Mastering Classic Video Games: A Deep Reinforcement Learning Approach for Non-Open World Environments

A. Audren, B. Phan, University of Cincinnati, deep learning (fall 2023) students

ABSTRACT - THIS paper has been written as part of an academic project. The original goal was to show what is possible to do when making learn an AI to play video games. We also wanted to make this type of project more accessible for the community by providing a brief state of the art of existing approaches. As mentioned in the title, complex video games, which often implies an open world environment, won't be covered here. The main reason for it is that the q-learning technic used in our project requires clear objectives and a well-defined reward given by the game as response for decisions taken. The term "classic" referring to arcade and platform games (simple racing games could also enter in this category). First, it remains important to understand it is not a concern of how recent a game is. In fact, a game like Pokemon Red game boy version (1996), would be much more challenging for the development of an AI than a game like Geometry Dash (2013). It is all about regular rewards to guide the player. For this reason, short levels with reduced number of secondary objectives will be a lot easier for the AI to "understand" the policy and making progress. In essence, this paper is not merely an academic exposition; it is a roadmap, a comprehensive guidebook navigating through the intricate intersections of classic video games and deep reinforcement learning in non-open world environments. It beckons researchers, enthusiasts, and practitioners alike to delve into the rich tapestry of possibilities and challenges that emerge when machines learn to play, and perhaps master, the games that have captivated human imagination for decades.

I. INTRODUCTION

In the realm of reinforcement learning and its application to gaming environments, the selection of an appropriate emulator stands as a critical cornerstone. This choice is paramount for practical reasons, as it serves not only to facilitate the illustration of the models expounded in this paper but also to establish a seamless interface with the rendering API of the game. The integration of an emulator with the ability to engage in dialogues with the game's rendering API is instrumental for constructing an overlay conducive to reinforcement learning techniques. This pivotal aspect is meticulously elucidated in the forthcoming Part II of this paper, where the practical considerations guiding the selection of emulators are comprehensively detailed.

Moving forward to Part III, the focus shifts to the foundational principles underpinning q-learning techniques, R. S. Sutton and A. G. Barto (1998). A pseudo implementation is thoughtfully provided to afford a nuanced comprehension of the underlying theoretical constructs. While q-learning has demonstrated efficacy in various scenarios, it grapples with a substantial drawback - the model responsible for action selection is also responsible for action evaluation. This inherent overlap introduces a bias with each decision-making instance. The ensuing section, Part IV, engages in a thorough exploration of the imperative need to introduce novel models (van Hasselt, 2010) aimed at mitigating these overestimation issues.

Part V delves into the multifaceted challenges associated with resource limitations during the training phase of a q-learning model. A conscientious examination of strategies to expedite the learning process takes center stage. Architecture of the parallelized q-learning model is presented. The multiplication of emulation agents enables us to collect an increasing number of training experiences and to train continuously a central model. We also mention the distributed q-learning which leverages from multiple GPUs to accelerate computation of gradients and training of model's parameters.

II. EMULATION AND LEARNING ENVIRONMENT

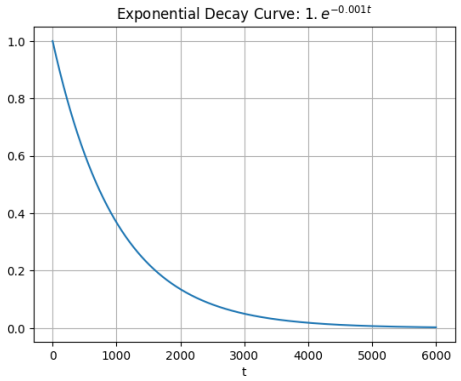
Because we need a perfect game feedback of our past decisions, namely the frames of the game to determine the current state, numerical scores, and statistics of the game to judge how close we are from "winning", it is essential to dispose from an emulator with accessible variables containing these parameters. If these conditions are not met, we at least need the source code of the game. With retro-engineering we can still find where these memory cells are located. The last option is to code the game ourselves from the beginning, but it could significantly increase the development duration.

