COMP390

2021/22

# Reinforcement learning

Deep Double Q-Learning Algorithm for Atari Games


| | |
|---|---|
| Student Name: | Yuhang Song |
| Student ID: | 201403970 |
| Supervisor Name: | Dominik Wojtczak |


DEPARTMENT OF
COMPUTER SCIENCE


University of Liverpool
Liverpool L69 3BX

Dedicated to HaoranQiu
With Loves.

# Acknowledgements

I'd like to thank my parents, Shengyun Song and Zhengsu Zhu, for their support throughout my studies. Thanks also to my friends, Shenpu Zhou and Shun Fu, for putting up with my stressful outbursts while I was writing this dissertation.

I dedicate this to my good friend Siyuan and my lifetime academic teacher YangYang, who have helped me enormously throughout my time at university, and to the memory of Haoran.

COMP390

2021/22

# Reinforcement learning

Deep Double Q-Learning Algorithm for Atari Games

DEPARTMENT OF
COMPUTER SCIENCE

University of Liverpool
Liverpool L69 3BX

# Abstract

This dissertation is mainly about an approach to use one of the Reinforcement Learning algorithms namely Double Deep Q-Learning to train an intelligent agent to play an Atari game called “DemonAttack-v0”, together with experience replay.

Referenced from many existing previous works done in this area, modifications are made for this particular game, including changes to Epsilon-Greedy policy, Customized Reward Policy and Multi-frame Capturing, etc.

In particular, I will first show the idea behind the DQN, then talk about the modifications and improvements made for game “DemonAttack-v0”.

# Table of Contents



# Table of Figures

# 1.Introduction & Background

Like many other common Reinforcement Learning scenarios, I am dealing with a situation that an agent is constantly interacting with its suited environment, exchanging information with it. With can be illustrated as shown in figure 1.1.

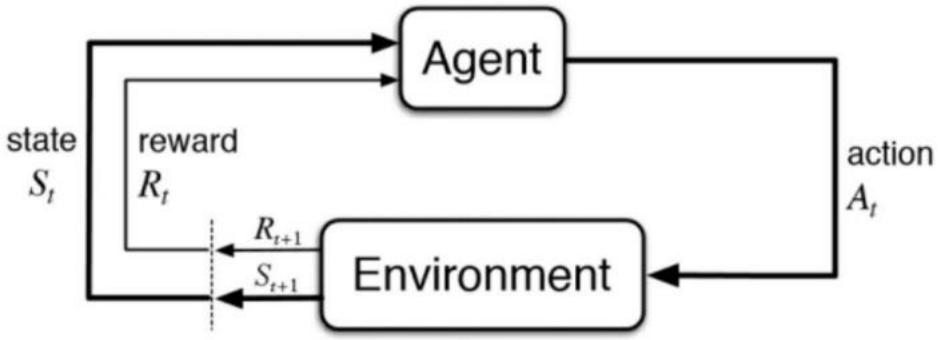


*Figure 5.1 - The Interactions Between Agent and Environment*

As pointed out above, during these interactions, each time an agent performs an action in the environment, it will be assigned with a reward indicating the goodness of the action, plus the information of the next state.

The objective of the task is, find a policy **π** that maximize the overall reward, given the reward information and state observations. Formally:

*Equation 1*

$$\max \pi \mathbb{E}\left[G_t | S_t = s\right]$$

,

where

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

**γ** - Decay Factor; **R** – Reward; **S** - State; **t** - Time variable

The popular Q-Learning algorithm proposed earlier is known to be suitable for solving Markov decision process of an agent quite efficiently and was used a long term of time in the history to address many real-world problems like decision making in complex environment. Based on Bellman Functions [0] of Markov Decision Process, it tries to maintain a table of action values (also known as the Q-values) for every possible state. Given the action value function as the following:

*Equation 2*

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi\left(a'|s'\right) q_\pi\left(s', a'\right)$$

$P^a_{ss`}$ stands for the probability of action **a** transfers state **s** into state **s`**

During the training, update the table by Temporal-Difference method, as result, theoretically, the final table indicates the optimal policy.

However, it is proven that Q-Learning algorithm is potentially going to be overestimate the action values under certain circumstances, was also proposed by Google DeepMind [1]. Furthermore, the practices show the Q-Learning algorithm is also not suitable for those environments where the states and actions have huge number of possible combinations. Issues stated above are very common in real practice, therefore a new technique is needed to address such problems.

