# Robot Learning - Applying Reinforcement Learning to Super Mario Bros.

Chengli Liu
*Robot Learning*
*Universitat Politècnica de Catalunya, UPC*
Barcelona, Spain
chengli.liu@estudiantat.upc.edu

Benjamin Harun Tezcan
*Robot Learning*
*Universitat Politècnica de Catalunya, UPC*
Barcelona, Spain
benjamin.harun.tezcan@estudiantat.upc.edu

Zhaoqi Li
*Robot Learning*
*Universitat Politècnica de Catalunya, UPC*
Barcelona, Spain
zhaoqi.li@estudiantat.upc.edu

***Abstract*—In reinforcement learning, an agent tries to make certain decisions in a complex environment. Apart from robotics, this method is often applied to video games such as Atari and Super Mario Bros. In this document, a double deep Q-learning algorithm is implemented in order to complete the first level of Super Mario Bros. Its performance is also compared to a normal deep Q-learning algorithm.**

***Index Terms*—Video Games , Markov Decision Process, Convolutional Neural Network, Double Deep Q-Learning.**

## I. Introduction

The goal of this project is to develop a reinforcement learning algorithm to play the first level of Super Mario Bros. This algorithm is based on double deep Q-learning which uses a Convolutional Neural Network to map states in form of images to appropriate actions.

### A. Software and Interface

To develop this project the gym-super-mario-bros library from OpenAI gym is used. gym is a famous Python library that provides a set of environments for reinforcement learning. In order to be able to interface with the game, an emulator is needed. In this case, the Nes-py emulator by Nintendo Entertainment System (NES) is used. Nes-py is a framework to interface between NES game environments and Python applications. Due to the need for a powerful GPU, Google Colab is used for the code implementation.

### B. Document Structure

This document is structured as follows: A concise state of the art is presented in Section II. The corresponding problem definition is elaborated in Section III in form of a Markov Decision Process. In order to solve this problem double deep Q-learning is applied that is described in Section IV. Finally, in Section V some experiments and their results will be examined. In addition, the performance of the DDQN will be compared to a normal DQN. Also encountered problems will be explained that occurred during the implementation.

## II. State of the Art

### A. Reinforcement Learning in Video Games

Reinforcement learning has already been applied to various video games. Probably one of the most popular occasions in this field was when DeepMind, now a Google company, created a policy-based Deep Neural Network called AlphaGo that beat Mr Lee Sedol, a legendary Go player, 4-1 in 2016. Other applications of reinforcement learning are for example Atari and even board games such as chess and sudoku.

### B. Approaches to Super Mario Bros.

Several different approaches have been applied to Super Mario Bros. In [1] members of Stanford University play Super Mario with a simple Q-Learning algorithm that accomplishes a win rate of 90% with about 5000 episodes. Other machine learning engineers deal also with deep Q-Learning and double deep Q-Networks (DDQN) as described in [2].

Q-values are tried to be approximated during training, either by simple exploration of the environment or by using a function approximator, such as a deep neural network (Deep Q-Learning: DQN). Mostly, the action that has the highest Q-value for each state is selected.

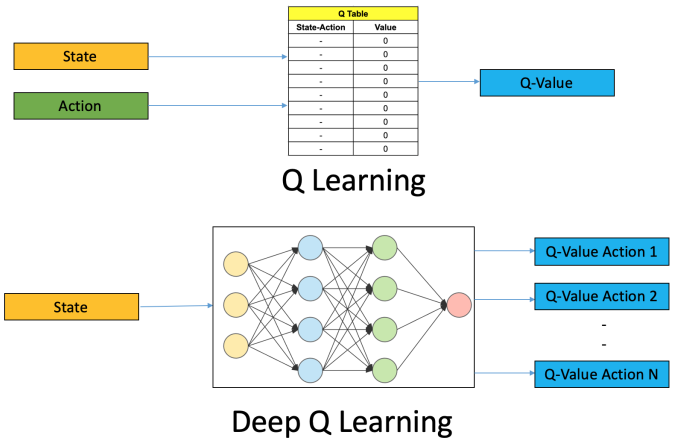

Fig. 1: Q-learning vs. Deep Q-learning

An issue of Q-learning could be that the model produces extraordinary high Q-values during training sometimes due to the selection of the maximum Q-value for determining the best action. Thus, the model might run into a locally optimal solution or select the same actions all the time. This problem can be solved by double Q-Learning.