The most known AI development environment for video games is the Open AI GYM environment [5]. A toolkit simplifying the development of reinforcement learning agents for arcade games. Gym's standardized interface streamlines integration, allowing us to focus on algorithms. With pre-built game environments, Gym provides a versatile platform for evaluating and benchmarking reinforcement learning approaches in video games. Another option available, especially if you prefer working on game boy video games: PyBoy, an emulator for the Nintendo Game Boy (DMG-01), distinguishes itself by implementing a fast and comprehensive emulation of the iconic handheld console in Python 2.7. Developed from scratch, it excels in running cartridge dump software at speeds equivalent to the original Game Boy. While sound and serial port functionalities are omitted, PyBoy utilizes SDL2 and NumPy for graphics and user interactions [1]. The main advantage for this game boy emulator compared to existing ones is the use of Python, where others (mGBA, BizHawk) are using Lua to dialog with the emulator. Which means that all the machine learning (ML) libraries can be imported and used to train an AI. Frameworks like Tensorflow or PyTorch can then be used, especially to train a convolutional neural network efficiently, which is crucial for real time learning. Some wrappers have been specifically designed for bot and AI development on PyBoy. One of them is covering Super Mario Land, the old version of Super Mario Bros which we'll be using to illustrate our research.

III. The Simple Deep Q-Learning Approach

A. Principle

Since most of games contain many different states, especially Super Mario Land, we won't cover the q-table use. Rather, we'll favorize use of a deep neural network to predict the q-values associated to a pair action-state. At each time step, the current frame feed the neural network, usually a convolutional neural network, so the agent can choose the most rewarding action to execute. The key here is the "experiences replay". Each experience consists of a first state, usually the frame of the game, an action executed by the agent, the next state/frame provided by the game consecutively to this previous action as well as a reward. A memory buffer containing all these past experiences is regularly shuffled, to ensure the agent won't learn from correlated experiences, and used to train the neural network to predict the correct q-values. First, the agent doesn't know anything about the policy of the game, it has to learn. For this reason, we are using an exponentially decreasing exploration probability. At the beginning of the game, the agent will execute randomly selected actions to constitute experiences. More the agent plays and more it will use its neural network to decide which action to choose.



$N(t) = N_0 e^{-\lambda t}$ with N_0 initial exploration probability, often equal to one.
λ the exploration decreasing decay, which must be relatively low.

Figure 3.1: *exploration probability decay*

B. Implementation

Algorithm 1 Simple DQN Main

```
while True do
    s_t ← Game.area()
    a_t ← Agent.compute_action(s_t)
    Game.send_input(a_t)
    Game.next_frame()
    reward ← Game.score()
    s_{t+1} ← Game.area()
    Agent.store_episode(s_t, a_t, reward, s_{t+1})
    if end_of_episode then
        Agent.train()
        Agent.update_exploration_probability()
        Game.reset()
    end if
end while
```

Algorithm 1:
The first algorithm provides a high-level implementation of a main script for the deep q-learning (DQN). The compute action method calls the deep neural network to predict the q-values vector as output for the game area which is a matrix representing the current frame. Depending on the uniform probability returned and the current exploration probability, the agent will choose the most rewarding action accordingly to the compute action method. The train method is mainly based on the q-learning equation that we recall here:

$$q_{new}(s_t, a_t) = R_{t+1} + \gamma \max_{a_{t+1}} q(s_{t+1}, a_{t+1}) \qquad (3.1)$$

Direct consequence of the Bellman optimality principle.

Algorithm 1 Simple DQN Class Agent

function COMPUTE_ACTION(s_t)
  $X \sim \mathcal{U}[0,1]$
  if $X < N(t)$ **then**
    return random action
  else
    $q(s_t, a_i) \leftarrow$ Agent.predict(s_t)
    return $\arg\max_{a_i} q(s_t, a_i)$
  end if
end function
function TRAIN
  shuffle(memory buffer)
  for experience in memory buffer **do**
    $q(s_t, a_i) \leftarrow$ Agent.predict(experience[s_t])
    $R_{t+1} \leftarrow$ experience[reward]
    $q(s_{t+1}, a_i) \leftarrow$ Agent.predict(experience[s_{t+1}])
    $q(s_t, experience[a_t]) \leftarrow R_{t+1} + \gamma \times \max_{a_i} q(s_{t+1}, a_i)$
    Agent.fit(experience[s_t], $q(s_t, a_i)$)
  end for
end function

Keep in mind that your neural network model has to work with real time inferences so keep it simple. The number of layers does not need to exceed the number of three for most of classical games. Regarding the input size, try to only feed your network with relevant and necessary information. With Super Mario Land, we are not using the entire frame of the game but only a matrix which summarizes the game area. The initial frame size of PyBoy emulator is 160×144 pixels. The matrix we are using has a size of 16x20 which considerably reduces the computation time knowing that your agent will reach a point where it will decide by himself for almost all of actions which implies that the neural network will process a current state every 100 ms.
We can leverage the number of channels by using them to help the agent to "understand" a movement more than an action by feeding the network with for instance the five successive frames following a decision taking. In our case, this means we will have an input of size (16, 20, 5). The outputs are simply the q-values predicted for each possible action (see annex).

