Using Deep Q-Network and Double Deep Q-Network Reinforcement Learning Algorithms to Complete Levels of Super Mario Bros. on the NES

Kasra Jalali, Alejandro Venegas, Joel Oh, Jason Chung
University of California, Los Angeles
`ven716@ucla.edu, joel.jy.oh@gmail.com, jcchung@ucla.edu, kasraj@ucla.edu`

Abstract

In this project, we aimed to use reinforcement learning to train an agent to complete a level from the game Super Mario Bros. on the NES. We utilized Open AI Gym's implementation of the game to easily create and access the environment. In addition, we preprocessed the state with different methods, and generalized the action space to speed up training. We used two algorithms, approximate Q-learning and approximate double Q-learning, as well as experience replay and fixed target to solve our problem. A neural network was also required to perform approximation due to the size of the state space. We compared the performances of the two methods, which exhibited that Double DQN performs well with a relatively high total reward and win rate whereas DQN doesn't quite work well as it over-estimates Q-values. The trained agent was also shown to not generalize properly as it failed to perform successfully on other levels, with and without transfer learning. Future work would be to try various reinforcement learning methods to allow the agent to learn a more generalizable approach so that it can defeat all levels without human interference.

1 Introduction

A common usage of reinforcement learning is to train a system to complete games. The method is highly efficient in finding the best possible ways to win, often resulting in better performances than humans. An example is AlphaGo, a program trained to play the game Go against human competitors. The trained agent proved to be a success by beating the world champion, Lee Sedol, multiple times. For our project, we wanted to have an agent complete the game Super Mario Bros. on the NES level 1-1. Successful training and completion would indicate that the same method can be utilized for all levels of Super Mario Bros. and the agent can finish the game without any human interference. It also signifies that with changes in the reward function, action space, and other parameters, a similar framework can be applied to other games. Through its success, reinforcement learning can be viewed as a powerful tool to easily beat any game and any records held by humans.

Some challenges would be in creating a comprehensible state space for the framework. The obvious state of the game is the screen, but this state would require high memory and complexity that the agent will take a very long time to train. Thus correctly preprocessing the state is crucial in successful learning. This also leads to the issue of appropriate neural network architecture. With such a large state space, it requires a suitable neural network to perform approximation and pick the correct action. Some other challenges would be generalizing the reward function and the action space so that learning isn't interfered, but also the time to train is reduced. Overall, the reinforcement learning framework has to be applied properly in order for the agent to actually beat the level.

2 Problem Formulation

We will be using the Open AI Gym's environment for Super Mario Bros [1]. It allows easier interaction with the environment and offers simple modifications. Reinforcement learning was applied to different aspects of solving the game as shown below.

Agent	Mario
Environment	Super Mario Bros level 1-1
Observed State	Preprocessed frames from the game
Reward	f(Distance, Clock, and Death)
Action	Left, Right, Jump left, or Jump right

Table 1: Reinforcement Learning framework for this problem

2.1 Agent

The agent will be Mario, which will take an action dependent on what is observed in the screen.

2.2 Environment

The environment is simply the game, specifically the level the agent is trying to beat. Open AI Gym provides multiple levels, but we only use SuperMarioBros-1-1-v0 which is the level 1-1. The agent, or Mario, moves around in this environment and then observes the state.

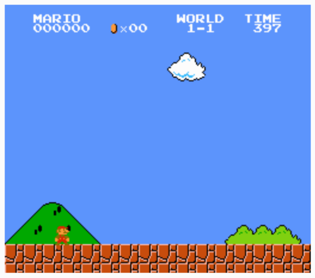


Figure 1: Super Mario Bros. Level 1-1, regular NES version.

2.3 Observed State

The state is simply the specific frame of the screen at each iteration. However, the environment provides a frame of size [3,240,256], but this is too large to process and unnecessary since we do not need color to make a decision. Therefore, the frames being provided to the agent are first down-sampled and set to gray scale. Then the frames will be of size [1,84,84], making the memory requirements much less and speeding up learning by decreasing the complexity need of our neural network. In addition, frames that are a small number of iterations away from each other do not vary much in what they show. So it would benefit training if only one frame is considered from a set of frames, preferably one with the most useful data. Thus the frames were first stacked by sets of 4, and then the max of the frames was taken in order to decrease overfitting as well as the amount of unnecessary training. This preprocessing method was taken from Feng et al [4].

