Reinforcement Learning
Deep and Double Deep Q-Learning for Playing the first Level of Super Mario Bros NES

Andrew Grebenisan, Yifei Yin, Xiaotian Liu

November 2019

1. INTRODUCTION

Ever since 2015, deep Q-learning has shown to achieve phenomenal results in the domain of video game AI, as seen in the 2016 paper from OpenAI. All three of us have grown up playing video games and find it really exciting when we see AI perform at super-human levels, so we thought it would be interesting to tackle an old video game for our final project. Our game of choice is the original Super Mario Bros on NES, and we decided to attempt creating an agent that beats the first level. The agent in this game, Mario, needs to move from the start level to a flagpole at the very end. He needs to avoid monsters and holes, otherwise he is killed and has to restart the level. This is a really interesting problem because platformers are episodic, and the agent needs to master sequential sections of the game before being able to acquire new experiences later in the level.

We chose to use deep function approximation to master this game because the state space is massive. Ultimately, given that the frames of the game are images, we thought a convolutional neural network (CNN) would be suitable to solve this problem, and used a variation of the original Atari Deep Q-Network CNN. Additionally, we were interested in creating a comparison between using deep-Q learning and double deep-Q learning. Interestingly enough, after 10000 episodes, using double deep-Q learning, we managed to create an agent that beat the level one out of every two attempts, whereas using just deep Q-learning, the agent hardly ever beat the level.

2. PROBLEM FORMULATION

2.1 Learning Environment

We used a python learning library, gym-super-mario-bros[1] as our reinforcement learning environment. This library is based on OpenAI's retro gym[1], which simulates SNES games. The environment takes an action vector as input and produces the next state, reward, termination flag, and other environmental information. States are represented by a $240 \times 256 \times 3$ image, rewards are represented as an integer value from 15 to -15 depends on agent's location, the termination flag is a boolean value that represents whether a game has finished. We also have other environmental information like number of lives, number of coins etc. In our experiment we ignored the other environmental info, and we only care about distance traveled and other factors which we will discuss in the rewards section.

2.2 State Prepossessing and Representation

We used a gym wrapper to down-sample the state images to a grey scaled 84×84 matrix to reduce the dimensionality of the problem. We then concatenate the past 4 frames in a buffer to represent a single state. This allows us to represent the motion of our agent. The final input for our Q-learning neural network is a $4 \times 84 \times 84$ tensor. The wrapper code that we used was taken and slightly edited from the AtariDQN github repo [2].

2.3 Reward Function

The reward function is represented as a sum of V, C, and D variables. V represents the difference between an agent's current x position and previous x position. Each increase in x position gives a +1 reward. C represents the difference between the previous game clock and the current game clock. Each passing game clock will give the agent -1 reward to prevent the agent from staying still. The D variable penalizes the agent for death. It gives a -15 reward for dying. Together this reward function penalizes an agent for staying still and dying while rewarding the agent for moving towards the goal state, and the final value is clipped between -15 and 15.

2.4 Action Selection

OpenAi retro uses a vector of size 121 to represent all possible user input. We limited our action space to just 5 actions. They are 'Stay Still', 'Move Right', 'Jump', 'Jump Right', and 'High jump'. We used a NES wrapper library to map these actions to the original action spaces.

3. SOLUTION OVERVIEW

3.1 Library and Package Requirements

List of Libraries and Packages(install using pip or conda):

- Programming Language: Python 3.X
- Machine Learning Library: pytorch
- RL Enviorments: gym, gym_super_mario_bros, nes_py
- Others: random, collections, cv2, numpy, tqdm, pickle, matplotlib

[1] https://pypi.org/project/gym-super-mario-bros/
[2] https://github.com/LiamHz/AtariDQN/blob/master/Liam's_DQN.ipynb

3.2 Running our Code

In the last cell of the code, set training mode to False if you would like to render and visualize the agent maneuvering the environment. If training mode is True, you will not see the environment rendered, but the deep q-network(s) will be trained and saved to the local directory at the end of training. If you would like to use a pre-trained model, set pre-trained to True. If this is False, then the models used will be randomly initialized. If you would like to run DDQN, in the instantiation of the agent, set double_dq to True, else set it to False for single DQN.