In double Q-Learning the selection process of the actions is not only determined by choosing the action with the highest Q-value. There are two deep neural networks involved, the local and the target net. The target net is a copy of the local net with frozen weights that are not directly trained. The selection process consists of three steps. First, the target network computes the target Q-values of the state after taking the action. Second, the local net computes the Q-values of the state after taking the action and selects the best action by finding the maximum Q-value. Then, the target Q-values are calculated using the target Q-values of the target net, but at the selected action indices of the local net. This process assures that an overestimation of Q-values does not occur because they are not updated based on themselves.

## III. Problem Definition

The problem to be solved is described as a Markov Decision Process.

### A. Agent and Environment

First, the agent and the environment are defined. The agent is Mario and the environment where the agent is moving is the first game level.

### B. Actions

In this case, the actions are predefined by OpenAI gym. In order to make the training process simpler the RIGHT_ONLY movement package is chosen. This package includes five actions: Standing, Right, Right A, Right B, Right A+B. A and B refer to the controls of the NES controller.

### C. States

Furthermore, the possible states are determined. A state is a list of 4 contiguous 84x84 pixels frames. Initially, the observation space shape is 240 × 256 × 3. 240 stands for the height and 256 for the width, 3 represents the 3 color channels. gym offers several ways to change the default settings of the environment in order to make the agent learn faster, these are called wrappers. One wrapper is used to transform the observation shape into 84 x 84 x 1 frames. More wrappers are applied to the environment which are presented in Section IV.

### D. Reward

The reward is predefined by the gym. Moving forward yields a positive reward for Mario whereas a penalty is given for dying, standing still and moving left. The rewards are more specified in [3].

### E. Learning Rate and Discount Factor

The learning rate is represented by $\alpha$. Setting the $\alpha$ value to 0 means that the Q-values are never updated, thus nothing is learned. If we set this value to a high value, it means that the learning can occur quickly. The discount factor $\gamma$ quantifies how much importance is given to future rewards. If this value is closer to 0, the agent will tend to consider only immediate rewards. If it is closer to 1, the agent will consider future rewards with greater weight, willing to delay the reward. $\alpha$ and $\gamma$ usually adopt values between 0 and 1. In this implementation, several values for both factors are tried out in order to see the difference.

The above defined problem components are summarized in Table I.

TABLE I: Problem Definition

| | |
|---|---|
| **Agent:** | Mario |
| **Environment:** | First game level of Super Mario Bros. |
| **Actions:** | RIGHT_ONLY movement: 5 actions |
| **States:** | A state is a list of 4<br>contiguous 84x84 pixels frames |
| **Reward:** | redefined rewards by gym-super-mario-bros<br>Positive reward: Moving forward<br>Penalty: Dying, standing still and moving left |
| **Learning rate $\alpha$:** | Several values are tried out |
| **Discount factor $\gamma$:** | Several values are tried out |

## IV. Proposed Approach

### A. Double Deep Q-learning

The proposed approach to solve the problem is a double deep Q-learning algorithm. As described in Section II, the double deep Q-learning consists of two different networks, the local and the target net. The local net considers the Q-values that are updated and the target net deals with the Q-table which is used for taking actions. Thereby, the overestimation bias of the Q-values is avoided. As a second algorithm normal deep Q-learning is introduced. Both algorithms are involved in the experimentation section to observe the differences between them.

In this project, the DQN consists of three convolutional layers and two linear layers. The net follows an epsilon-greedy policy.

A DQN is used because the amount of required time and memory would be too high for simple Q-learning due to the huge state pace of the game. This is why DQNs are often used for video games. As input, states in form of preprocessed game screens are used. The provided input shape is 4×84×84, and there are 5 actions. As output, the Q-values for every action are estimated.

### B. Environment Wrappers

In order to make the agent learn faster, several wrappers are applied. These wrappers transform the environment in different ways. The five applied wrappers are described below:

*1) MaxAndSkipFrameWrapper:* The agent makes the same action over 4 frames.

*2) FrameDownSampleWrapper:* The original {240,256,3} gets downsampled to {84,84,1}. This wrapper switches from RGB to grayscale.

*3) ImageToPyTorchWrapper:* It converts the frames into PyTorch tensors {1,84,84}.

*4) FrameBufferWrapper:* Only every fourth frame is collected.

*5) NormalizeFloats:* This wrapper normalizes the pixels from [0,255] to [0,1].

