CS5079 Assessment 1: Reinforcement Learning with OpenAI Gym on Atari Game (Asterix)

Aleksandar Stefanov, Radmil Raychev, Hasan Ahmedov

Department of Computing Science

December 10, 2021

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed: Aleksandar Stefanov, Radmil Raychev, Hasan Ahmedov

Date: December 10, 2021

Contents

Chapter 1

Introduction

Atari 2600 also known as the Atari Video Computer System is a home video game console developed to play arcade video games. It uses ROM cartridges which are essentially memory cards containing ROM to be used for loading and running software such as video games.

Asterix, as seen in Figure 1.1, is a game where a player guides a character between stage lines to accumulate rewards. The character has 3 lives that are gradually decreased as they collide with lasers. The player dies when they have no more lives.

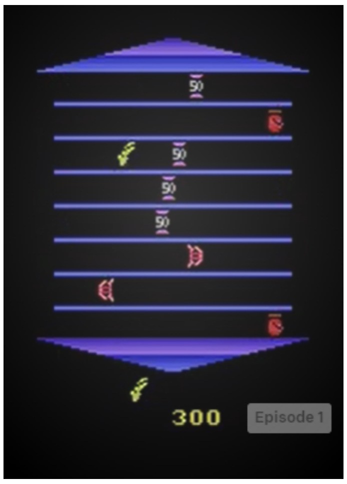


Figure 1.1: Sample Atari Asterix game screenshot.

The idea is to teach an intelligent agent to play Asterix by using an easy progress metric, such as game score, to optimize against. The game score can rise up when the agent takes moves that reward them, and fall down when the agent makes wrong decisions. We will measure the intelligence of the agent by using Reinforcement Learning (RL) within a Deep Neural Network (DNN) that takes advantage of the Q-learning algorithm.

The aim of this report is the application of a RL algorithm to teach the character how to play the game. This is done using different approaches for learning from the available input information such as screen frames or pixels, RAM memory, and the combination of the two. Furthermore, these approaches will be further optimised.

Chapter 2

Methodology and Results

2.1 Tasks specification

2.1.1 Environment Introduction *(Task 1.1 and Task 2.1)*

In order to achieve a performant model, this paper proposes a number of optimisation techniques that can be easily reproduced and applied to different Atari 2600 games. Initial analyses of the possible observations, action spaces, rewards, info dictionaries, and episodes for both the RAM and image models are critical. A plot is created showing what the agent perceives when images are used, and what the agent finds as valuable information when RAM is utilized (please refer to the corresponding notebooks for these plots). This paper uses a batch of 32 as its sample size.

- **Observations**: RGB image which is an array of shape $210 \times 160 \times 3$ indicating a height of pixels, a width of pixels and channels.

- **Action space**: Discrete with 9 possible actions — *"NOOP", "UP", "RIGHT", "LEFT", "DOWN", "UP-RIGHT", "UPLEFT", "DOWNRIGHT" and "DOWNLEFT".* Each action is performed repeatedly for a duration of k frames, where k is sampled from the set 2, 3, 4 uniformly. The discrete space allows for a fixed range of non-negative numbers within a grid in the $[0, 255]$ range

- **Reward**: 0 but is accumulated in the process of the game-play

- **Info dictionary**: it contains the value of the lives left. In the game, a player has only 3 lives that decrease as he collides with the lyres

- **Episode**: An episode is concluded when the player loses all his lives (3).

As a means of achieving a deterministic game environment, we applied a seeding function (with a value of 1337) to the environment. This ensured the reproducibility of the experiment results.

2.1.2 Q-learning algorithm applications *(Task 1.2)*

Q-learning is an RL technique used to determine the optimal policy in Markov Decision Processes (MDP). It uses a Q-table which is a $m \times n$ lookup table, where $m = states$ and $n = actions$, used for the calculation of the maximum expected future reward for action at each state.

Each score in the Q-table will be the maximum expected future reward for that specific action at that specific state. To calculate these values, we use the Q-function depicted in Figure 2.1. Under a given policy, the true value of action a in a state s is π where *gamma* is in the range $[0, 1]$ is a discount factor that performs the immediate and later rewards importance trade-off.

$$\underbrace{Q^{\pi}(s_t, a_t)}_{\text{Q value for that state given that action}} = \underline{E}\underbrace{[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots}_{\text{Expected discounted cumulative reward ...}}\underbrace{|s_t, a_t]}_{\text{given that state and that action}}$$

Figure 2.1: Q-Function. (source: link)

The Q-function can identify an optimal action-selection policy given any finite Markov decision process and return the expected rewards for an action taken in a given state. It is estimated using Q-learning which iteratively updates $Q(state, action)$ using the Bellman Equation.

An epsilon greedy strategy is implemented in this paper in order to ensure the balance between exploring states and exploiting results, shown in Figure 2.2.

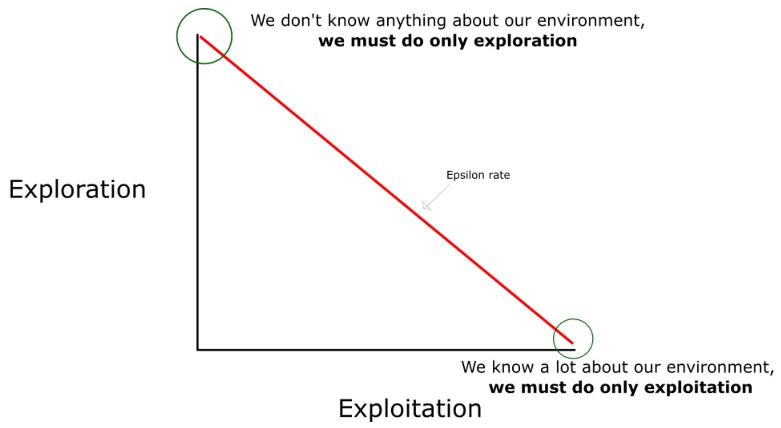

We don't know anything about our environment,
we must do only exploration
Exploration
Epsilon rate
We know a lot about our environment,
we must do only exploitation
Exploitation

Figure 2.2: Epsilon rate change description. (source: link)

2.1.3 Preprocessing Procedures *(Task 1.3 and Task 2.1 contd.)*

• Image Preprocessing

Working with raw Atari frames of shape $210x160x3$ can be very computationally demanding. The first part of the methodology manages simple image preprocessing, such as the conversion of the RGB pictorial input to grey-scale. An image that only contains shades of grey is an essential method to speed up the process of training, as less information needs to be provided for each pixel. The grey-scale intensity is stored as an 8-bit integer value with 256 possible shades of grey ranging from white to black [1]. Following is the downsampling of the frames to a final input representation of the size $84x84$ which is the optimal capture of the area we want to concentrate on, or essentially the main playing area.

• Ram Preprocessing

When using the RAM version of the game, we now only have a one-dimensional array of size 128 where the 0 corresponds to black and higher values are brighter colours. The one-dimensional array is reshaped when stacking is applied (detailed later). We apply normalisation by dividing the inputs by 255 setting the highest value to be 1.

• Preprocessing Techniques used on all the inputs

- **Scaling** - usually, the lowest and highest value for the observational space is 0 and 255 respectively. For faster execution, this paper also implements a scaling technique employed directly on the observations, where the observation space can have a minimum value of 0 and a maximum value of 1.

- **Frame Skipping** - another common preprocessing step is the introduction of frame skipping [2], which is what restricts the agent's decision points by repeating some selected action for k consecutive frames, making the RL problem simpler and speeding up execution.

- **Frame Stacking** - this methodology follows an approach where the agent is designed with a richer observation space by combining past frames with the most recent ones. This is commonly referred to as frame stacking [3] which enables the agent to detect the direction of motion of in-game objects. This paper has chosen to use 5 frames, but the algorithm might also be robust with different values for k such as $k = 3$ or $k = 4$. This research experimented with all 3 options and concluded that the use of 5 images in a stack is the most beneficial approach with regards to total reward.

- **The *NOOP* Action** several preprocessing experiments using the *NOOP* action are conducted in this paper. This is to introduce variety in the initial game states. However, it was quickly concluded that this is detrimental to the learning process as the agent can get killed immediately or lose time doing nothing. Therefore, the experiments were kept and analysed, but the final model has its *NOOP* function omitted.

2.1.4 The Neural Networks *(Task 1.4, Task 2.2 and Task 3.1)*

Five Double Deep Q Networks (DDQN) are designed after thorough testing and hyper-parameter optimization.

• Image Network

The architecture in the first task begins with an initial hidden layer that convolves 32 filters of 88 with stride 4 with the input image and applies rectifier non-linearity. The second hidden layer convolves 64 filters of $4x4$ with stride 2, again followed by rectifier non-linearity. This is followed by a third convolutional layer that convolves 64 filters of $3x3$ with stride 1 followed by a rectifier. The final two hidden layers are fully-connected and consist of 512 and 256 rectifier units respectively. The output layer is fully-connected with a single output for each valid action. The more performant model chosen has $4,179,113$ trainable parameters and its architecture is depicted in Appendix A Figure A.1. [3] follow a similar structure but with fewer layers.

• RAM Network

The second task has implemented 2 different DDQNs. The tests include a neural network with either 2 or 4 Dense layers, respectively. Each one contains 128 rectifier units. In this configuration, the last layer is the output layer. The chosen DDQN takes as an input stacked RAM of size $128 \times$ (frame stack size) $= 640$. It uses 4 Dense layers and also has $99,721$ trainable parameters as opposed to the $132,745$ trainable parameters of the other RAM model. Both DDQN architectures are depicted in Appendix A Figure A.2 and A.3 respectively.

• Mixed Network

Lastly, there are two architectures implemented for the mixed input DDQN, shifting from the previously applied sequential approach to a functional configuration. This is necessary when handling models that have non-linear topology, shared layers or multiple inputs/outputs. In the first model, we first feed the image input into the DDQN convolutional layers, which are the first three. Then we flatten the convolved tensor into one dimension. We then feed the RAM input into two Dense layers with 512 and 256 rectifier units respectively. Lastly, concatenation is performed between the flattened convolutional layer outputs and the Dense layer outputs. We then pass the concatenated tensor to the last Dense output layer. This is the chosen final model with $611,561$ trainable parameters. The difference in the second configuration is that the concatenation is performed directly between the RAM state input and the flattened convolutional layer output, leaving us with $4,506,793$ trainable parameters. For further explanation, the paper has provided pictorial evidence of the difference of the models in Appendix A, Figures A.4 and A.5.