With the evolving importance and development of Reinforcement Learning algorithms and Neural Network techniques, these two are being used in parallel to solve the real-world problems. The mathematical approximation ability of the neural network can be used instead of the Q-table mentioned above.

Our implementation also uses Q-Learning algorithm but combining it with deep neural network techniques (Figure 1.2) and many other improvements.

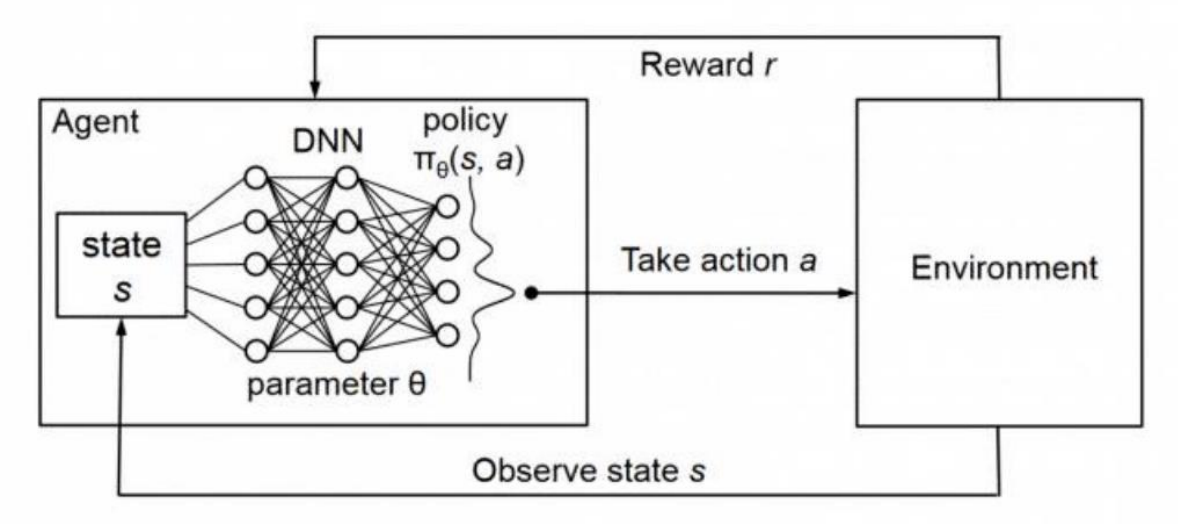


*Figure1. 6 – DQN*

*Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).*

As mentioned above, the game environment used in my case is called “DemonAttack-v0”, of which I have used its simulator from a third-party dependency of machine learning environment called OpenAI-GYM[2], according to the corresponding documentation of OpenAI, the game has observation space of an RGB image with shape 210*160*3, and the action space with six discrete actions. Also, the agent will initially gain 3 lives, if being hit by the enemies, the number of live drops 1, however, when successfully cleaned all enemies in present level of the game, the number lives of the agent will increase 1 as reward, therefore, this environment only returns game done equals true when the agent lost all its lives.

Please also note that, although the state information can be extracted at every single frame of the game, each action taken by the agent is repeatedly performed for a duration of k frames, where k is uniformly sampled from number {2,3,4}. This would apparently bring to the algorithm with some necessary changes, which will be further explained in later sections.

Apart from that, some important modifications are mode to improve the performance, as will be shown in Implementation section.

# 2.Design

Because this project has more of a research focus, this section will contain more information of the methodology design and corresponding concerns instead of User Interaction designs.

Similar to almost all Machine Learning research procedures, the majorities of my implementation can be divided into two parts, training and testing, since the project itself does not imply fancy User Interactions, this structure is clearly divided with separate files.

**File & System Structures**

The files named with starting of "**v3_***.py**" indicate they are training python scripts, v3 means version three, the most stable version with best performance by experiments. During the training process, after each 30,000 iterations, the parameters of the neural networks will be automatically saved as **"e_net.pkl"** and **"t_net.pkl"**, for evaluation network and targeting network respectively.

File of name **"DDQN_test.py"** is the testing python script, which would load the saved "**e_net.pkl"** file and uses its parameter to run the testing. Note that **"t_net.pkl"** is not used by **"DDQN_test.py"**, it is just a backup of the targeting network for develop-time analyze.

The folder named **"parms"** are the network pre-trained parameters provided, with roughly 1,000,000 iterations. Another folder **"logs"** contains data generated automatically by TensorBoard for real-time performance monitoring.