3.3 Our CNN Function Approximator

Given that we had a large state space (image frames), we knew that tabular Q-learning would be very inefficient; thus, we decided to use convolutional neural networks (CNNs) as function approximators.[2] They take in states (4 frames of size 84×84 as input), and output the state-action values. The architecture that we used, identical to the Mnih *et al* 2015 paper,[3] is proven to work very well in retro games such as Breakout and Space Invaders. The architecture of the network is below:

1. Input layer: $4 \times 84 \times 84$ stacked gray scaled frames from the environment.
2. Convolution layer: 32 (8×8) kernels with stride size of 4, activated by rectified linear units.
3. Convolution layer: 64 (4×4) kernels with stride size of 2, activated by rectified linear units.
4. Convolution layer: 64 (3×3) kernels with stride size of 1, activated by rectified linear units.
5. Fully connected layer: 512 densely connected nodes, activated by rectified linear units.
6. Fully connected layer: 5 densely connected nodes, activated linearly.

The network that we used is from the same github repo that we used for the environment wrapper (footnote 2 on previous page). Beyond the wrapper and the network architecture, the rest of the code is entirely our own. More details on how the network(s) is/are trained will be stated in the next few sections.

3.4 Experience replay

Q-learning takes advantage of memory buffer. Previous experiences are stored in the memory of the agent. During training, a small amount of the memory will be sampled from the memory buffer and used to train the agent.[4] This, helps the agent to not forget previous outcomes of actions associated with previous environments. Having sufficient memory size can increase the accuracy and convergence rate.[5] In our experiments, current state, action, reward, next state and whether the game has ended are kept in the memory buffer for the training. In most implementations, a python deque is used to store memories, but in ours, we kept a separate tensor buffer for each of the STATE, ACTION, REWARD, STATE2, and DONE, which sped up training to around 2 seconds per episode from around 30 seconds to a minute. When we sampled, we made sure to sample the same indices from each of the buffers.

3.5 Deep Q-Network Algorithm

Below is the algorithm that we used for deep Q-learning. It is very similar to the one we saw in class, yet we slightly modified it since we using function approximation.

Algorithm 1: Q-learning algorithm from class, modified for deep Q-learning

Initialize weights for network Q_θ using Kaiming initialization, replay buffer $\mathcal{D}$
Initialize S
repeat (for each episode)
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update sample **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) = r_t + \gamma Q_\theta(s_{t+1}, \max_{a'} Q_\theta(s_{t+1}, a^{'}))$
    Perform gradient descent step on $L(Q^*(s_t, a_t), Q_\theta(s_t, a_t))$, where L is the Huber loss
  until s is terminal

3.6 Double Deep Q-Network

The DQN networks for double deep Q-learning used have the exact same architecture as the single DQN discussed above, yet the training algorithm that we used was slightly different from the one seen in class. In class, we observed the following algorithm for double Q-Learning.

Algorithm 2: Double Q-learning algorithm from class, modified for double deep Q-learning

Initialize Q^a_θ, Q^b_θ, s
repeat
  Choose a based on $Q^a_\theta(s,\cdot)$ or $Q^b_\theta(s,\cdot)$, observe r, $s^{'}$
  Choose either UPDATE(A) or UPDATE(B)
  if UPDATE A, **then**
    Define $a^* = \arg\max_a Q^a(s^{'}, a)$
    Compute target Q-value:
      $Q^*(s,a) \leftarrow r + \gamma Q^b_\theta(s^{'}, a^*)$
    Perform gradient descent step on $L(Q^*(s,a), Q^a_\theta(s_t, a_t))$
  else if UPDATE B, **then**
    Define $b^* = \arg\max_a Q^a(s^{'}, a)$
    Compute target Q-value:
      $Q^*(s,a) \leftarrow r + \gamma Q^a_\theta(s^{'}, b^*)$
    Perform gradient descent step on $L(Q^*(s,a), Q^b_\theta(s_t, a_t))$ where L is the Huber loss
  end if
  $s \leftarrow s^{'}$

The algorithm from class would be very inefficient since we would require two separate optimizers and the policy that we choose is randomly defined by the next best action of one of the two networks with probability $1-\epsilon$, where ϵ is the exploration rate. A solution to this problem would be to have a primary policy network Q_θ and a target network $Q_{\theta'}$, and copy the policy network weights into the target network every certain number of episodes. The algorithm is as follows:

Algorithm 3: Our modified double deep Q-Learning algorithm, similar to the Hasselt *et al.* 2015 paper

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $N_{steps} = 0$, copycount C initialized arbitrarily
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
    $N_{steps} = N_{steps} + 1$
  for each update sample **do**
    Update target network parameters:
    if $N_{steps} \mod C = 0$, **then**
      $\theta^{'} \leftarrow \theta$
    end if
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) = r_t + \gamma Q_\theta(s_{t+1}, \max_{a'} Q_{\theta'}(s_{t+1}, a^{'})$
    Perform gradient descent step on $L(Q^*(s_t, a_t), Q_\theta(s_t, a_t))$, where L is the Huber loss

The original algorithm[6] had the target network being updated for each update sample; however, we chose to limit the number of updates for computational efficiency. We also chose to use the Huber loss function instead of the originally proposed mean-squared error, since it balances out the effects between mean-squared error and mean-absolute error. The Huber loss function is defined as

$$L(y, f(x)) = \begin{cases} \frac{1}{2}\left[y - f(x)\right]^2 & \text{for } |y - f(x)| \le \delta, \\ \delta\left(|y - f(x)| - \delta/2\right) & \text{otherwise.} \end{cases} \tag{1}$$

where δ is a user-defined hyper-parameter, y is the true value, and $f(x)$ is the approximated value of y.

3.7 Training Setup and Hyper-parameter Selection

We limited our episode length 10000 episodes for both networks. Initially, we used hyper-parameters suggested by the original deep-Q learning paper.[7] To encourage our agent to explore, we used a decreasing ϵ. We set our discount ϵ rate to 0.9, and we used a variable ϵ that decreases from 1 to 0.1 by a factor of 0.995. However, we observed that our reward became much more stable with a lower learning rate. We then adjusted our minimum learning rate to 0.02 and the discount factor to 0.99. We used Pytorch for training our neural networks.

For optimization, we used the Adam optimizer, Huber loss function, and a batch size of 32 experiences. For experience replay, we kept torch buffers of size 30000 for each of the states, actions, rewards, next states, and boolean terminal flags, and kept a variable which represented the next index to insert an experience. These buffers have the deque quality in that when they are full, the 0^{th} index is used to insert the next experience. After each step our agent has taken, we randomly sampled 32 states and updated our deep-Q network accordingly. A summary of our tested hyper parameters are listed below.

Hyper Parameters	Deep-Q	Double Deep-Q
Batch Size	32	32
Memory Buffer Size	30000	30000
Max ϵ	1	1
Min ϵ	0.02	0.02
Discount ϵ	0.99	0.99
Discount Rate	0.9	0.9
Optimizer	Adam	Adam
Loss Function	Huber	Huber
CNN Learning Rate	0.00025	0.00025
Copy count C	N/A	5000

4. RESULTS

Our Deep-Q Learning Network took 6 hours to finish 10000 episodes, and our Double Deep-Q learning Network took 5.5 hours to finish 10000 episodes. We recorded total rewards gained by the agent at the end of each episode for both training sessions. Due to exploration in the ϵ-greedy policy, rewards can be volatile. The resulting graph would be hard to interpret. We therefore used a moving average of 500 episodes to describe our agent's improvement over time. Another way to measure our network performance is the win probability for each network. We calculated the percentage of beating the level over the past 500 episodes. These results are shown in graphs below.