Ultimately, for the mixed DDQN training, we adopt and step through two Atari game environments simultaneously. One of them yields the environment state as images and the other — as RAM states. To guarantee that each action taken by the agent produces the same result in both environments, we seed them with the same seed value (1337). Also, after thorough experimentation, this paper can conclude that simple architectures with a lower number of trainable parameters yield faster convergence and better runtime.

- **Neural Network Parameters**
    - **Optimizer** - adaptive gradient methods like Adam have proven to outperform RMSProp, but it is a fact that Adam can be sensitive to hyper-parameter tuning [4]. However, this paper takes advantage of Adam's ability to combine the heuristics of both SGD's Momentum and RMSProp. The proposed paper uses the value of 0.00025 for the learning rate of the model, which is a shift from the default 0.0001. Learning rates of higher value correspond to faster movement in the parameter space, making learning faster.
    - **Loss Function** - Huber loss is the loss function that is highly suited for relevant problems where the rewards can be impacted by the training environment. As lasers spawn from different places, this means that there is a probability the agent sometimes collects large rewards or very low rewards due to being lucky. Huber loss is also incredibly robust to the presence of outliers.
    - **Kernel Initializer** - this paper proposes the utility of using HeUniform to initialise the network weights as it works very good with the adopted ReLu activation function [5]. HeUniform draws samples from a uniform distribution within the range $[-limit, limit]$. The limit equals $limit = \sqrt{\frac{6}{fan\_in}}$ where $fan\_in =$ number of input units in the weight tensor [6].

2.1.5 Agent *(Task 1.4 contd.)*

The agents used in the tasks are based on the same techniques as presented below.

- **Double Deep Q Network**

One of the main problems in the DQN algorithm is the overestimation of the Q function value, due to the using of *max* function used in the target setting formula:

$$Q(s,a) \rightarrow r + \gamma \max_a Q\left(s', a\right)$$

This problem is solved by the introduction of the concept of double learning [7], which has two independently learned Q functions, Q_1 and Q_2. These values are randomly updated with one of the two formulas:

$$Q_1(s,a) \rightarrow r + \gamma Q_2\left(s', \operatorname{argmax}_a Q_1\left(s', a\right)\right)$$

$$Q_2(s,a) \rightarrow r + \gamma Q_1\left(s', \operatorname{argmax}_a Q_2\left(s', a\right)\right)$$

This is how maximisation bias is attempted to be eliminated and why this paper has chosen this approach for the execution of the training tasks [7].

Initially, this paper attempted to solve Task 1.4 using double-ended queue (deque) memory without priority, which proved to be inefficient and is omitted. Therefore, we introduce the prioritized experience replay strategy to change the sampling distribution by taking into account the experience priorities[8]. We use the leverage of treating all samples in a different manner, which as a sequence helps the model learn from transitions it considers more important. We define the error of a given sample $S = (s,a,r,s')$ as the distance between the output of the Q value prediction and its target $T(S)$: error $= |Q(s,a) - T(S)|$. The agent memory stores this error (as an experience priority) and gets updated with each learning step. The error is simply the reward given in a sample which gives us the summarised error formula:

$$\text{error } = |Q(s,a) - T(S)| = \left|Q(s,a) - r - \gamma\tilde{Q}\left(s', \operatorname{argmax}_a Q\left(s',a\right)\right)\right| = |r|$$

where $\tilde{Q}$ is our target network Q value estimate.

- **Agent's Memory**

A simple approach towards storing and sampling past observations would be to store states in an array and sort them by their priorities. However, the space-time complexity would be highly inefficient $O(nlogn)$ for insertion and $O(n)$ for sampling so we shift to a different method. We use the concept of an unsorted sum tree [8] which is a binary tree where the value of the parent comes from the sum of its children. This data structure can perform sampling in $O(logn)$ and insert and update in $O(1)$. In order to test the performance of the image, ram, and mixed input agents in the Asterix Atari game environment an Experience Replay Memory (ERM) based on the sum tree technique is utilised for all the DDQNs.

- **Legacy**

A penalisation policy was implemented in order to increase the performance of the model (distinguish between doing nothing and dying). However, no significant increase in performance was noticed therefore it is not added to the final version of the agents.

2.1.6 Network Training and Evaluation *(Task 1.5, Task 2.3 and 3.2)*

Please refer to the bullets below for all the employed preprocessing techniques that are common for all the RL parameter combinations:

- Image is **converted to grayscale**.
- **Input downsampling** and **Frame Stacking** - final input size: $84 \times 84 \times 5$ (height $\times$ width $\times$ stack size).
- **Frame skipping** implemented ($k = 4$).
- Observation space. **scaling** to have minimum value of 0 and maximum of 1.

- **Image Network**

The only hyperparameter that is tuned is the **NOOP** action (The agent either skipped the first 10 frames or did not skip any frames at all). At first when this experiment was run *"env.render()"* was used. However, after running the same code without rendering the environment a significant increase in the performance of the agent was noted. Therefore, all the consecutive experiments are carried out without rendering the environment.

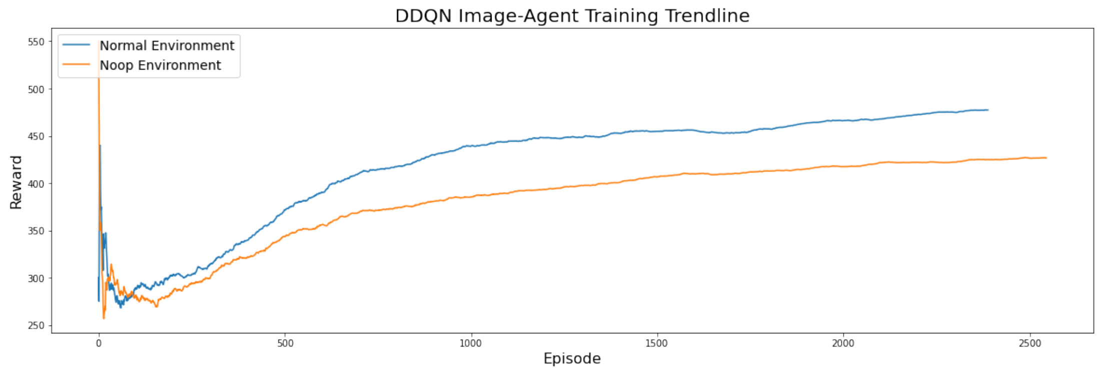


Figure 2.3: Image agents mean reward per episode.

Figure 2.3 demonstrates the behaviour of both agents as the number of episodes increases. It can be observed that the agent who could make use of the first 10 frames after the environment reset (477.3 mean score) performed significantly better than the one who could not (426.6 mean score). Even though the first agent outperformed its competitor with regards to the average mean, both agents eventually achieved a high score of 1350. Despite the fact that agent "freezing" can be beneficial in some environments, Figure 2.3 depicts that the initial frames of Asterix environment are crucial and skipping the first 10 steps is a hindrance to the agent. Please refer to the notebook *"Plots"* or Appendix B for additional plots that compare these two models.

- **RAM Network**

Although the previous experiment proved that "freezing" the agent for the first couple of inputs is an obstacle to the image agent, it is reintroduced to estimate this assumption in the RAM state environment. As noted in subsection 2.1.4, RAM models require fewer parameters (and less training time) than image ones. Therefore, in this case, a total of 4 experiments are carried out.

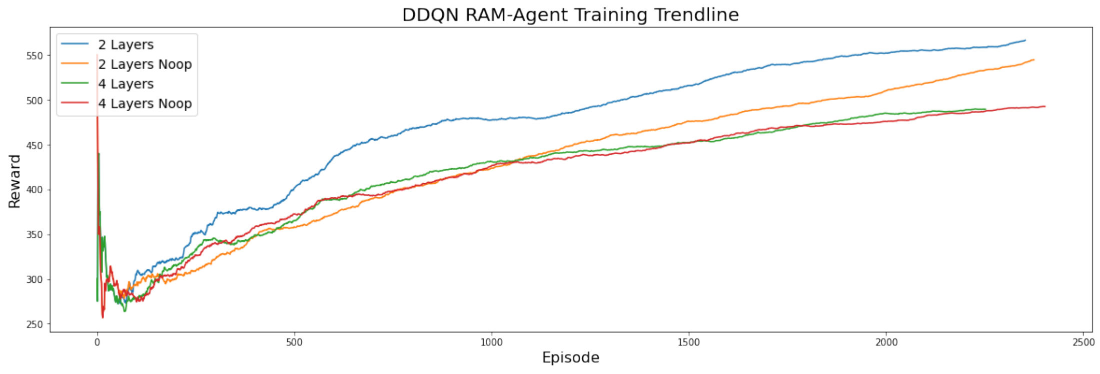


Figure 2.4: Ram agents mean reward per episode.

On Figure 2.4 it is shown that the agent that makes no use of "freezing" and has only 2 layers in its DNN performs the best. It achieved the highest average mean score with regards to total episodes of 566.5. It also scored a high score of $2,000$ once. Once again, it can be concluded that the agent that learns on Asterix-ram environment requires initial frames in order to produce better results for the span of 1 million episodes. *Plots* notebook and the Appendix B contain additional graphs and plots that show each of the RAM DDQN agent performances.

- **Mixed Network**

As it is already concluded that "freezing" the agent obstructs both RAM and image agents. Therefore, no further experimentation with NOOP was conducted. In order to verify the process of concatenating both types of data two techniques are employed (refer to Subsection 2.1.4). The first one concatenates the inputs in the middle of the DDQN and the second one concatenates them in the end.