**Convolutional Neural Network Architecture**

As mentioned above, my idea is to combine deep learning techniques with the Q-Learning algorithm, that is, use a neural network as alternative to the Q-table. The network has the following architecture:

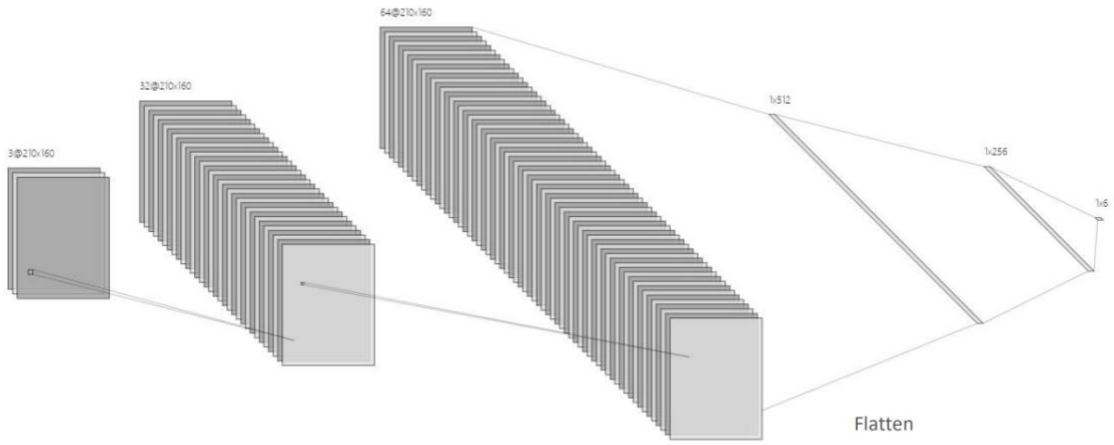


*Figure 2.1 - DQN Architecture*

Above architecture is referenced from GoodMind`s paper [1], with three convolutional layers and two fully connected layers maps to finally six discrete actions. Despite the very last layer, all other layers use activation non-linear function ReLu, the output layer uses linear activation function.

I have used MSE (Mean Square Error Equation 3) as my loss function, because the final output is set to be the numerical action values (Q-Values) of the six actions of shape 1*6, and the network backpropagates with stochastic gradient decent optimization algorithm - Adam.

*Equation 3 - MSE*

$$\mathcal{L}(\theta) = \left(\left[r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}|\theta)\right] - Q(s_t, a_t|\theta)\right)^2$$

However, this approach has a potential problem of so called "bootstrap" [1], as can tell from Equation 3, the network used for prediction and the network used for pretending as the current ground truth based on Temporal-Difference method are identical networks.

This will cause the network to overestimate the action values as what did by Q-Learning algorithm.

As suggested by the name of my algorithm DDQN, two neural networks are used in my algorithm, one is called **Evaluation Network** and is used for Q-Values prediction, another one is called **Targeting Network**, which used for ground truth approximation. The two networks are with the identical architectures as illustrated in Figure 2.1, but with different updating policies. Evaluation Network updates itself by gradient decent method each 4 iterations, due to the action repeating nature stated in section 1. In other hand, the Targeting Network does not perform any backpropagation at all, it instead copies Evaluation Network`s parameters after each 100 times of updates of Evaluation Network. With this being implemented, the bootstrap problem is welly addressed. The Loss Function is this case becomes :

*Equation 4 - MSE with Targeting Network*

$$L(w) = \mathbb{E}[(\underbrace{r + \gamma \max_{a'} Q(s', a', w)}_{Target} - Q(s, a, w))^2]$$

**Memory Buffer & Experience Replay**

In order to improve the reusability and efficiency of the training data, I have implemented a memory buffer to the agent. According to the definition of Reinforcement Learning from Figure 1.1, every time a single action is taken by the agent, an corresponding reward and the next state is then returned by the environment and passed back to the agent, now, instead of directly learn the current state, the agent combines all information within current iteration including the current state, the action taken, the rewards and the next state returned in to an tuple, such a tuple is a form of experience. Formally, the tuple is in the form:

$$<s_t, a_t, r_t, s_{t+1}>$$

This information will be stored into the memory buffer with form of 2-dimentional tuple. As mentioned earlier, the agent using my algorithm learns every 4 frames, while learning, the agent extracts a batch of 32 experiences from the memory buffer and feed it into the neural

network for learning purpose. Similarly, by using memory buffer, the correspondences between the data are weakened.