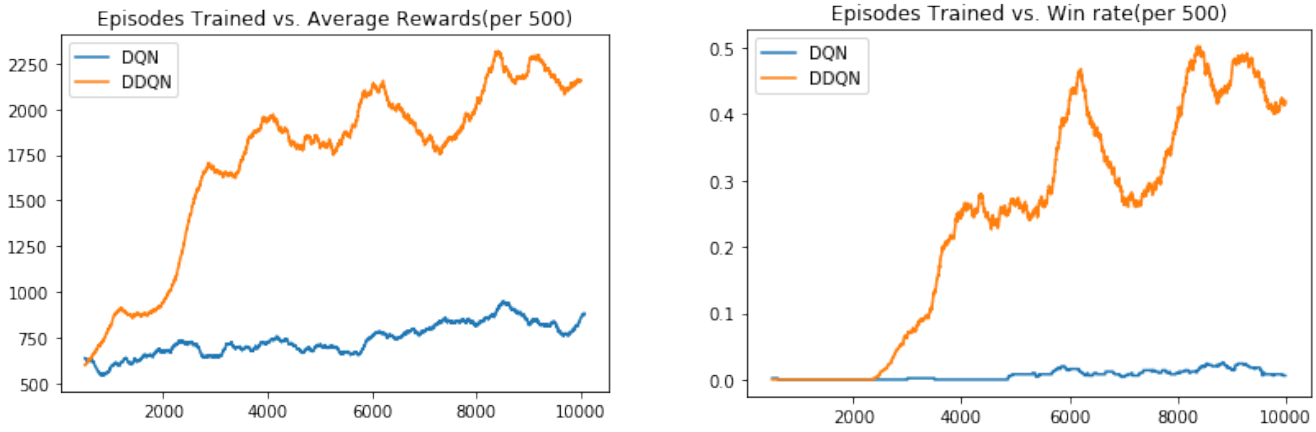

Figure 1: **Left:** Average reward per 500 episodes for 10000 episodes in total, DQN and Double DQN solutions, **Right:** Average percentage of wins per 500 episodes for 10000 episodes in total, DQN and Double DQN solutions

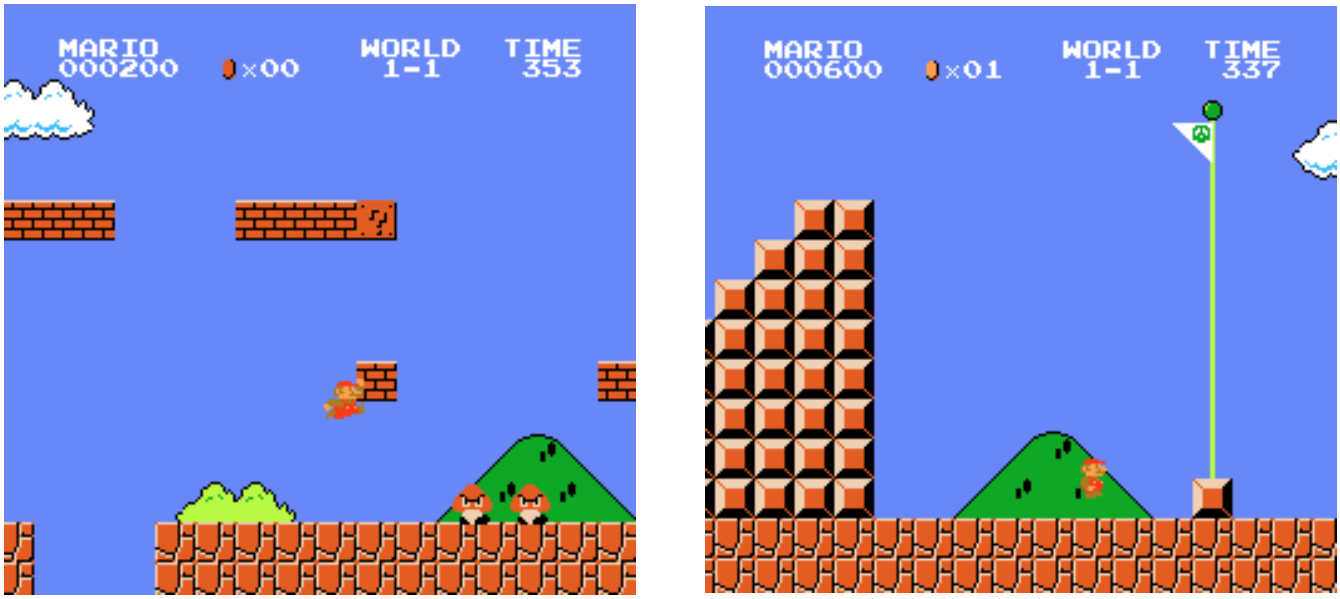

Figure 2: **Left:** Example screenshot from our DQN agent while training, **Right:** Example screenshot from our DDQN agent completing the level

We also provided three videos in .gif format, one which shows a randomly initialized agent acting in the environment, one which shows an attempt at beating the level using just a DQN (however this method did not make it to the end of the level), and one video which shows the DDQN method beating the level.