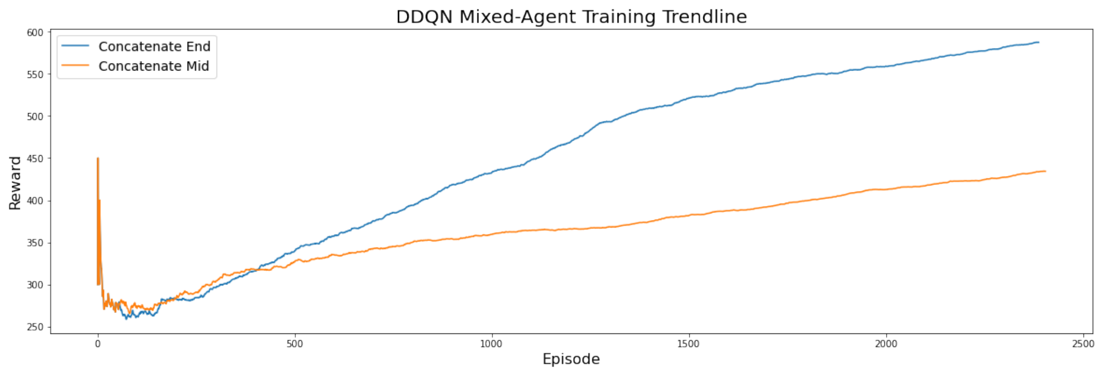


Figure 2.5: Mixed agents mean reward per episode.

Figure 2.5 illustrates the difference in the performance of the above-mentioned models. Around 500*th* episode the first model starts to significantly dominate the other one in regards to average reward per episode. The final mean and highest score of the former are 587.2 and $1,750$, respectively. This discrepancy is due to the larger amount of parameters in the second model and the slower convergence rate of the second DDQN. *Plots* notebook and the Appendix B include extra graphs and plots for each mixed model.

2.1.7 Results Comparison *(Task 2.4 and Task 3.3)*

Metric	Img_1	Img_2	RAM_1	RAM_2	RAM_3	RAM_4	Mix_1	Mix_2
Total Games	2,389	2,546	2,354	2,374	2,253	2,404	2,389	2,406
Total Score	1,140,200	1,086,100	1,333,500	1,294,000	1,1026,00	1,183,700	1,402,800	1,044,950
Average Score	477.27	426.59	566.48	544.61	489.39	492.39	587.19	434.31
High Score	1,350	1,350	1,650	1,750	2,000	1,750	1,800	1,350

Table 2.1: All model performances comparison.

On Table 2.1 performances of the image and RAM agents can be observed. The agent/network combinations are evaluated on four different metrics. In regards to the total games played RAM_3 (4 layers no NOOP) played the least amount of games - therefore stayed alive the most. A trend that can be observed is that the first three RAM agents played fewer games than both of the image agents. RAM_1 (2 layers no NOOP) achieved the highest overall score (1.3 million). Again, three of the RAM agents achieved higher scores than both of the image agents. When the Average Score per Episode is taken into account RAM agents fully dominate image ones. It can be deduced that in the current experimental settings the average reward is inversely proportional to the number of Dense layers in the RAM DDQN architecture. Lastly, RAM_3 is the only agent that has achieved a high score of $2,000$. Maintaining the trend from the previous metric none of the image agents achieved a high score that is better than the high scores achieved by the RAM agents.

Moreover, Table 2.1 can be also used to draw an overall conclusion about all agent performances (image, RAM and mixed). The agent that plays the least amount of games for 1 million episodes is again RAM_3 (4 layers no NOOP). It is followed by the other two RAM agents, therefore agents that learn from RAM tend to die less. Mixed agents play either equal or the same amount of games as agents that learn from image data. With respect to the total reward collected for the period Mix_1 (Concatenate End) is significantly superior to all its competitors. It is followed by three of the RAM agents. The second mixed agent accumulate the least amount of total reward due to its slow convergence rate. The DDQN agent that achieve the best mean score for the corresponding played games is Mix_1 (587.19). It is followed by the RAM models. The DDQNs that perform the worst are Img_2 (no NOOP) and Mix_2 (Concatenate Mid). Finally, RAM_2 is still the model that achieves the highest score $(2,000)$. Mix_1 is placed second (gathered $1,800$ rewards per episode). Both image agents and the Mix_2 agent achieve the highest score of $1,350$ which is the lowest among all the agents.

Chapter 3

Conclusion and Future Work

The evident conclusion to be drawn is that image agents are dominated by both RAM agents and mixed agents on all the recorded metrics. Mixed agent that uses DDQN that concatenates the inputs in the middle dies significantly less than all its competitors and therefore accumulates the highest overall and mean score for 1 million episodes. RAM_3 Agent achieved the highest score among all the 8 agents of $2,000$ points. According to the plots, neither of the agents converged. This paper can also suggest a wide variety of possible future improvements to the proposed models. Due to time constraints, the following list of improvements has been theoretically reviewed but not implemented:

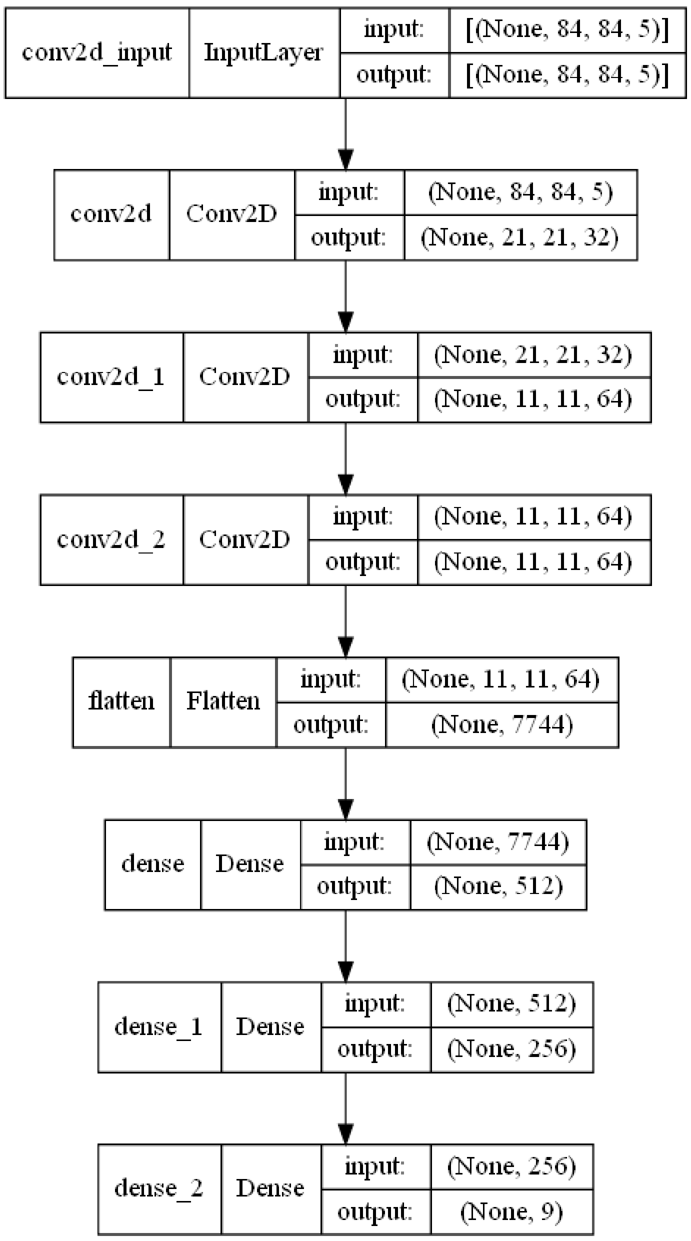
- Introduction of a better penalty policy to the agent RL in an attempt to boost the game performance.
- Various agent memory size experimentation as this had been considered but it is something that could not have been achieved in due time.
- Using a different architectures such as deeper networks or as some papers propose, the utility of the Long-Short-Term-Memory model (LSTM).
- Target DDQN model step update as this was out of the scope of this project.
- It is possible to reduce the variance of the learning process with the introduction of the standard neural network regularization technique as know as the *dropout*.
- Hyper-parameter optimization can be applied to the learning rate in order to further reduce the variance of the learning process.
- Different batch sizes can be employed in further research.
- Train all the models until convergence (not done in the report due to time constraints).

Finally, this paper would like to experiment by scaling the scope of the research by including more games such as ones where there is a different magnitude in rewards. In that particular case, the future work can also implement reward clipping.

Appendices

Appendix A

DDQN Architectures



conv2d_input | InputLayer | input: [(None, 84, 84, 5)] | output: [(None, 84, 84, 5)]
conv2d | Conv2D | input: (None, 84, 84, 5) | output: (None, 21, 21, 32)
conv2d_1 | Conv2D | input: (None, 21, 21, 32) | output: (None, 11, 11, 64)
conv2d_2 | Conv2D | input: (None, 11, 11, 64) | output: (None, 11, 11, 64)
flatten | Flatten | input: (None, 11, 11, 64) | output: (None, 7744)
dense | Dense | input: (None, 7744) | output: (None, 512)
dense_1 | Dense | input: (None, 512) | output: (None, 256)
dense_2 | Dense | input: (None, 256) | output: (None, 9)

Figure A.1: Image input DDQN architecture.

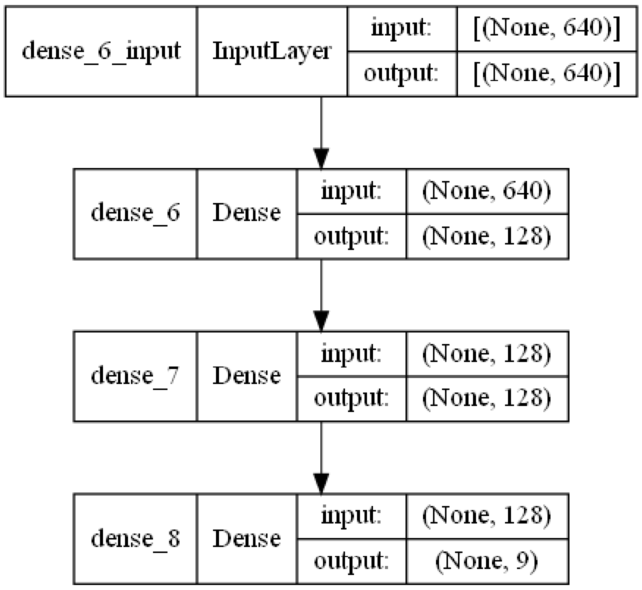


Figure A.2: Best RAM input DDQN architecture (2 fully-connected layers).