### C. Experience Replay

In addition, a method called experience replay is introduced. Experience replay is the act of gaining experience and sampling from replay memory that stores this experience. An experience is called a tuple of {state, action, reward, next_state, terminal}. It is stored in a buffer to use later in a memory named Replay Memory with the remember method. In the experience_replay method, the agent just has to sample a batch of experiences and update the network weights. To do so, three important functions are needed:

- *remember:* An experience is pushed onto the buffer so that the data can be used for later.
- *recall:* It samples a batch of experiences from memory.
- *experience_replay:* This is the main function of experience replay. This method is what will allow the agent to learn. It contains the local and the target net.

## V. Experimentation and Results

### A. Deep Q-Learning

In this subsection, the deep Q-learning algorithm is applied to the training model, to obtain the average reward after the agent is trained for 1000 episodes. The hyperparameters for this experiment are $\alpha$ = 0.00025 and $\gamma$ = 0.9. The result is shown in Figure 2, the average reward is around 400.

In basic Q-learning, the optimal policy of the agent always chooses the best action in any given state, this assumption is based on that the best action always has the maximum Q value. However, in the experiment, the best action may not have a maximum value in the Q-table, especially in the beginning of the environment. The overestimation will happen when the agent tends to take the non-optimal action in any given state.

To reduce the effect of overestimation, [8] proved if the noises of all Q-values have a uniform distribution, more specifically, the Q-values are equally overestimated, then overestimations are not the problem since these noises do not have an impact on the difference between the Q(s', a) and Q(s, a).

When the DQN result is compared to the DDQN result, which has the same hyperparameters, as shown in Figure 3, it is easy to observe that the DDQN algorithm performs better in this situation.

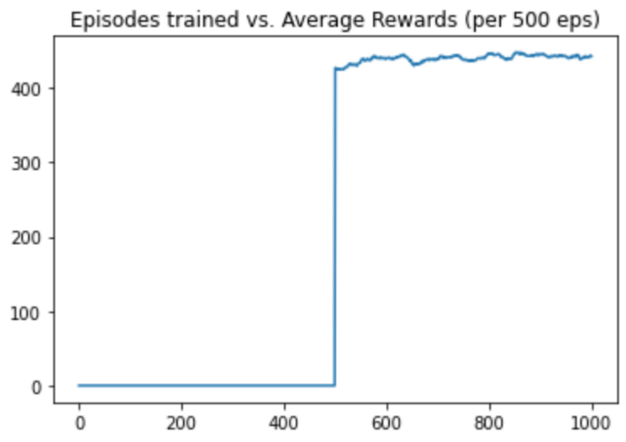

Fig. 2: DQN Result

### B. Double Deep Q-Learning

In this subsection, some experiments applying double deep Q-learning with different parameters are done.

In the first experiment, the agent is trained for 1000 episodes. The hyperparameters are $\alpha$ = 0.00025 and $\gamma$ = 0.9. As it can be seen in Figure 3, the average reward is increasing towards the end. This implies a positive behaviour of the model.

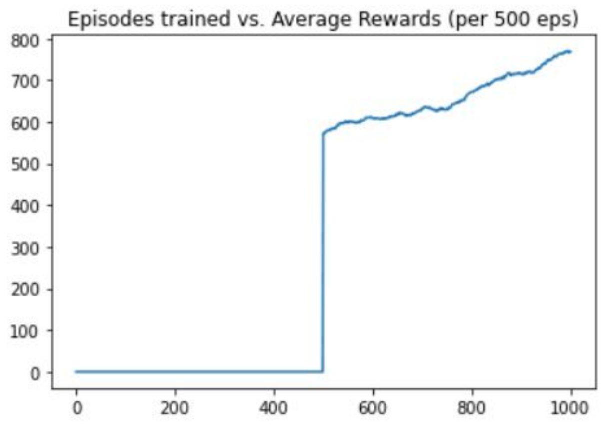

Fig. 3: 1st Experiment

During this experiment some first problems were encountered. The first problem is that Mario has problems when standing in front of big pipes. He has to jump several times until he gets past them (Figure 4).