**Reward Policy**

The reward policy is proven to be very critical with respect to the performance of agent, essentially the idea if the reward is to inform the agent about what is good and what is bad, therefore a good reward policy is important and is a key factor deciding whether the neural network is going to converge, as well as the converging speed.

Different reward policies are being tested out and evaluate the performances, eventually I have come up a reward policy that allows the network to converge. The reward policy of mine and some other tested policies are illustrated as the following:

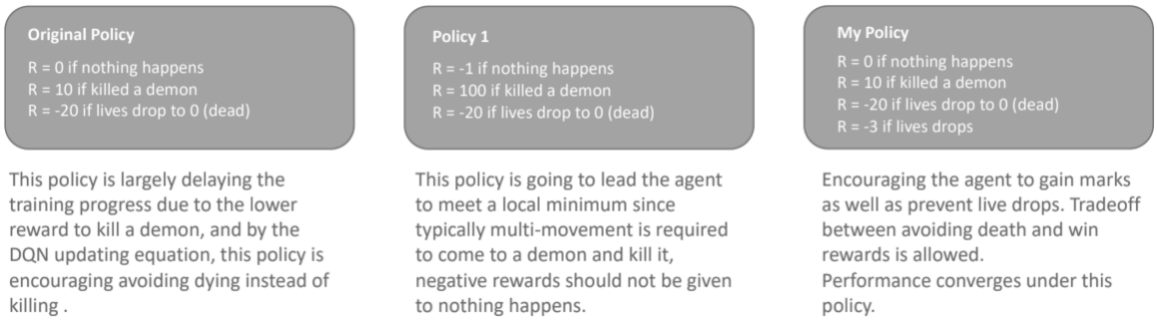


*Figure 2.2 - Reward Policies*

**Experimental Process**

Like agreed widely in the Machine Learning society, the most challenging thing of implementing a good agent or a good neural network is the hyper-parameter tuning process. Therefore, all the hyper-parameters should set to be flexible, critical parameters including:

· *Epsilon – Greedy Rate*
· *GAMMA (Decay Rate)*
· *Learning Rate*
· *Capacity of memory buffer*
· *Number of Iteration gaps for the Targeting net to copy the parameters of Evaluation net*
· *Training Episodes*

The plan is to try out using different settings of the parameters and monitor the real-time performance of each different settings.

To ensure the reproducibility, once the training is done, the corresponding generated network parameters will be used for testing many iterations. However, I must point out here, regardless of the general algorithm steps of the DDQN Reinforcement Learning algorithm, the actual performance of the agent is largely dependent on the hyper-parameters tuning and would be different depend on different environment. Despite of that, during the research, some universal properties of the hyper-parameters would still be observed and will be concluded in later section.

# 3.Implementation

The whole implementations are not going very well if just stick to the previous mentioned designs, in this section, I will first introduce my complete DDQN algorithm and some idea behind it, then, you will see some challenges that I have encountered, the potential causes of those problems and the corresponding attempted solutions.

**Training – Initializations**

Firstly, contract the neural network as mentioned in the design section, in my implementation, most of the deep learning part are being implemented based on Pytorch Deep Learning dependencies.

```
class DQN_network(nn.Module):


  def __init__(self):


    super(DQN_network, self).__init__()


    self.model = nn.Sequential(
      # first layer inputsize = 250
      nn.Conv2d(channels,32,(8,8),(4,4)),
      nn.ReLU(),
      # second layer inputsize = 250
      nn.Conv2d(32,64,(4,4),(2,2)),
      nn.ReLU(),
      # fourth layer inputsize = 62
      nn.Conv2d(64,64,(3,3)),
      nn.ReLU(),
      # FC layer input features =  64 * 30 * 30 = 576000
      nn.Flatten(),
      nn.Linear(3136,512),
      nn.ReLU(),
      nn.Linear(512,256),
      nn.ReLU(),
      nn.Linear(256,N_actions),
    )
```

Furthermore, in the beginning of the program, all the hyper-parameters mentioned in the design section are all set to be variables for the convenience of later tuning.

Likewise, in this stage, for the purpose of real-time monitoring, the Summary Writer of the TensorBoard is also being set up. It will be recording the real-time performance of the agent for each iterations of both training and testing, in the form of Rewards against Iterations.