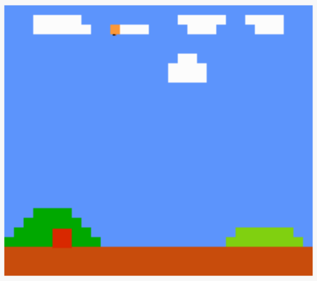


Figure 2: Super Mario Bros. Level 1-1, downsampled.

2.4 Reward

Open AI Gym provides a reward function for every step, which we used for our program as well. The distance metric provides a value equal to the distance between positions where moving right returns a positive reward. For the clock metric, Mario is given -1 point for every second it spends trying to complete the level. Lastly, a -15 point reward is given if Mario dies in the game. Overall, the reward is capped at -15 to 15 points for each step.

$$\begin{gathered}\text{reward} = \text{distance} + \text{clock} + \text{death} \\ \text{distance} > 0 \text{ if moving right} \\ \text{clock} = -1 \text{ for every second taken} \\ \text{death} = -15 \text{ if Super Mario Dies}\end{gathered}$$

2.5 Actions

The action space was purposely reduced so that training would speed up. The reward function only depends on how far Mario gets, how long he takes, and if he stays alive. Therefore, adding more complexity and actions would not affect the reward that Mario could receive. Thus, Mario has its actions reduced from 256 discrete actions [1], to just 4 actions: left, right, jump left, and jump right. We believed these are the only movements needed for Mario to get over obstacles, dodge enemies, and ultimately finish the level.

3 Proposed Solution

The point of this project was to see which algorithm would perform the best at solving the level. Therefore, an online off-policy algorithm is needed which leads us to use a Deep Q-Network (DQN) and a Double Deep Q-Network (DDQN) with epsilon-greedy policy [2] [3] [5] . We also utilize approximation through a neural network because the state space is too large to handle in tabular settings. In addition, both algorithms are supported by experience replay and fixed Q-target in order to help stabilize learning.

3.1 Neural Network Architecture

Input: (84,84,4)
CONV2D(Filters:32, kernel=(8,8),stride=4)
ReLU
CONV2D(Filters:64, kernel=(4,4), stride=2)
ReLU
CONV2D(Filters:64, kernel=(3,3), stride=1)
ReLU
Flatten Layer
Fully Connected (3136,512)
ReLU
Fully Connected (512, action size)

Table 2: Neural Network Architecture

3.2 Pseudocode

Algorithm 1: Super Mario Reinforcement Learning (training)

```
1 main();
  Input  : Super Mario Environment and Agent(either with DQN or Double DQN)
  Output : Agent,reward, win-rate
2 for number of episodes do
3     action = agent.act(past state) (using epsilon-greedy);
4     state,reward, done = environment.step(action);
5     save to Experience Replay: past state, state, action, reward, and done;
6     randomly sample Experience Replay to find error between fixed target and policy Q function;
7     optimize Double DQN parameters based on error;
8 end
9 return (trained Agent,Average Reward, Average Win Rate)
```

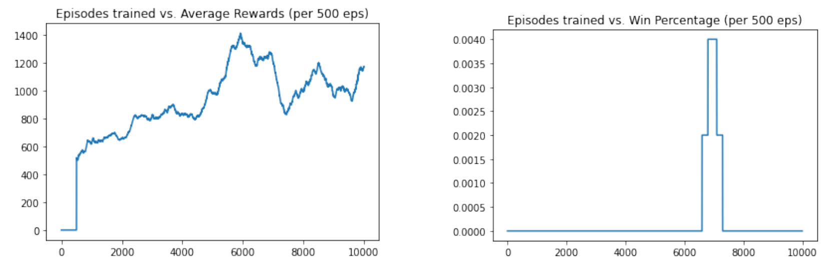
4 Training and Results

To see how the agent progresses, a graph of the score over episodes as well as win rate are obtained. The performances then are analyzed through completion rate, how fast each algorithm is able to solve the game, and the highest score achieved.