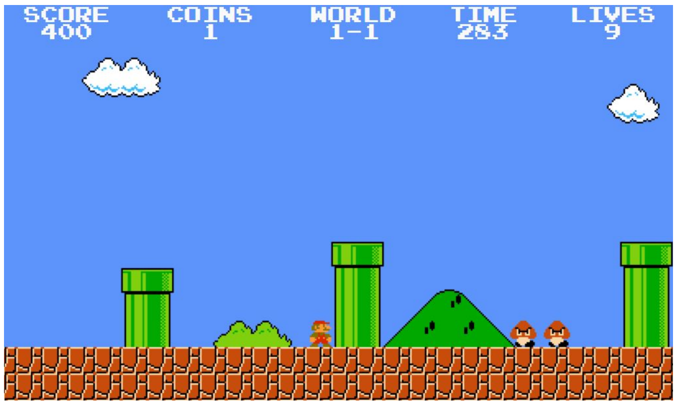

Fig. 4: Mario in front of a big pipe

Another problem could be that the agent needs more episodes in order to train better in the environment. Training the agent for 1000 episodes takes about 4 hours in Google Colab (depends also on thy hyperparameters). The problem is that Google Colab provides a maximum of 12 hours running time. Therefore, a maximum of around 2500 episodes can be executed technically due to the limited resources. In addition, the Colab Notebook might not run 12 hours continuously due to RAM problems and therefore crash. According to our experiences, 1700-1800 episodes often were the maximum. Very few times it could accomplish 2000-2500 episodes.

In the next experiment, 2500 episodes are applied. $\alpha$ is 0.01 and $\gamma$ = 0.9. This experiment needed a running time of 11 hours and 30 minutes. It can be seen in Figure 5 that the initial progress of the reward is quite similar to the first experiment. It starts decreasing at around 1200 episodes and starts increasing again later. It has a very wavy behaviour.

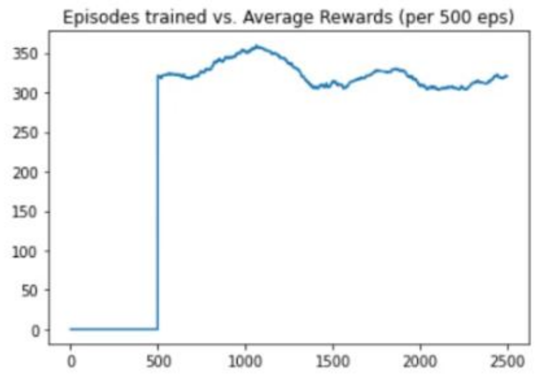

Fig. 5: 2nd Experiment

In the third experiment, the episodes are set back to 1000 and $\alpha$ increases up to 0.35 and $\gamma$ remains at 0.9. By looking at the result in Figure 6, it can be seen that the reward is not really increasing nor decreasing. It is remaining at the same level more or less.

In the fourth experiment, 1000 episodes are applied and

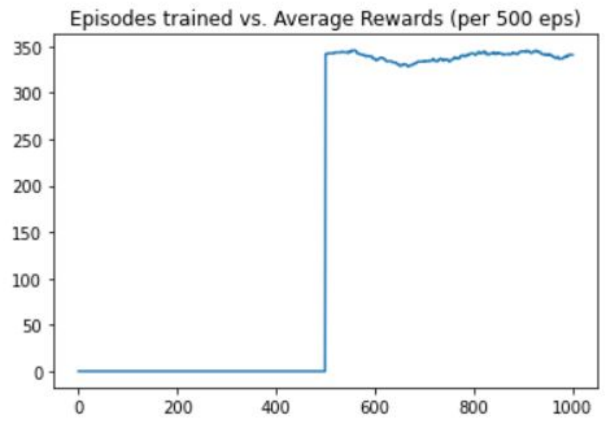

Fig. 6: 3rd Experiment

$\alpha$ is 0.35 again. This time, $\gamma$ is set to 0.5. According to the results in Figure 7 the behavior is very similar to the previous experiment. A significant difference is the running time which has almost doubled. The duration was 5 hours and 30 minutes. It can be concluded that decreasing gamma does not influence the performance in a positive way.