**Training – Build the Agent**

Benefited from the idea of Object-Oriented Programming, construct the agent as a class, which has two internal neural networks, a memory buffer of python array, and some variable flags recording the iteration counts. According to the design, the agent is capable of doing the following actions:

1. Pro-process the Image (img_preprocessing)
2. Store Experience (store_trans)
3. Take an Action (choose_action)
4. Learning from the Experience Buffer (learn)

Each of these actions are functions defined within the agent class, and the name of the implemented function is given in underlining form inside the brackets.

The img_preprocessing function takes an RGB image as input and convert it into 4-dimontinal tensors of shape (Batch*Channel*Height *Weight). The store_trans function takes an state image, an action taken, a reward value, and frame of next state as parameters, make them as a tuple of five, and store into the memory buffer, note that here if the memory buffer is full, the oldest memory will be removed from the buffer. The choose_action function takes an state as parameter, then according to the Epsilon variable, decide whether to choose an random action or choose action based on what suggested by the returning Q-values of Evaluation network, it returns the action as a numerical action index. The learn function takes no parameters, when it is called, randomly samples 32 experiences from the memory buffer, split the information of sampled data, pass the states of 32 samples into the targeting network, calculate the loss according to the Equation 4, finally, backpropagate the loss function and performs gradient decent to updating the weights of Evaluation network. Note that the learn function will also check if the parameters of Targeting network needs renew by coping parameters of Evaluation network according to counter variable.

**Training – DDQN Algorithm**

Within each episode, initialize the GYM environment, and epsilon rewards. For each frame of iteration, take an action by calling the choose_action function of the agent, alter the reward according to the reward policy (Figure 2.2), then store the information of current frame of paying into the memory buffer. Moreover, after each 4 frames of iterations, plus if the memory buffer has more than 20,000 experience, the algorithm will call the learn function of the agent. Note that the agent only learns the experiences after 20,000 experiences were collected, this would largely speed up the average training time.

The Pseudocode of the general DDQN algorithm I am using is shown in Figure 3.1 [3].

**Algorithm 1** Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
**for** episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  **for** $t = 1, T$ **do**
    With probability $\epsilon$ select a random action $a_t$
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  **end for**
**end for**

*Figure 7.1 - Pseudocode of General DDQN*

**Testing**

Testing is not hard in my implementation, which uses the same network architectures as the training step, and the same definitions of the agent as what used in the training process. The differences is, during testing, no learning process is needed, the agent directly choose action only based on the Evaluation network`s parameters saved from the training process, without any random action. In the meantime, the real-time performance of the agent will also be captured by the TensorBoard.

---

Up to this point, the implementation of the algorithm is almost done, however, by the experiments, the agent seems not to be converging with the current algorithm, I performed a quick evaluation by the data recorded by TensorBoard on the performance of the agent, and obtained the following figure:

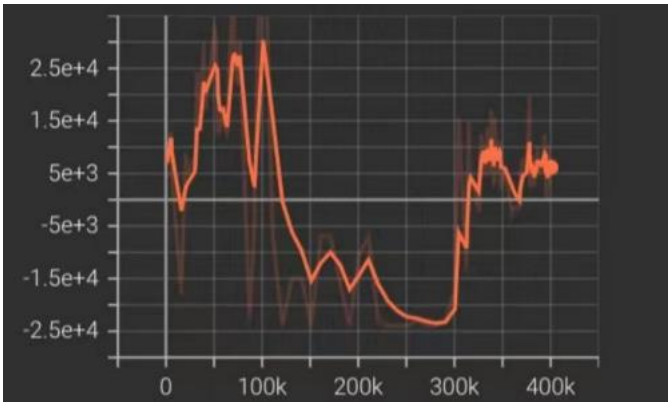


*Figure 3.2 – Performance for Now*

It is clear that the agent is not currently learning, the issues was assumed in the beginning with the hyper-parameters, but by altering them makes the graph looks worse. As mentioned

earlier, the general Reinforcement Learning algorithm are not 100% suitable in all cases, the following issues was found during the later implementation checks.

**Issue – Insufficient Frame of Playing Passed into CNN**

As suggested by the title, the above designs and implementations only considering store a single frame of state into the memory buffer, hence a single frame is passed into the Convolutional Neural Network, the backpropagations are also dependent on that.

This could be problematic, consider about the game "DemonAttack-v0", almost all the instances within the game are moving, including the agent itself, the bullets, the enemies, therefore a single frame of input is not sufficient for the CNN to extract movement information, like speed, or accelerations of the instances. Alternatively, in my implementation, capture the 4 consecutive frames and combine them by a frame buffer of size four, as shown in Figure 3.3.