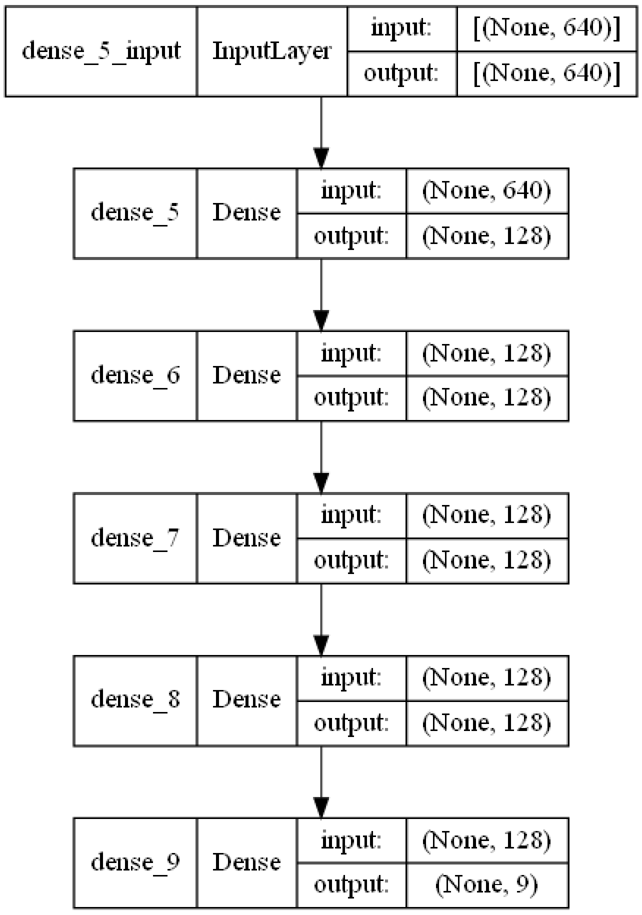


Figure A.3: RAM input DDQN architecture (4 fully-connected layers).

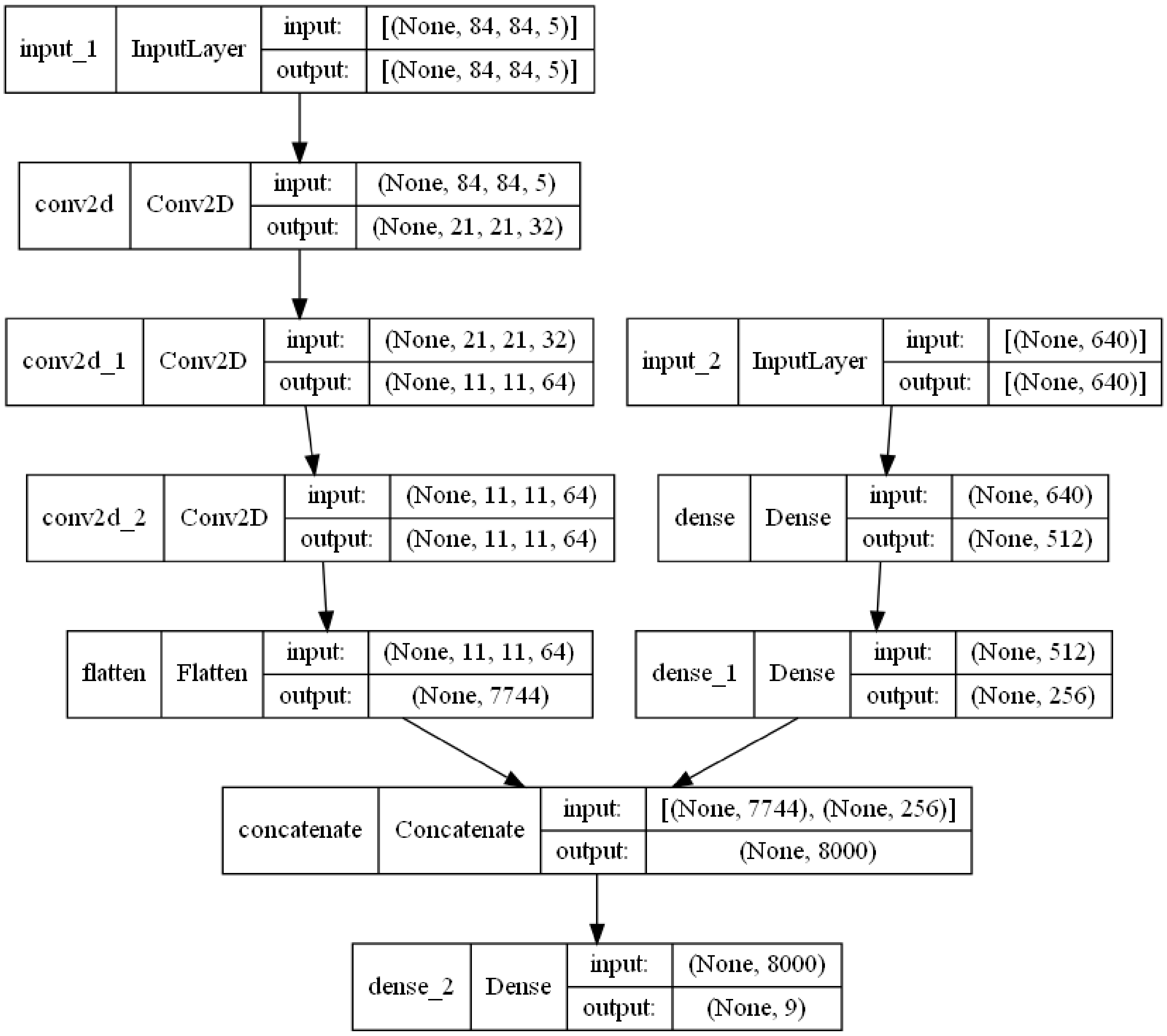

input_1 | InputLayer | input: [(None, 84, 84, 5)] | output: [(None, 84, 84, 5)]
conv2d | Conv2D | input: (None, 84, 84, 5) | output: (None, 21, 21, 32)
conv2d_1 | Conv2D | input: (None, 21, 21, 32) | output: (None, 11, 11, 64)
input_2 | InputLayer | input: [(None, 640)] | output: [(None, 640)]
conv2d_2 | Conv2D | input: (None, 11, 11, 64) | output: (None, 11, 11, 64)
dense | Dense | input: (None, 640) | output: (None, 512)
flatten | Flatten | input: (None, 11, 11, 64) | output: (None, 7744)
dense_1 | Dense | input: (None, 512) | output: (None, 256)
concatenate | Concatenate | input: [(None, 7744), (None, 256)] | output: (None, 8000)
dense_2 | Dense | input: (None, 8000) | output: (None, 9)

Figure A.4: Best mixed input DDQN architecture.

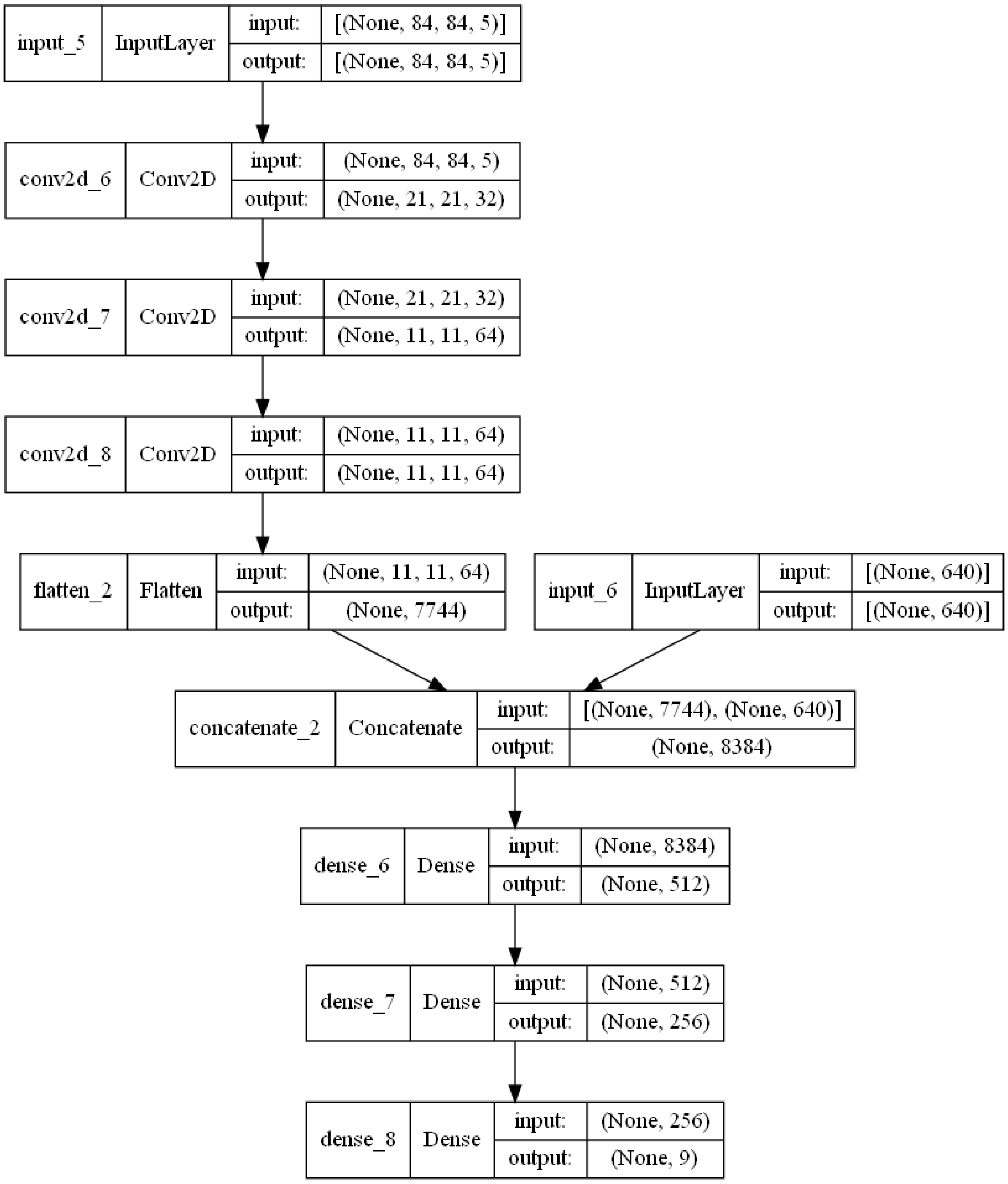

input_5 | InputLayer | input: | [(None, 84, 84, 5)]
output: | [(None, 84, 84, 5)]
conv2d_6 | Conv2D | input: | (None, 84, 84, 5)
output: | (None, 21, 21, 32)
conv2d_7 | Conv2D | input: | (None, 21, 21, 32)
output: | (None, 11, 11, 64)
conv2d_8 | Conv2D | input: | (None, 11, 11, 64)
output: | (None, 11, 11, 64)
flatten_2 | Flatten | input: | (None, 11, 11, 64)
output: | (None, 7744)
input_6 | InputLayer | input: | [(None, 640)]
output: | [(None, 640)]
concatenate_2 | Concatenate | input: | [(None, 7744), (None, 640)]
output: | (None, 8384)
dense_6 | Dense | input: | (None, 8384)
output: | (None, 512)
dense_7 | Dense | input: | (None, 512)
output: | (None, 256)
dense_8 | Dense | input: | (None, 256)
output: | (None, 9)

Figure A.5: Mixed input DDQN v2 architecture (Flatten layer output + RAM input).