We trained our agents for 10,000 episodes. We found that the agent was best able to learn if the ϵ in the ϵ-greedy policy was decayed by a rate of 0.999 per episode. We found that a decay rate of 0.999 performs better than a decay rate of 0.99. We also experimented with lower bounding ϵ to 0.00001 in order to guarantee some exploration. We found that lower bounding ϵ performed better with regards to reward accrued than if we didn't lower bound ϵ. However, during training we would observe a slight decrease in win rate as the episodes go on because the agent is exploring more than it would if there was no lower bound on ϵ. We also found that any ϵ lower than 1 while decaying ϵ would be insufficient to encourage exploration. The DQN and Double DQN were optimized using ADAM and a SmoothL1 loss function. The pseudo code used to train the agent can be found in Algorithm 1.

4.1 Deep Q Network

As you can see below, DQN performs poorly with regards to accruing reward and especially poorly with respect to Win Rate, only winning a few times over the 10,000 episodes of training.

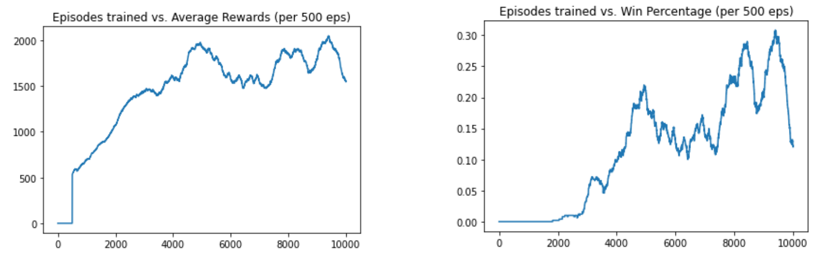


(a) Reward (b) Win Rate

Figure 3: Reward and Win Rate for DQN with ϵ=1.0 and Decay=0.999, ϵ lower bound = 0

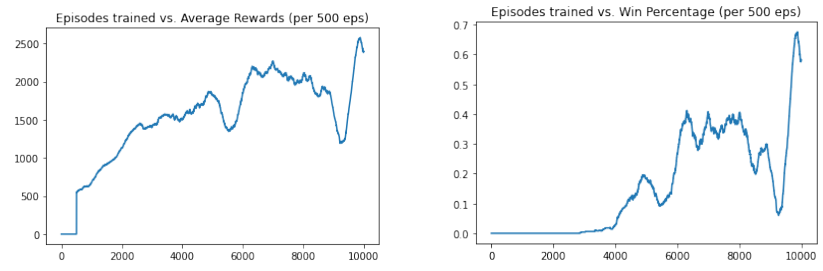
4.2 Double Deep Q Network

DDQN, regardless of the hyperparameters used, performs much better than DQN. The results for varying hyperparameters are shown in Figure 4, Figure 5, and Figure 6. We can see the best performing hyperparameters were ϵ = 1.0, decay = 0.999, and ϵ lower bound = 0.00001. Overall, Double DQN displayed high success with total reward reaching 2500 for some episodes, and high win rate by the end of training.



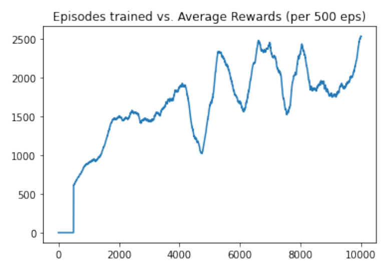
(a) Reward (b) Win Rate

Figure 4: Reward and Win Rate for DDQN with ϵ=1.0 and Decay=0.99, ϵ lower bound = 0

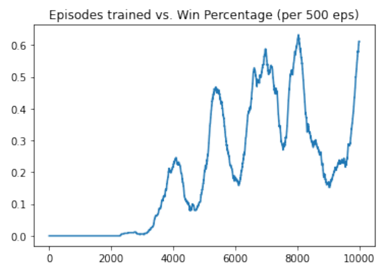


(a) Reward (b) Win Rate

Figure 5: Reward and Win Rate for DDQN with ϵ = 1.0 and Decay=0.999, ϵ lower bound = 0