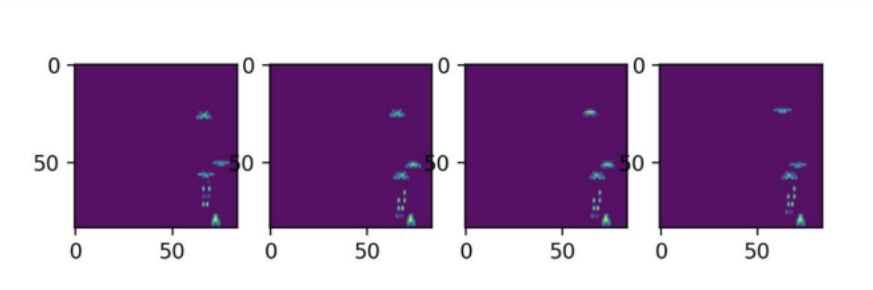


*Figure 3.3 Consecutive Frames Capturing*

If the buffer is full, the oldest frame will be removed from the buffer to allowing new frame to come. Eventually, during the memory storing, store the whole frame buffer into the memory. Other than that, to improve the quality of the data, the typical image preprocessing steps are also within design. Figure 3.4 illustrates the whole image pre-processing steps implementation.

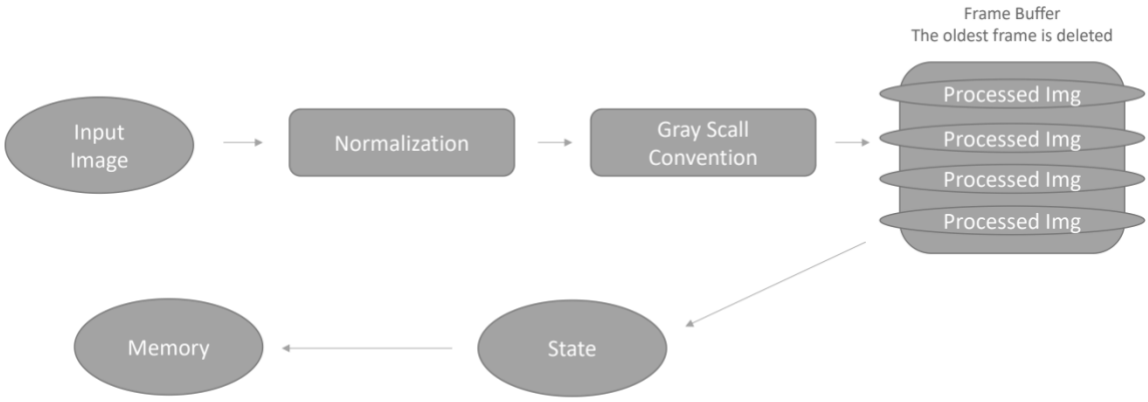


*Figure 3.4 - Image Pre-Processing*

I have provided two versions of the algorithm, one with single frame passing into the neural network, one with four consecutive frames passing into the neural network, named "v3_singleFrame.py"and "v3_4frames.py", respectively.

**Issue – Epsilon Greedy Policy**

The purpose of using Epsilon Greedy Policy is for the agent to explore the environment randomly, this factor was meant to be a small value, it therefore been set to be fixed at the beginning. Problem shows up quickly, the agent trained under the fixed Epsilon Greedy Policy does not tend to perform well, it is observed that the agent is essentially following a local minimal solution in this case, that is, instead of trying to gain the reward, it ties to not losing any reward. This policy in short-term might be optimal, however it does not in long-term view. This issue was later solved by altering the Epsilon variable during the training iteration.

For the beginning of training, set the Epsilon-Greedy variable to 1.0, to allow the agent to explore the environment completely randomly, and collecting experiences. Then when the learning condition is reached, set the Epsilon variable to 0.28, still maintains some randomness, and decrease by -0.01 after each iteration, until reaches the minimum training Epsilon value 0.15.

With this being implemented, the agent started to learn, as will be shown in next section.