Appendix B

Plots

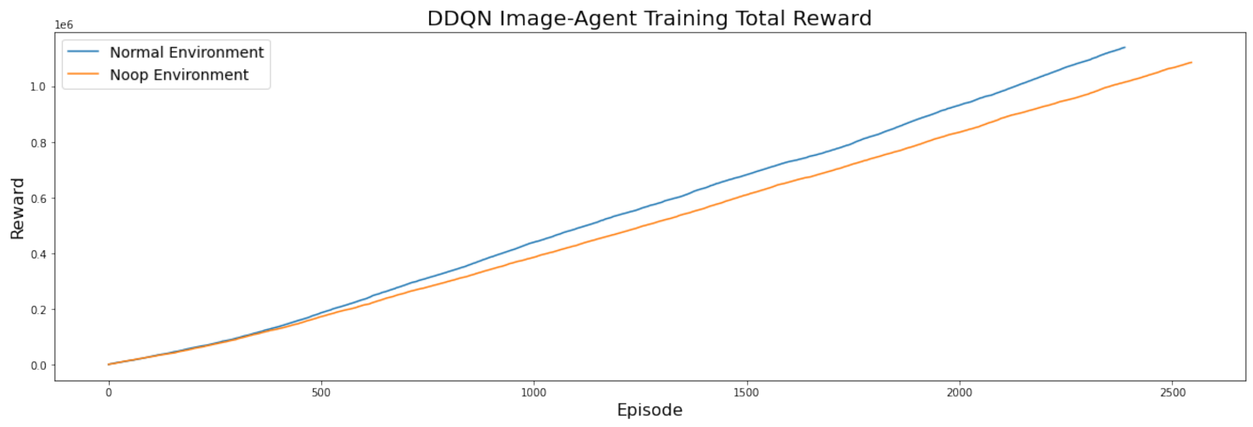


Figure B.1: Image DDQN Agents Total Reward.

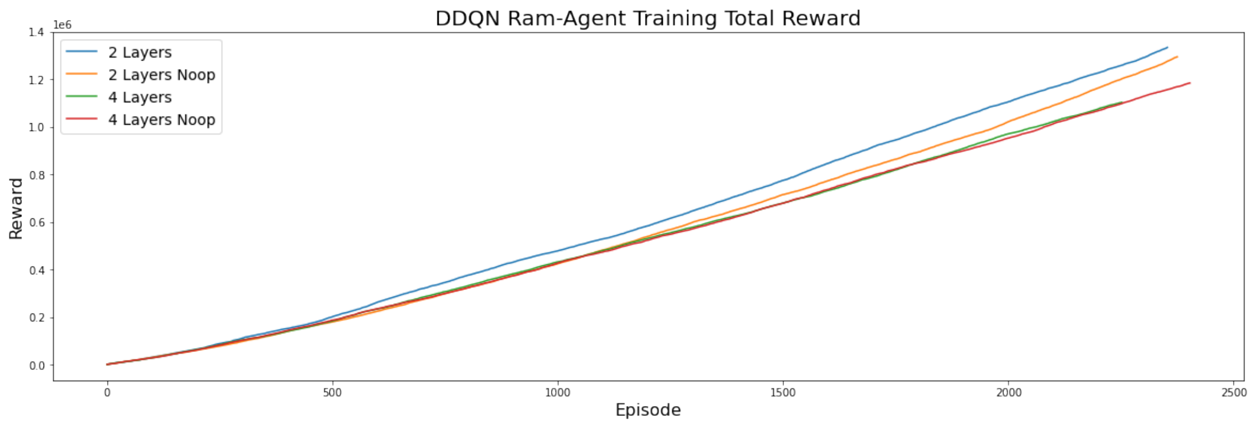


Figure B.2: RAM DDQN Agents Total Reward.

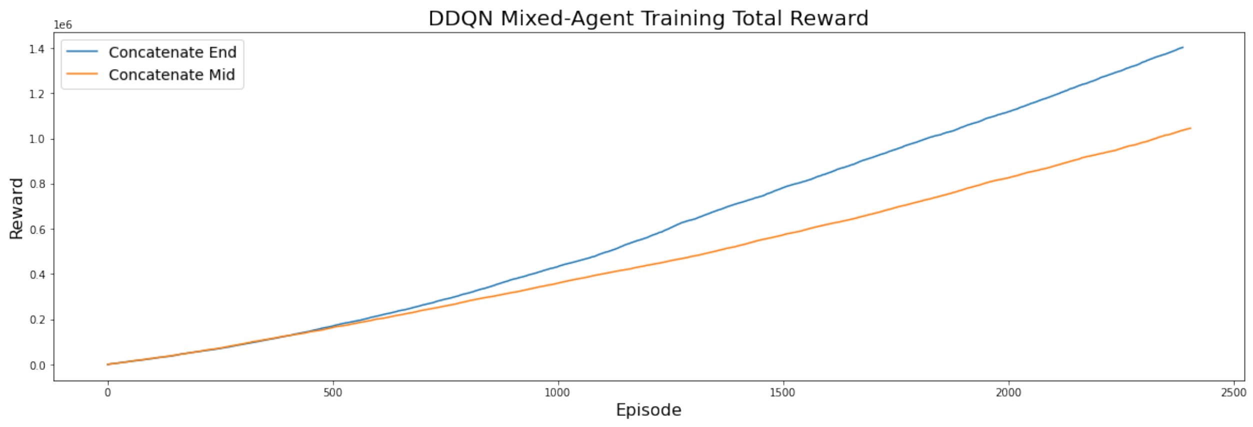


Figure B.3: Mix DDQN Agents Total Reward.

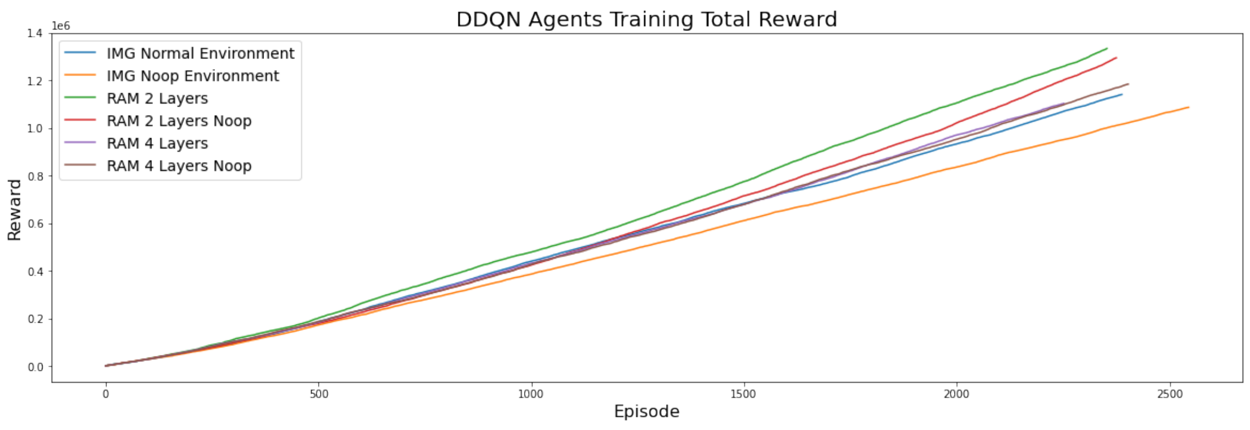


Figure B.4: Image and RAM DDQN Agents Total Reward.

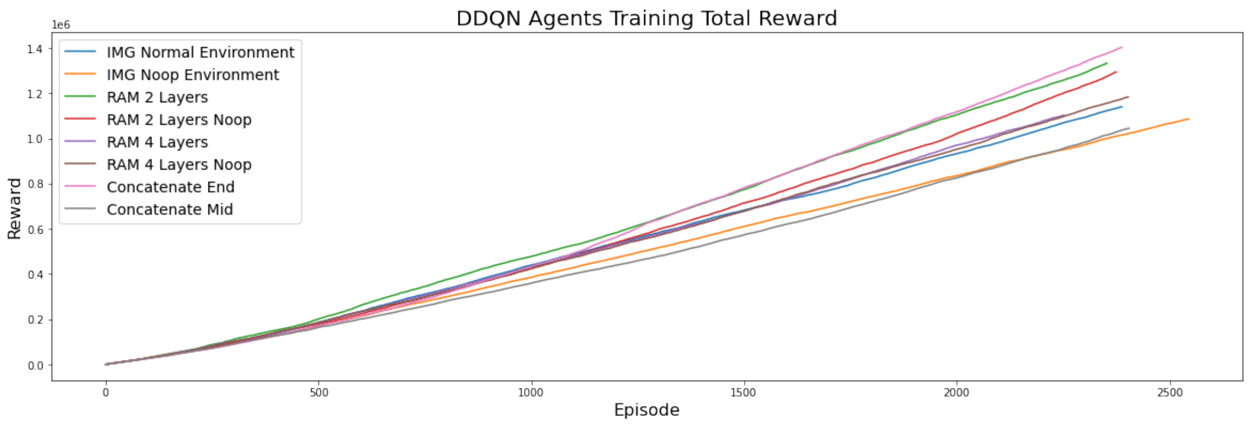


Figure B.5: Image, RAM and Mix DDQN Agents Total Reward.