(a) Reward



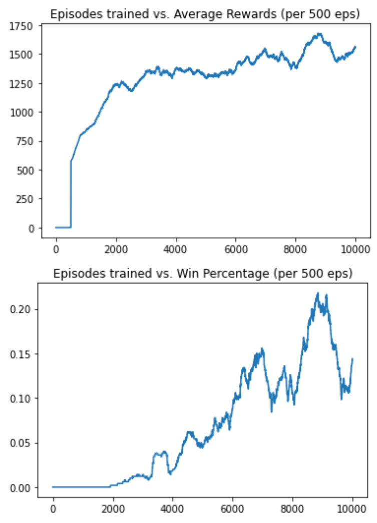
(b) Win Rate

Figure 6: Reward and Win Rate for DDQN with ϵ=1.0 and Decay=0.999, ϵ lower bound = 0.00001

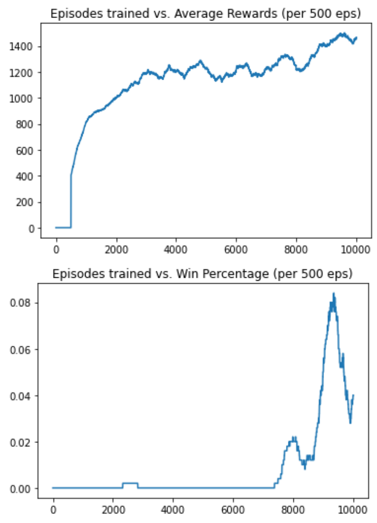
4.3 Extensions to Other Levels

We next experimented with other levels from the OpenAI Gym Mario environment. Specifically, we wanted to test the generalizability of our model to other Mario stages. We first tested our fully trained 1-1 Mario on other levels. Unfortunately, the model did not directly translate to other levels and Mario quickly died. It seems this is because of the addition of new enemies and obstacles in other levels that causes Mario to fail.

We then experimented with using transfer learning to speed up training on new levels. This was achieved by using a trained network on 1-1 and then beginning a new training process on another level. We experimented with the next level 1-2 and got the following results in Figure 7.



(a) Learning from Scratch



(b) Transfer Learning

Figure 7: Learning from scratch vs transfer learning on level 1-2

The plots show that although using the pretrained weights in transfer learning accelerates initial learning on the level, the network that learned from scratch outperforms it and is able to complete the level far more consistently at the end of training. This demonstrates that our model is almost overfitting to each level, learning how to avoid the unique obstacles instead of learning a more general approach that dodges new enemies as they come up.

5 Conclusion

We used an OpenAI Gym environment of a level of Super Mario Bros. to study how approximate Q-learning can be used to train an agent to complete the level. We wanted to observe how using an experience replay buffer and fixed Q-targets stabilizes learning for our agent. This led us to study how DQN and DDQN perform with respect to average reward accrued and win rate of the agent over 10,000 episodes. We showed DDQN works better than DQN in every performance measure regardless of what hyperparameters were used. We were able to find the optimal hyperparameters to train an agent to complete the level while maximizing the rewards obtained while attempting to finish the level.

In the future, we hope to extend this work on DDQNs to other environments including training on random levels from the Mario NES game to improve generalizability. Additionally, we could implement improvements on DDQNs like Dueling DQNs for testing on the same environment to compare their performances.

6 Member Contribution

Kasra Jalali - Training DQN and DDQN. Hyper-parameter tuning with respect to epsilon, epsilon decay, epsilon lower bounding, and more. Research and idea collection.

Jason Chung - Responsible for extensions to other levels/transfer learning and visualizing agent interacting with environment. Found and tweaked environment code for our application.

Joel Oh - Implementation and cleaning of algorithms along with gathering information about preprocessing code. Research on related works and various neural networks.

Alejandro Venegas - responsible for training the algorithm for solving in Super Mario Bros level 1-1 and hyper parameter tuning to maximize reward. In addition, responsible for DQN code.

References

[1] Christian Kauten. Super Mario Bros for OpenAI Gym. GitHub, 2018.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[3] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. 2015.

[4] Howard Wang Steven Guo Yuansong Feng, Suraj Subramanian. Train a mario-playing rl agent. Pytorch.

[5] Tom Zahavy, Nir Ben Zrihem, and Shie Mannor. Graying the black box: Understanding dqns, 2017.