# 4.Testing & Evaluation

After the implementation, several testing and evaluations are carried out, I have put the agent under current algorithm to execute for more than 700000 iterations and obtained the following result:

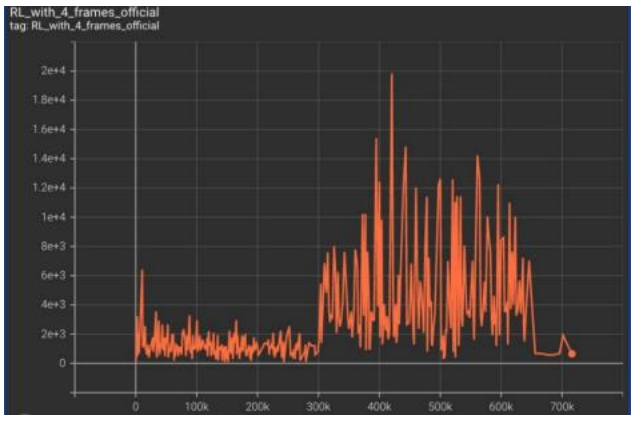


*Figure 8.1 - Training Performance*

Above data was collected automatically by TensorBoard to record the performance of the agent during training process, with its rewards against training iterations.

I have then used the trained agent to play the game 15 rounds, and record the round rewards against round count, the following result is obtained:

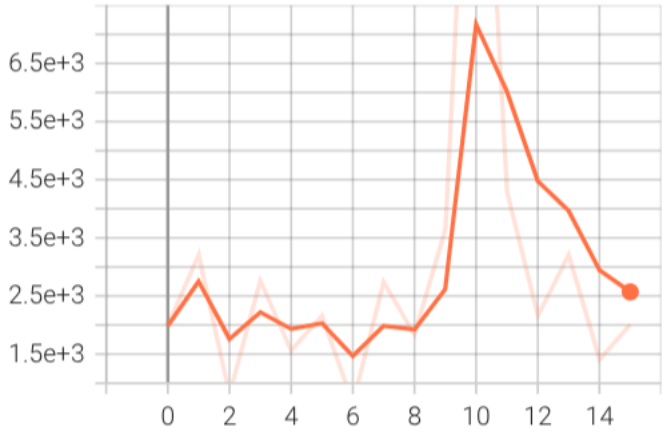


*Figure 4.2 – Testing Rewards*

It is clear that the agent is now performing with some sense of learning, because the device I am currently using is does not support CUDA GUP acceleration for training, plus the limitation of CUP and Memory conditions, training with more iterations is not realistic, with this result, however, we can tell that the agent is learning.

A demonstration video made for thee user instructions and demonstrations of trained AI playing “DemonAttac-v0” can be found in Appendix.

# 5.Conclusion

As mentioned in the introduction section, the purpose of the project is mainly about solving the Sequential Decision problems. In my implementation, some Reinforcement Learning techniques combining with Deep Learning Mechanisms are being carried out to solve the problem. While the agent is interacting with its environment, we take actions based on Epsilon Greedy policy, with some degree of randomness to take random actions, alternatively the agent takes actions from convolutional neural networks according to the current state. Moreover, two convolutional neural networks are used in parallel to perform the Q-value prediction and ground-truth approximation according to Temporal-Difference method. During the interactions between the agent and environment, the current state of four consecutive recent frames and other corresponding state information will be stored into memory buffer, after each 4 iterations, the agent will extract some experiences from the memory buffer and feed it to the one of the networks namely Targeting network and calculate the error between the results and the memory record by Mean Square Error and perform gradient decent with respect to Evaluation network. The targeting network copies the parameters of the Evaluation network after each 100 times of Learning.

The approach to train an agent to play “DemonAttck-v0” is mostly finished, with the agent plays the game of averagely 4.5k rewards gained under the current reward policy. However, there are still space of improvements, due to the limitations of device, more iterations of training did not come out, according to Google DeepMind [1] where they have tried some DDQN algorithms on several Atari games as well, they have normally trained the agent more

than 1000000 Episodes, more iterations of training certainly related to the performance of the agent.

Other than that, since the deep neural network I am currently using has significant numbers of layers, despite some batch normalizations are already carried out, there are still potential problem of Network degradation [4], therefore, a Deep Residual Network can be used as instead of ordinary Convolutional Neural Network to address the problem. Furthermore, a better version of the deep double Q-Learning algorithm namely the Dueling Deep Double Q-Learning could also be implemented to overcome the potential overfitting issue of the current network.

Also, worth to point out here that because of the particularity of the current algorithm, many changes are made specifically to the game "DemonAttak-v0", the algorithm is not applied to other Atari games.

# 6.BCS Project Criteria & Self-Reflection

In this section, I would explain in below how my research project satisfies the BCS project Criteria and talk about some self-reflections during the implementation of the project.

