Q-Learning and Approximate Q-Learning

It the both of algorithms, we set similar parameters

- α (learning rate): 0.1 for Exact Q-Learning and 0.01 for Approximate Q-Learning
- γ (discount factor): 0.95 for both of them
- ϵ (minimum random move probability): 0.05 for both of them
- λ (eligibility trace decay): 0.8 for Approximate Q-Learning, this parameter doesn't use in Q-Learning

5.3.2. Deep Double Q-learning

The tile state generalization returns an array with size 13×16 each frame. We decided using 4 continuous frames because we want to track the movement of Mario and armies. The size of a state is a matrix with size $4 \times 13 \times 16$ is We use some transformations to present a state into a vector of tensor in figure 1.

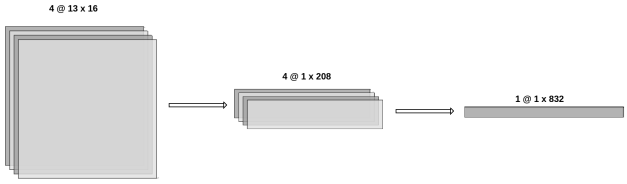

Figure 1. Pre-processing

After the transformation, the size of input layer is $4 \times 13 \times 16$ tensor to present the a state, and the output is one of four actions since from our space actions. We designed a simple linear neural network as you can see in the figure 2.

6. Result

After implementation, we realize that we can not disable the GUI of FCEUX. So it makes training process slow very much. The computers which used to train require the display so we can not train in a server computer. We must train in our personal computer. Therefore we can not run the

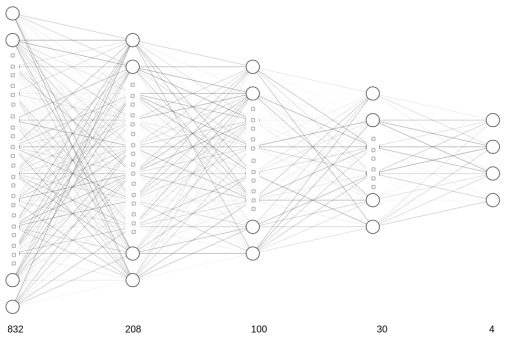

Figure 2. Neural Network

training with a hug number of iterator as we believe. Most of algorithms are difficult to get over the checkpoint, at distance 1400. The training process takes about 30 hours in total.

6.1. Q-Learning

As you can see in figure 3, the learned states increase more and more over $100,000$ iterators and so on, our personal computer was crashed because the memory ran out.

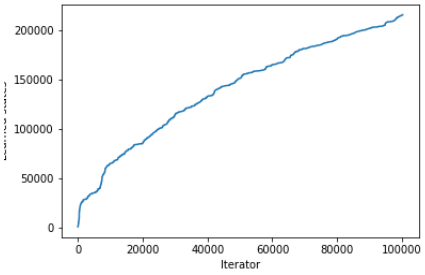

Figure 3. Number of learned states over iterators - Q-Learning

As you can see in figure 4, It can reach to the end in several times, about $16,26\%$, but not stable.

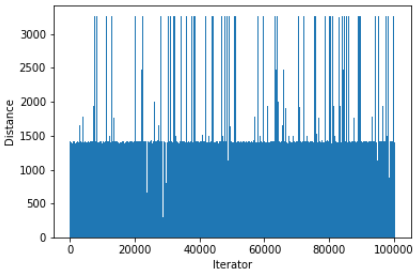

Figure 4. Distance over iterators - Q-Learning

6.2. Approximate Q-Learning

As you can see in figure 5, the learned states increase so fast at the beginning, but after about $32,000$ iterators, it is difficult to learn new state. I just tried to tuning the parameter θ to increase the exploration, but it doesn't affect too much. The Mario seem more careful in each step.

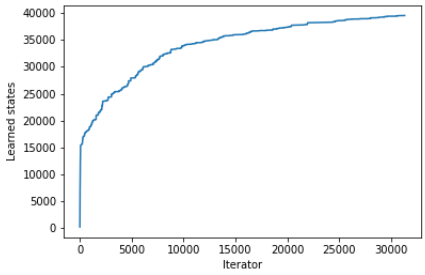

Figure 5. Number of states learned over iterators - Approximate Q-Learning

As you can see in figure 6, most of time, Approximate Q-Learning is stuck at 1400 distance checkpoint. It can not win the game anytime.

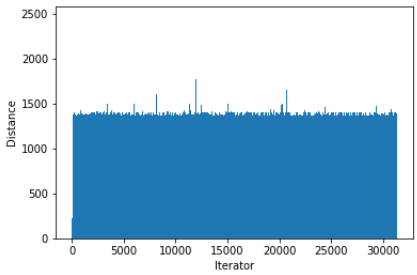

Figure 6. Distance over iterators - Approximate Q-Learning

6.3. Deep Double Q-learning

We try to reduce the size of statement small as possible as we can ans use GPU to speedup training. But as we mentioned before, we can not disable the interface, it's slow very much. The model is difficult to converge unless we can reach to a huge of epoch. As you can see in figure 7, after about $32,000$ iterator, the Mario still stuck in the checkpoint,

7. Discussion

We think that tuning hyperparameters is important, which decides for an algorithm whether works well or not. It can reduce the memory, the complexity, and so on. Q-learning is a simplest algorithm, we just tuning a few hyperparameters. Basically, Q-learning just run and learn. But it converges

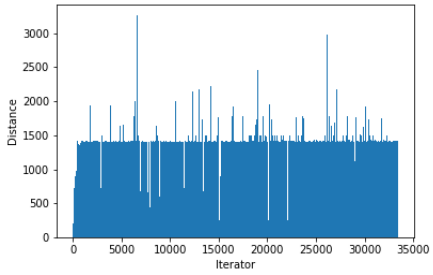

Figure 7. Distance over iterators - Deep Double Q-learning

slowly. After a lot of iterators, it can not converge to achieve the goal. It's still learn new state until we ran out of memory. Approximate Q-Learning, which has a lot of hyperparameters. We use must of time to tuning some primary hyperparameters because the Mario's behavior seem excellent, so we believed that the Mario can get over if we tuning well. But after all, we could not find optimal hyperparameters for approximate function to pass the level 1-1. And the last algorithm we tried is Deep Double Q-learning, this is a amazing algorithm. We had thought about it very much before we implemented: How can we reduce the memory but remain the important information? How can we design a neural network? It's necessary for use convolution layer in the network? Because we could not run it until it achieve the goal. So it may be a potential algorithm which we must use more time for experience.

Implementing RL for Mario game is really a interesting challenge. We can see the improvement of the agent step by step directly. By trying many variant RL algorithms , we can see the performance these algorithms and have a overview about advantages and disadvantages of each algorithm. The lack of time and hardware sources lead algorithms performance not as we expected. In addition, we have learned and explored interesting experience about RL.