[1] Original paper [4].

IV. The double deep Q-learning Approach

A. *Necessity of dual networks*

As you already noticed the q-learning (3.1) computation includes a maximization step which tends to favorize overestimations to underestimated q-values. It could have no consequences on the learning if these overestimations were uniformly spread which is unfortunately not the case. It has been showed that the average overestimation (Thrun and Schwartz (1993)[1]) which has originally been defined by:

$$\mathbb{E}\left[\gamma\left(\max_{a_i} q(s_t, a_i) - q^*(s_t)\right)\right]$$

if the noise variables are independents and uniformly distributed in the interval $[-\varepsilon, +\varepsilon]$, is equal to $\gamma\varepsilon\frac{m-1}{m+1}$ (proof in appendix). where m is the number of possible actions. These overestimations are asymptotically leading to a sub-optimal policy.
The noise introduces can be from different sources: environmental, function approximation (especially the neural network) or non-stationarity. What this result is showing us is, even an extremely small error on q-values approximations, because of the maximization step, will propagate through the time so we eventually end up with estimations high above the true values.

Without the independent assumptions for noise variables, we can still obtain an upper bound for the q-values estimations. Assuming these errors are unbiased such as:

$$\sum_{i=1}^{m}\left(q(s_t, a_i) - q^*(s_t)\right) = 0\ ,$$

but not all equal to zeros in a way that

$$\frac{1}{m}\sum_{i=1}^{m}\left(q(s_t, a_i) - q^*(s_t)\right)^2 = C \quad with\ C > 0.$$

$$\text{Then } \max_{a_i} q(s_t, a_i) \geq q^*(s_t) + \sqrt{\frac{C}{m-1}}.$$ [2]

Which shows that even if the errors are mutually compensated, the smallest non null error can drive the estimation up. Even though this lower bound decreases with the number m of action, the average overestimation increases with the number of actions as shown with the figure 2.
The solution found to avoid that the noise introduced is propagating is the use of a second prediction model, often called target model. In simple q-learning, both evaluation and selection steps are realized by a unique model. In double q-learning, the initial model θ is assigned to select the maximum rewarding action to execute and a second model θ^- has to evaluate the q-values.

$$q_{new}(s_t, a_t, \theta) = R_{t+1} + \gamma q^-\left(s_{t+1}, \underset{a_i}{argmax}\, q(s_{t+1}, a_i, \theta), \theta^-\right) \qquad (4.1)$$

[2] Details on the proof can be found on appendix of [3].

In practice, the second model θ^- should be updated regularly with the parameters of the first model θ.

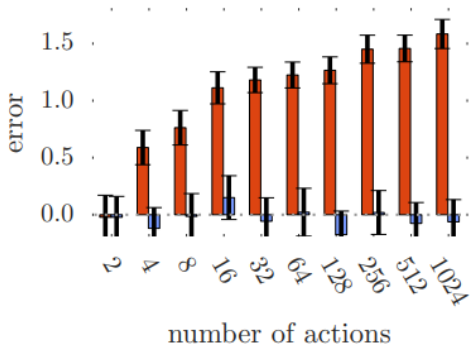


Figure 4.1:
Simple and double q-learning error comparison. The red bars are representing the bias introduced when using a simple q-learning approach:

$$\max_{a_i} q(s_t, a_i) - q^*(s_t)$$

While the blue bars are representing the bias obtained with a double q-learning approach:

$$q^-\left(s_t, \arg\max_{a_i} q(s_t, a_i)\right) - q^*(s_t)$$

The noise generated are standard normal random variables.