5. DISCUSSION

Our DDQN was able to beat the level around 50% of the time within 10000 episodes. Our behaviour network is ϵ-greedy with an exploration rate of 0.02. This means that at each step our agent has 2% chance of performing a random action. These random exploration compounded through an entire episodes, which contains hundreds of steps, will not allow win probability to converge to 1. We believe that 50% win-rate is a good result comparing to other works in related area.[7] These results clearly show superior performance of a DDQN compare to a DQN. Although both networks have shown improvement in performance over time, DDQN has proven to converge much faster than DQN in our experiments. We believe that maximization bias plays a central role in explaining the superior performance of the DDQN network. The choice of CNN structure is the same for both networks and the hyper parameters are the same as well. Due to the computation time constraint, we didn't greedy explore all possible combinations of hyper parameters. If given more time and resources, we could experiment with more the single DQN. We can set a much higher minimal ϵ to encourage our agent to explore more.

5.1 Deep-Q Network

Our single Deep-Q Network converges very slowly during our experiments. The average rewards are no better than random during the first 1000 episodes of training. After around 4000 episodes, we observed slight improvements, and the network begins to complete the level. The win percentage hovers around 2 percent for the remaining episodes. The reward per episode approaches to 1000 at the end of the 10000 episodes. We manually rendered some of the models that are saved during the experiments. We found that the agent is very prone to stuck or jumping into pits. The agent eventually learns to avoid these problems but at much slower than the Double Deep-Q Network. We believe that this is an example of maximization bias for the single Deep-Q Network. Due to the stochastic nature of ϵ -greedy policy and fuzziness of image as input, the agent's action and reward behave in a stochastic manner. Once an agent gains a large reward for an action, it will continue to repeat it because its optimal policy was updated according. This explains why the single DQN agent gets stuck in a negative loop much more often compared to a Double DQN.

5.2 Double Deep-Q Network

The Double DQN managed to converge to a policy that won the game one every two times. The fact that it did not reach the flagpole every single time is probably due to the fact that we set the minimum exploration rate to 0.02. It is interesting that within 2000 episodes, it learned to successfully make it through most of the level, but it began learning very slowly after this point. This is most likely due to the fact that since we are randomly sampling 32 experiences from a buffer of size 30,000, most of the experiences that the agent learned from are towards the beginning of the level.

6. CONCLUSION AND LESSONS LEARNED

As can be seen, using Double Deep Q-Learning, we managed to create an agent that found an optimal policy for making it through the first level of Super Mario Bros NES, however a single DQN would need many more episodes of learning to reach the performance of the DDQN. The DDQN outperformed the single DQN because it reduced the maximization bias that is prevalent in standard Deep Q models. From class, we knew some of the pitfalls of function approximation in reinforcement learning, the most important one being that when the approximation gets better for some state-action value pair, it will indefinitely affect other unrelated state-action value pairs. Now we actually got to see a practical application of this knowledge, with the reward function swaying up and down, mainly in the DDQN experiment, representing how the agent unlearns occasionally.

We learned that maximization bias plays an important role in training speed. Single DQN consistently stuck in local minima while a DDQN can overcome these problem much faster. Although single DQN works for many games with immediate rewards such as breakout and space invaders, DDQNs work much better for games with sparse rewards, like Mario. Throughout this exercise, we became much more careful when accessing the performance of different reinforcement learning algorithms and their potential applications.

For possible future works, we would like to implement a prioritized experience replay algorithm suggested by Schaul *et al* .[8] Instead of randomly sampling from past experiences, prioritized replay algorithm will first update memories with large discrepancy between prediction and actual returns. Schaul *et al* has shown a huge improvement for DQN in terms of convergence speed. Also with more time and computational resources, we would like to add special indicator features such as speed of agent, enemy presence, and getting stuck in front of pipes on top of our current models.

REFERENCES

1. C. Kauten, "Super Mario Bros for OpenAI Gym." GitHub, 2018.
2. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds., pp. 1097–1105, Curran Associates, Inc., 2012.
3. V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature* **518**(7540), p. 529, 2015.
4. L.-J. Lin, *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX93-22750.
5. R. Liu and J. Zou, "The effects of memory replay in reinforcement learning," *CoRR* **abs/1710.06574**, 2017.
6. H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," *arXiv e-prints* , p. arXiv:1509.06461, Sep 2015.
7. V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602* , 2013.
8. I. A. D. S. Tom Schaul, John Quan, "Prioritized experience replay," *arXiv:1511.05952* , 2016.