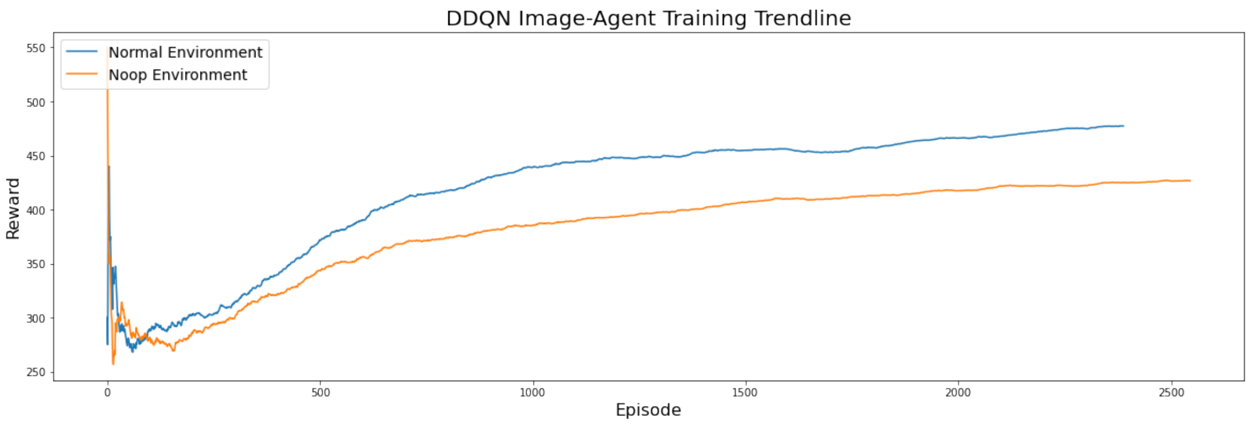


Figure B.6: Image DDQN Agents Average Reward.

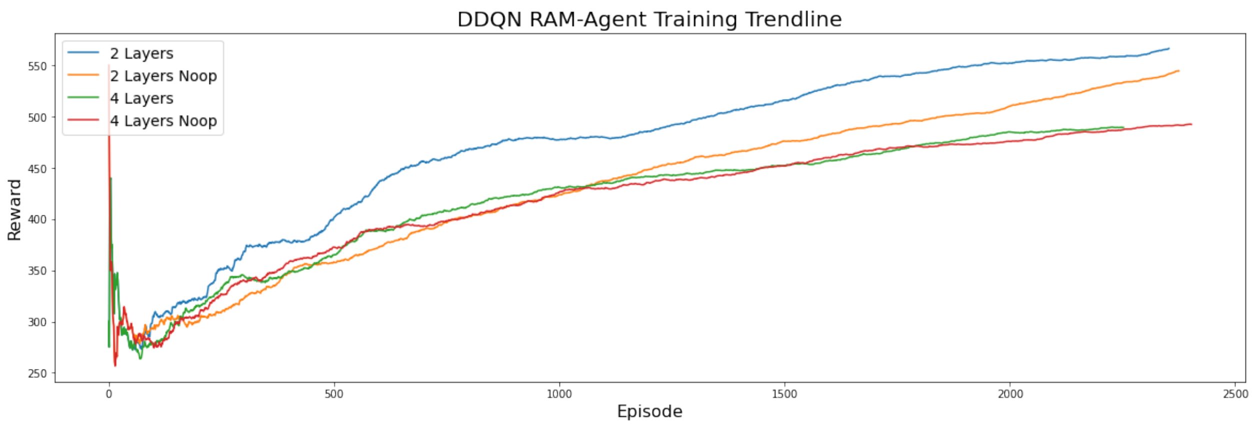


Figure B.7: RAM DDQN Agents Average Reward.

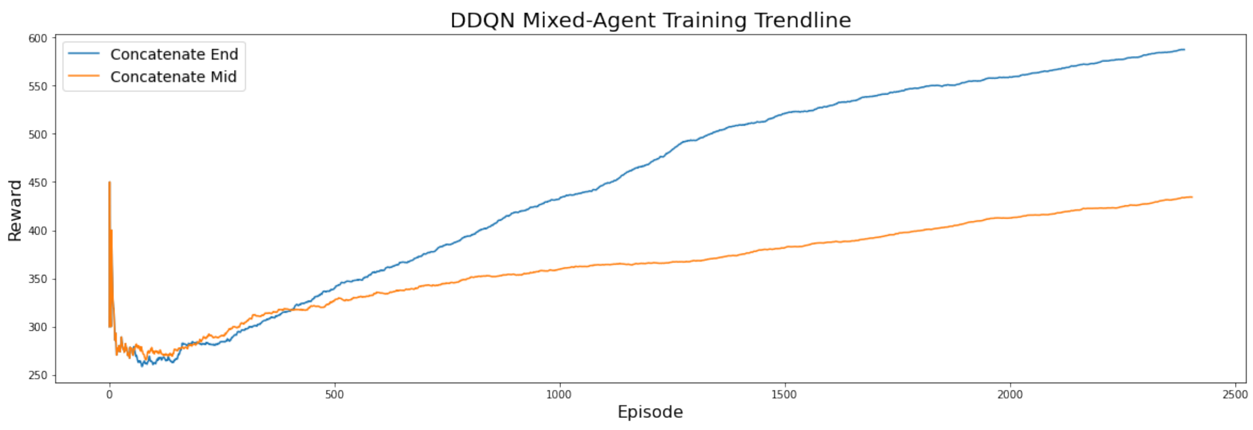


Figure B.8: Mix DDQN Agents Average Reward.

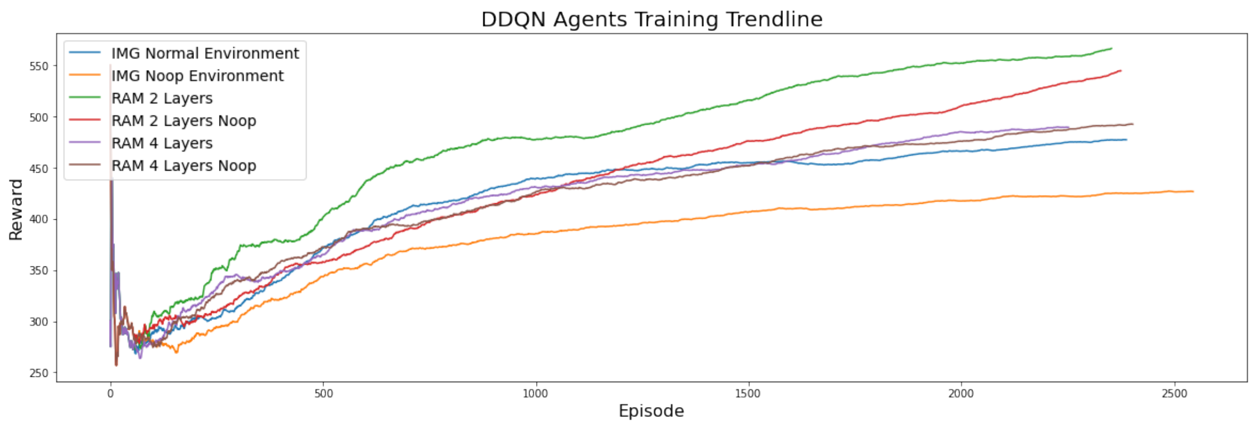


Figure B.9: Image and RAM DDQN Agents Average Reward.

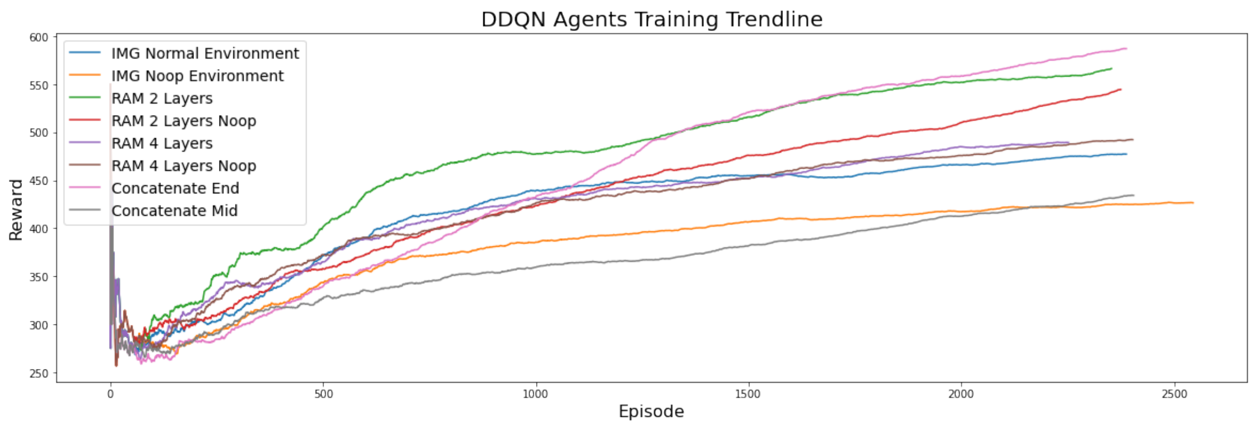


Figure B.10: Image, RAM and Mix DDQN Agents Average Reward.