B. Implementation

Algorithm 2 Double DQN Main

```
while True do
    s_t ← Game.area()
    a_t ← θ.compute_action(s_t)
    Game.send_input(a_t)
    Game.next_frame()
    reward ← Game.score()
    s_{t+1} ← Game.area()
    θ.store_episode(s_t, a_t, reward, s_{t+1})
    if end_of_episode then
        θ.train(θ^-)
        θ.update_exploration_probability()
        if time_to_update then
            θ^- ← θ
        end if
        Game.reset()
    end if
end while
```

Algorithm 2 Double DQN Class Agent

```
function TRAIN
    shuffle(memory buffer)
    for experience in memory buffer do
        q(s_t, a_i) ← Agent.predict(experience[s_t])
        R_{t+1} ← experience[reward]
        q(s_{t+1}, a_i) ← Agent.predict(experience[s_{t+1}])
        q(s_t, experience[a_t]) ← R_{t+1} + γ × q(s_{t+1}, arg max_{a_i} q(s_{t+1}, a_i, θ), θ^-)
        Agent.fit(experience[s_t], q(s_t, a_i))
    end for
end function
```

Algorithm 2:
The double deep q-learning implementation is pretty like the simple one. Instead of instantiating only one agent, we are instantiating two agents at the beginning of the main script. The evaluation of target values changes accordingly to equation (4.1). As we mentioned, the target model must be updated regularly to follow the learning process.

C. Results

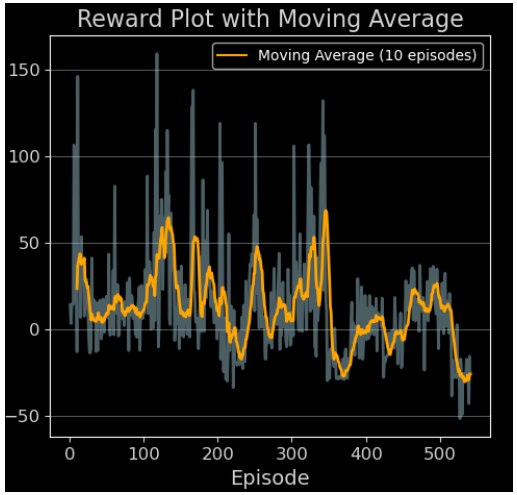


Figure 4.2:
We plotted the moving average of the reward the agent obtains for each episode. An episode can be considered as a sufficient number of steps or simpler when Mario loses his two lifes.

When setting the training size (parameters corresponding to the size of the batch used to feed the model) to a low number (relatively to the complexity of the game), the

exploration probability is decreasing faster than the model learns. We then end up with only based-network decision making but the model still is naïve.

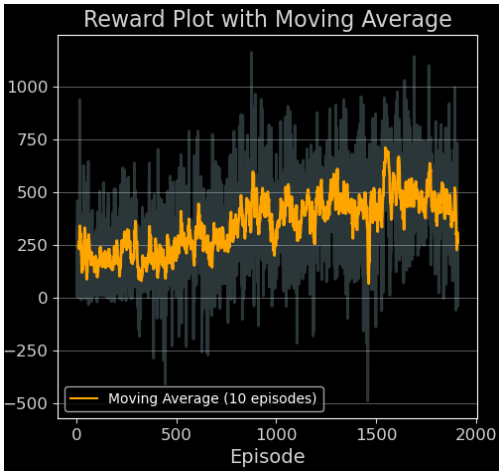


Figure 4.3:
When fixing the previous issue related to the training size, we are obtaining a better curve of reward. But the agent can sometimes get stuck in a pattern and is not moving forward. To avoid encountering this specific issue, we set a minimum value the exploration probability cannot exceeds so random actions help the agent to unlock it.

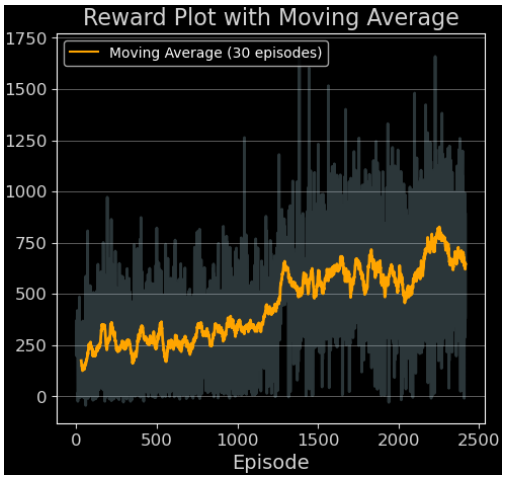


Figure 4.4:
Once we update the algorithm with the previous note, we get an increasing curve of reward which means the agent is learning.
We successfully completed 50% of the first level.