**• An ability to apply practical and analytical skills gained during the degree program.**

The COMP338 – Computer Vison, COMP305-Biocomputation, ELEC319-Image Processing, and some other programmes I studies last year gains knowledge of general approaches and interpretation of artificial intelligence. Moreover, some modules of second semester of year 3 study including ELEC320-Neural Networks, COMP341-Robot perception and manipulation provided more academic support for implementing the Convolutional Neural Networks, as well as understand the idea and mathematical principals, those largely helped me during the implementation process and the related paper reading.

**• Innovation and/or creativity.**

The Reinforcement Learning algorithms and its fundamental idea is roughly mentioned in COMP341, however, as stated in earlier sections, although the Reinforcement Learning algorithm steps like DDQN has some general guidelines of steps, the performance of the final outcome is largely depend on the parameters tuning and modification according to different environment, in my project, while following the general algorithm process of DDQN, some important modifications to the input images and hyper-parameters are considered to improve the performance of the agent, please refer back to implementation section for more details about this. That reflexes the Innovation and creativity ability required by the BCS project criteria.

**• Synthesis of information, ideas and practices to provide a quality solution together with an evaluation of that solution.**

This part refers to the Background & Design & Implementation Summary and the Evaluation section, which includes my idea of combining techniques to achieve the project requirements. Please refer to the above sections for more information.

• **That your project meets a real need in a wider context.**

An intelligent agent that can performs high-level analyzable information and choose reasonable action is needed nowadays in many fields that requires autonomous systems.

• **An ability to self-manage a significant piece of work.**

The source code would be stored and maintained in both my local PC and Github, for code history managing, however, I have to point out here, since this project have more of research focus, many versions of the code were implemented to find out the best solution, plus the Reinforcement Learning itself is an Black-Box problem, it is not easy to fully understand what is happing insider the box, the only thing I can do to locate problems are trying different paths, therefore the versions of the code is not well managed, causes a lot problems during the implementation stage.

• **Critical self-evaluation of the process.**

Please refer to the Testing & Evaluation section for the details of Evaluation on the project itself, here I want to mention some evaluations for my research skills, as mentioned above, although the Github is used for the project for version control purpose, some improvement would also be feasible, for example, a nice command line user interface could be implemented to allow more clear and manageable parameter tuning. Furthermore, this project is an important lesson to me for the future research activities for realizing the importance of research notes, I have not put any notes during the research process, despite I have learned so many things during the implementation. but it is better to do so, not just for the need of research report, also for me to memories more information regarding to the project.

# 7.References


[0] https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7

[1] Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning." Proceedings of the AAAI conference on artificial intelligence. Vol. 30. No. 1. 2016.

[2] https://gym.openai.com/

[3] Jafari, Reza & Javidi, M.M. & Kuchaki Rafsanjani, Marjan. (2019). Using deep reinforcement learning approach for solving the multiple sequence alignment problem. SN Applied Sciences. 1. 10.1007/s42452-019-0611-4.

Baird. Residual algorithms: Reinforcement learning with function approximation. In Machine Learning: Proceedings of the Twelfth International Conference, pages 30–37, 1995.

Richard S.Sutton and Andrew G. Barto, 2014,2015, Reinforcement Learning : An Introduction. The MIT Press.

Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. Machine learning, 8(3):293– 321, 1992.

Lin, Zichuan, et al. "Episodic memory deep q-networks." arXiv preprint arXiv:1805.07603 (2018).

Baird. Residual algorithms: Reinforcement learning with function approximation. In Machine Learning: Proceedings of the Twelfth International Conference, pages 30–37, 1995.
Richard S.Sutton and Andrew G. Barto, 2014,2015, Reinforcement Learning : AnIntroduction. The MIT Press.
Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. Machine learning, 8(3):293– 321, 1992.
Lin, Zichuan, et al. "Episodic memory deep q-networks." arXiv preprint arXiv:1805.07603 (2018).
Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep reinforcement learningwith double q-learning." Proceedings of the AAAI conference on artificial intelligence. Vol. 30. No. 1. 2016.

# 8.Appendices

Github Source Code:
https://github.com/HuskyKingdom/self-works/tree/main/Artifitial%20Intelligent/Reinforcement_Learning

User Instructions & Demonstrations:
https://liverpool.instructuremedia.com/embed/6326b3ba-abd8-4435-914a-e89ae9d296df