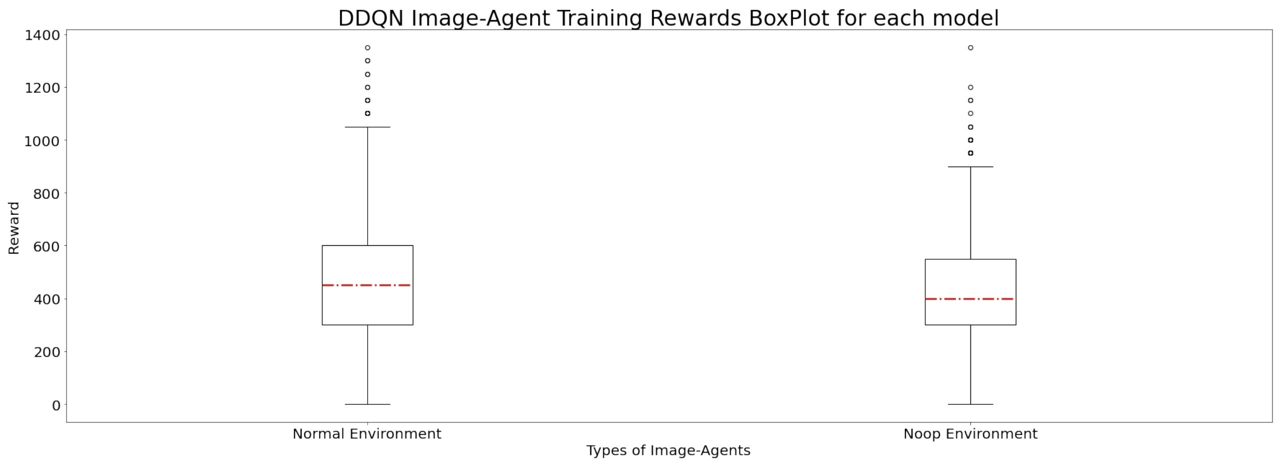


Figure B.11: Image DDQN Agents Box Plot for Reward.

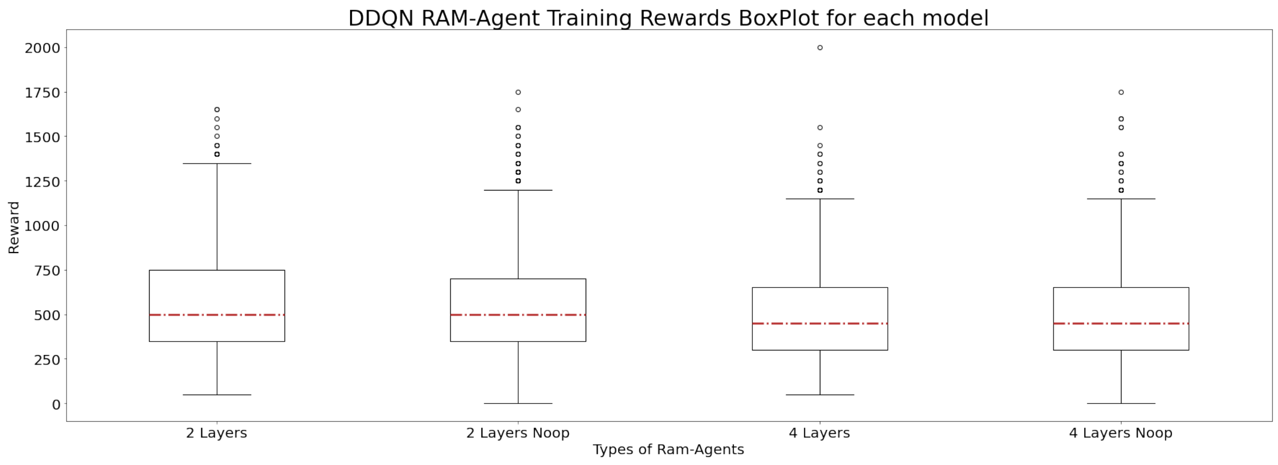


Figure B.12: RAM DDQN Agents Box Plot for Reward.

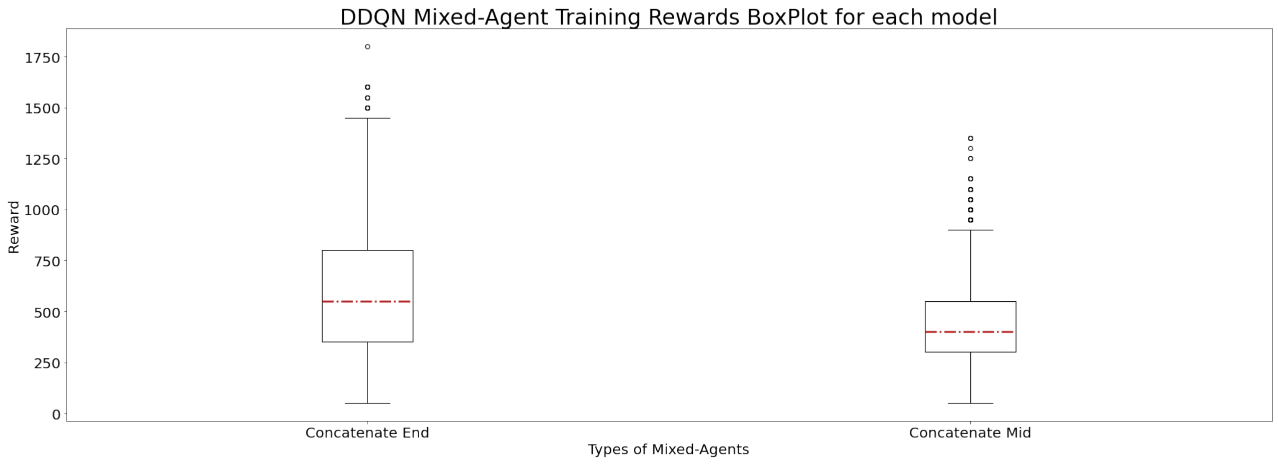


Figure B.13: Mix DDQN Agents Box Plot for Reward.

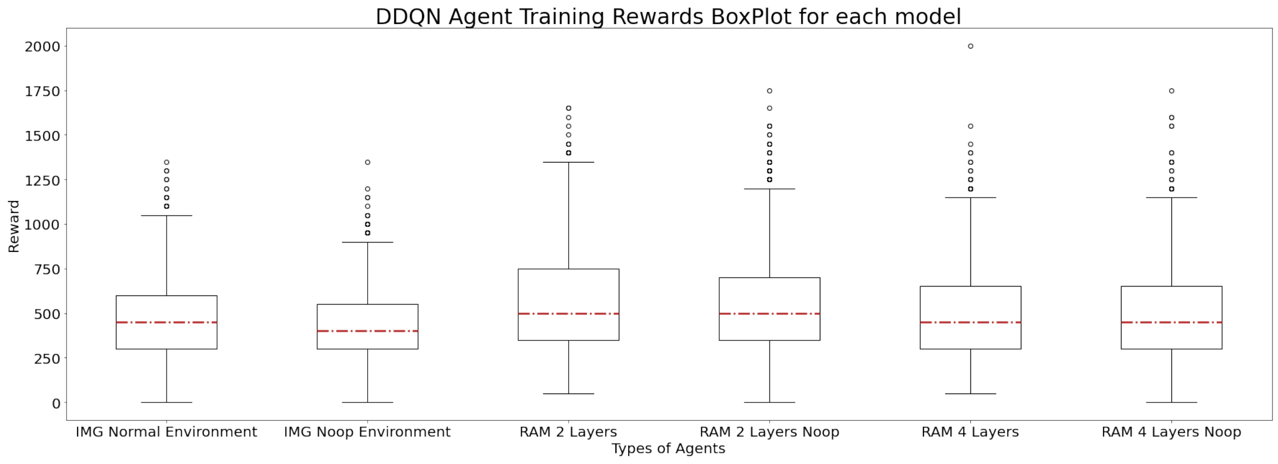


Figure B.14: Image and RAM DDQN Agents Box Plot for Reward.

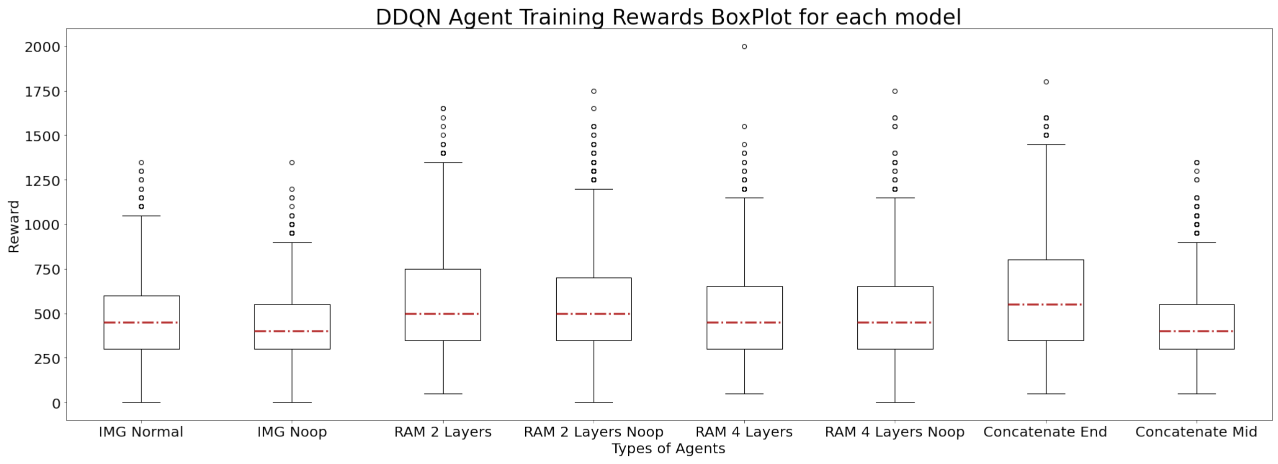


Figure B.15: Image, RAM and MIx DDQN Agents Box Plot for Reward.