One way to fix the "not moving forward" issue is to force the agent to head to the right of the screen by rewarding it whenever it decides "go right" and when this enabling it to discover new parts of the level.
Another improvement could be added by adopting an adaptative exploration strategy with a dynamic exploration probability depending on the current performance of the agent. This way, each time the agent encounters new obstacles, its performance will level off resulting in an increasing exploration probability enabling him to learn new local patterns.

In reinforcement learning, the way we are computing the reward function is crucial as all the decisions made are depending directly on the q-values and thus on the reward. If we emphasize the number of collected coins, then Mario will adapt his behavior to collect more coins since we defined "collecting coins" as a winning behavior by assigning a larger part of the reward function to the number of collected coins. All the point is finding the "right" balance between the different scores inside the reward function (either scores of the game or score we decided to set up).

The time and resources needed to make an agent learn to play Super Mario Land are extremely expensive. The fifth part of this paper is presenting some implementations to accelerate the learning process.

V. Parallelized Learning

A. Principle and implementation

One of the main issues with games containing many different states is the time necessary for a DQN model to converge to an optimal policy. There are numerous ways to accelerate the learning process. We are providing here the most popular.
The time spent to collect experiences seriously affects the learning speed.
With only one process running, when we finally filled out the batch memory, the game must stop so the model can start fitting with target q-values. Multiprocessing then becomes obvious to multiply the learning speed. The reason we are using multiprocessing but not multithreading is that python uses a global interpreter lock (GIL) which makes the multithreading a non-real parallelized process. The architecture remains nonetheless simple. The idea is to run parallelized instance of the game through child processes. Once a child has collected enough of experiences it pushes its buffer through a waiting list or queue. On the other side of the pipe, the parent process, who's the assignment is to train the shared model, will process the shared experiences and fit with it. It can now push back its model parameters to another pipeline so the child will retrieve the updated parameters to take better decisions.

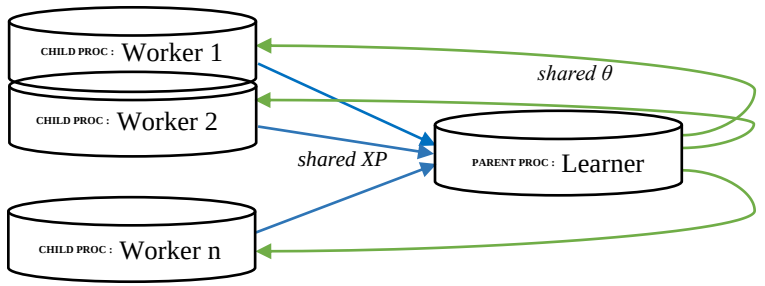


Figure 5.1: *Parallelized DQN. Blue arrows are representing the flows of experiences and green arrows are representing the flows of new model 'parameters.*

If enough workers have been launched, we then have enough of new experiences at each time step, the learning can thus be executed continuously which enables us to improve our model in a continuous way while collecting experiences through child processes. The parallelization of training agents avoids dead time which considerably reduces the learning time necessary to converge.

Another way to accelerate the convergence consists of distributing the fit step between different GPUs (see [6]). We can split the buffer of experiences between our available GPUs and wait them to return a gradient specific to the batch we sent them. A centralized server can then reassemble the gradients to train the central model. The distributed learning enables to reduce considerably the amount of time necessary to fit the model with target q-values. In practice, python provides a multiprocessing library which comes with many methods depending on use cases. In particular, the Manager class provides an efficient way to control a server process and handle shared objects with other processes using proxies.

VI. Conclusion

As we seen, a deep q-learning model is well adapted to master a video game involving clear objectives and regular rewards. But its simple DQN approach is suffering from an overestimation bias when evaluating the q-values. The double DQN approach brings a way to correct these values and being able to make unbiased decisions. For simple (mostly arcade) games, running these implementations on a single process can be sufficient. But when training an agent to play a more complex game, involving many different states (in practical terms, frames) a multiprocessing approach as well a distributed learning become essential. It enables to both to avoid dead time during learning and accelerate the training of the convolutional network.