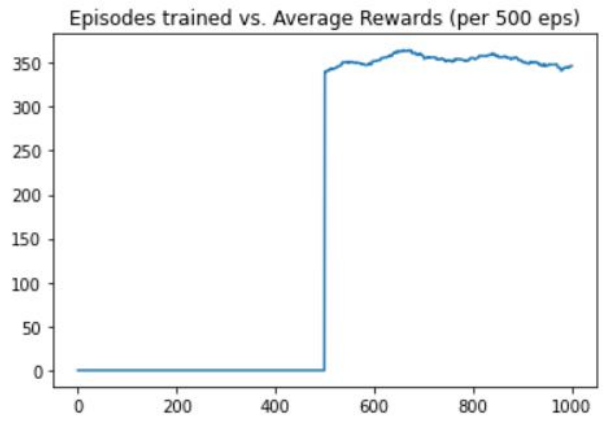

Fig. 7: 4th Experiment

A last experiment was done with 1500 episodes, an $\alpha$ of 0.5 and $\gamma$ was set back to 0.9. In this case, the average reward is even decreasing towards the end, meaning that a high value for $\alpha$ is affecting the performance negatively (Figure 8).

After observing the changes of the single parameters and their resulting influences on the performance of the model, it can be said that a low value for $\alpha$ and a high value for $\gamma$ yield the best results (as shown in the first experiment). A next step would be to apply these values to a training process of 3000, 4000 episodes and so on which would require more powerful hardware.

A video of the results of the first experiment is attached to this report.

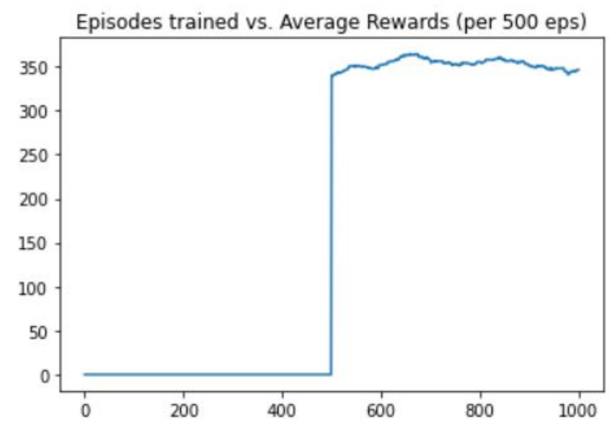

Fig. 8: 5th Experiment

## VI. Conclusions

After comparing the results of the DQN with the DDQN, it can be concluded that the DDQN is better suited for this problem of Super Mario Bros. In Section V it was also shown that changing the training parameters have a significant impact on the performance of the model. In the end, a low value close to 0 for $\alpha$ and a high value close to 1 for $\gamma$ yield promising results.

However, several problems were encountered during the implementation. More episodes are needed in order to solve the problem, so that Mario can finish the level successfully. For that, more running time and also better hardware in terms of GPU and CPU are required.

Another possible solution could be to simplify the environment even more. Unnecessary scenes of the environment could be deleted in order to compress the training process. Future work can also lie in dealing with the problem of the big pipe. Mario could be confronted with this scene over and over again until he developed the appropriate behaviour.

## References

[1] Liao, Y., Yi, K. Yang, Z. (2012). *CS229 Final Report Reinforcement Learning to Play Mario.* Retrieved from http://cs229.stanford.edu/proj2012/LiaoYiYang-RLtoPlayMario.pdf

[2] Heinz, S. (2019, May 19). *Using Reinforcement Learning to play Super Mario Bros on NES using TensorFlow.* Retrieved from https://www.statworx.com/ch/blog/using-reinforcement-learning-to-play-super-mario-bros-on-nes-using-tensorflow/

[3] Kauten, C. (2020, June 12). *gym-super-mario-bros 7.3.2.* Retrieved from https://pypi.org/project/gym-super-mario-bros/

[4] Grebenisan, A. (2020). *Play Super Mario Bros with a Double Deep Q-Network* Retrieved from https://blog.paperspace.com/building-double-deep-q-network-super-mario-bros/

[5] Clark, R. *Mario's Gym Routine* Retrieved from https://towardsdatascience.com/marios-gym-routine-6f095889b207

[6] Sagar, M. (2021, March). *Top 6 Baselines For Reinforcement Learning Algorithms On Games* Retrieved from https://analyticsindiamag.com/reinforcement-learning-top-state-of-the-art-games-alphago/

[7] Schneider, N. *A Simple Guide To Reinforcement Learning With The Super Mario Bros. Environment* Retrieved from https://medium.com/geekculture/a-simple-guide-to-reinforcement-learning-with-the-super-mario-bros-environment-495a13974a54

[8] van Hasselt, H. (2010). *Double Q-learning* Retrieved from https://papers.nips.cc/paper/2010/hash/091d584fced301b442654dd8c23b3fc9-Abstract.html