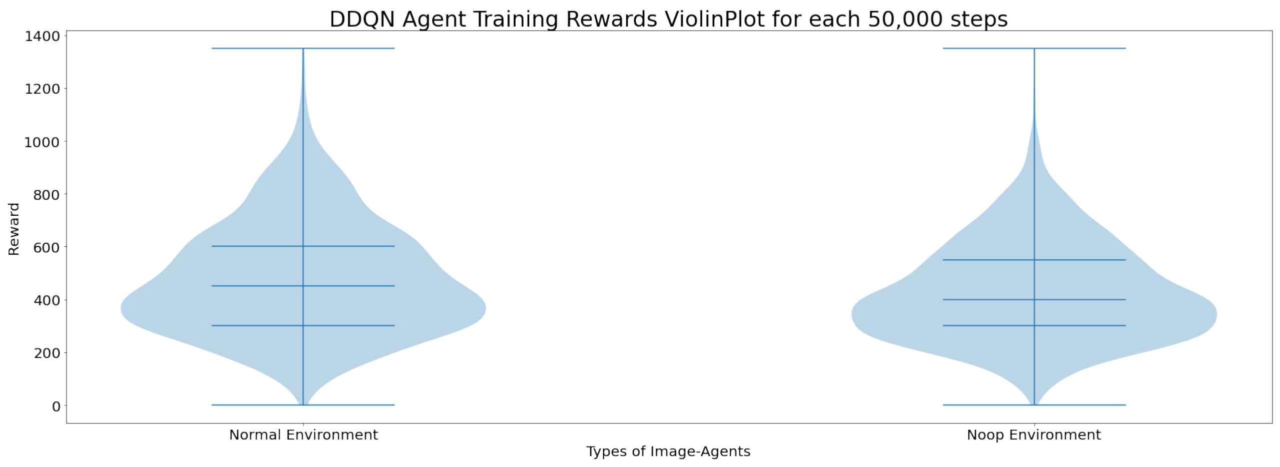


Figure B.16: Image DDQN Agents Violin Plot for Reward.

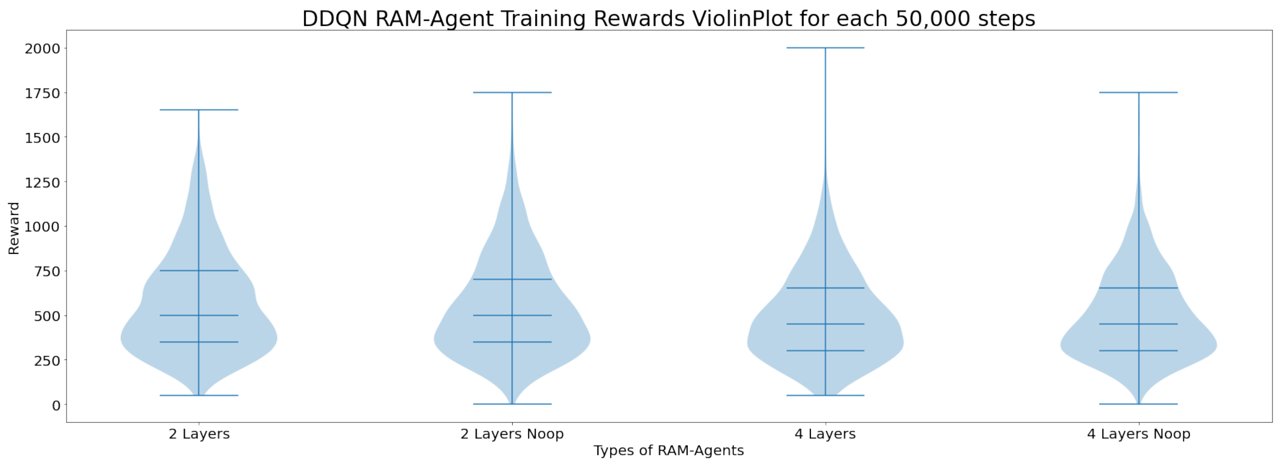


Figure B.17: RAM DDQN Agents Violin Plot for Reward.

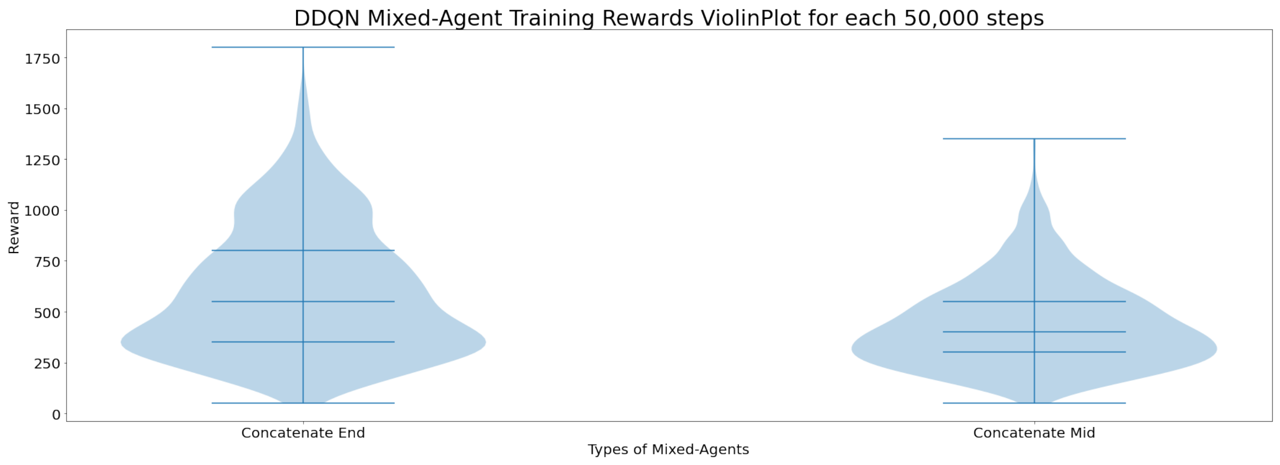


Figure B.18: Mix DDQN Agents Violin Plot for Reward.

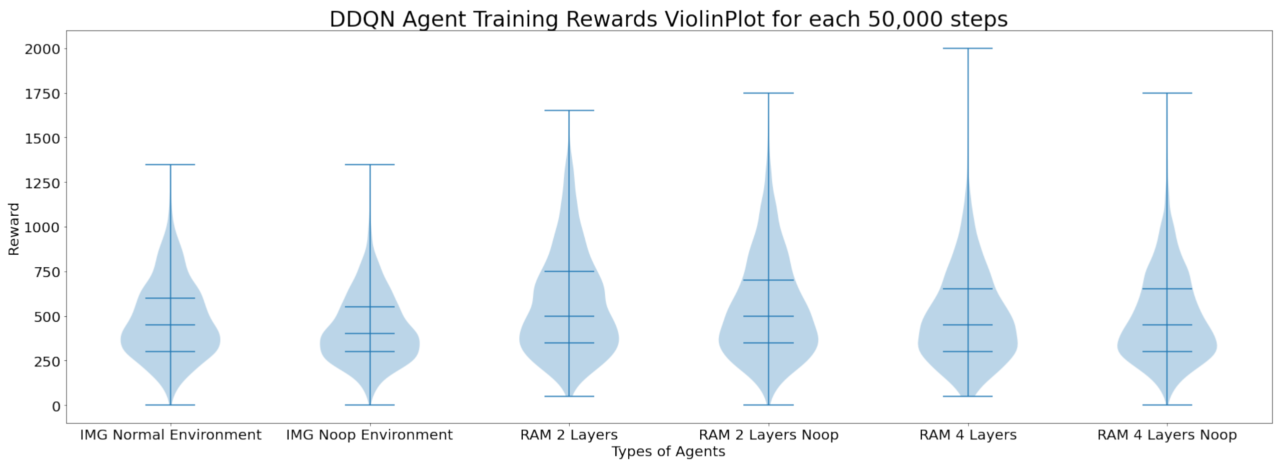


Figure B.19: Image and RAM DDQN Agents Violin Plot for Reward.

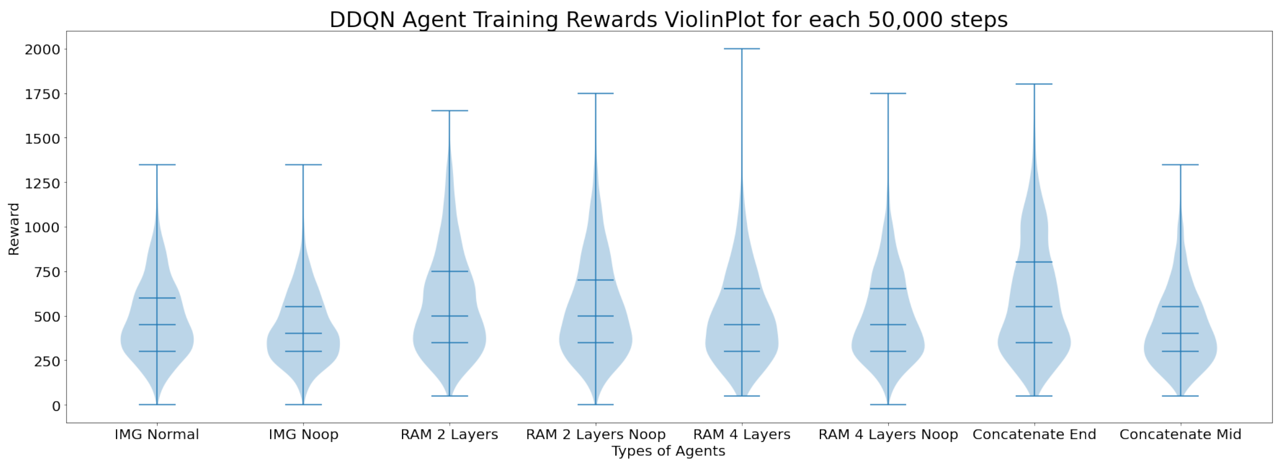


Figure B.20: Image, RAM and Mix DDQN Agents Violin Plot for Reward.

Appendix C

Notebooks

Please refer to the '*notebook_appendix*' folder that has been submitted together with the report.

Bibliography

[1] A. Walker R. Fisher, S. Perkins and E. Wolfart. Grayscale images, 2003.

[2] Yavar Naddaf. Game-independent ai agents for playing atari 2600 console games. 2010.

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

[4] Peter Henderson, Joshua Romoff, and Joelle Pineau. Where did my optimum go?: An empirical analysis of gradient descent optimization in policy gradient methods, 2018.

[5] Christian Versloot. He/xavier initializatio & activation functions: choose wisely, 2019.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[7] Hado Van Hasselt. Double q-learning. pages 2613–2621, 01 2010.

[8] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.