The network used so far remains simple (the complexity involved is similar to solve a MNIST classification problem) for the task we're trying to achieve. The real difficulty lies in the setting of hyperparameters, namely the exploration probability, sizes of buffer and batch used, the reward computation based on given scores and the number of episodes. Most of the work was "adjust setting and retry".

VII. Appendix

Result 1:

Assuming the estimation errors are independently and uniformly distributed in the interval $[-\varepsilon, +\varepsilon]$,

$$then, \quad \mathbb{E}\left[\max_{a_i} q(s_t, a_i) - q^*(s_t)\right] = \varepsilon\frac{m-1}{m+1}$$

Proof:

Note $\mathcal{E}_{a_i} = q(s_t, a_i) - q^*(s_t)$ the estimation errors. We are searching for the density of the random variable $\max_{a_i} \mathcal{E}_{a_i}$. Its cumulative distribution function (CDF) can be written as

$$\mathbb{P}\left(\max_{a_i} \mathcal{E}_{a_i} \leq x\right) = \mathbb{P}\left(\bigcap_{i=1}^{m}(\mathcal{E}_{a_i} \leq x)\right)$$

$$= \prod_{i=1}^{m}\mathbb{P}(\mathcal{E}_{a_i} \leq x), \ where\ \mathbb{P}(\mathcal{E}_{a_i} \leq x) = \begin{cases} 0\ if\ x \leq -\varepsilon \\ \dfrac{\varepsilon + x}{2\varepsilon}\ if\ x\ \epsilon[-\varepsilon, +\varepsilon] \\ 1\ if\ x\ \geq +\varepsilon \end{cases}$$

Therefore, the density is:

$$\frac{d}{dx}\mathbb{P}\left(\max_{a_i} \mathcal{E}_{a_i} \leq x\right) = \frac{d}{dx}\left[\left(\frac{\varepsilon + x}{2\varepsilon}\right)^m\right]$$

$$= \frac{m}{2\varepsilon}\left(\frac{\varepsilon + x}{2\varepsilon}\right)^{m-1}$$

$$\mathbb{E}\left[\max_{a_i} q(s_t, a_i) - q^*(s_t)\right] = \int_{-\varepsilon}^{+\varepsilon} x\frac{m}{2\varepsilon}\left(\frac{\varepsilon + x}{2\varepsilon}\right)^{m-1} dx$$

$$= \left[x\left(\frac{\varepsilon + x}{2\varepsilon}\right)^m\right]_{-\epsilon}^{+\varepsilon} - \int_{-\varepsilon}^{+\varepsilon}\left(\frac{\varepsilon + x}{2\varepsilon}\right)^m dx$$

$$= \varepsilon - \frac{2\varepsilon}{m+1} = \varepsilon\frac{m-1}{m+1}$$

References

Online documents:

[1] PyBoy official GitHub repository: https://github.com/Baekalfen/PyBoy.git

[2] Deep Learning for Video Game Playing - Niels Justesen, Philip Bontrager, Julian Togelius, Sebastian Risi, IT University of Copenhagen, New York University arXiv:1708.07902v3 [cs.AI] 18 Feb 2019

[3] Deep Reinforcement Learning with Double Q-learning - Hado van Hasselt and Arthur Guez and David Silver, Google DeepMind arXiv:1509.06461v3 [cs.LG] 8 Dec 2015

[4] Issues in Using Function Approximation for Reinforcement Learning - Sebastian Thrun and Anton Schwartz, Dec. 1993 https://www.ri.cmu.edu/pub_files/pub1/thrun_sebastian_1993_1/thrun_sebastian_1993_1.pdf

[5] Open AI gym official website https://gymnasium.farama.org/

[6] Distributed Deep Q-Learning: Kevin Chavez, Hao Yi Ong and Augustus Hong
Stanford University
https://stanford.edu/~rezab/classes/cme323/S15/projects/deep_Qlearning_report.pdf

ANNEX

Link to our GitHub project:

https://github.com/BenjaminPhan34/DL_Final_Project_RL.git

Hyperparameters used:

$learning\ rate = 0.001$ $min\ exploration\ probability = 0.05$ $\gamma = 0.99$

$exploration\ decay = 0.005$ $train\ size = 2560$

Model architecture