Building a Double Deep Q Network for Seaquest

ABSTRACT

For the first assignment of the Applied Artificial Intelligence module at the University of Aberdeen, students were tasked with building a Deep Q Network for the Atari game Seaquest and evaluating its performance on three different types of input (RAM, pixel images and both). Our team built a Double Deep Q Network using model architectures based on research, and explored different optimizations such as loss clipping, reward clipping, frame skipping, prioritized experience replay and different target network updates. Our results show that the model trained on pixel frame input had the best average performance by the end of the training period, but the mixed frame and RAM input model performed the best out of all the models when observing the highest reward values obtained in a single episode.

1. REINFORCEMENT LEARNING FROM THE SCREEN FRAMES

Prior to commencing this report, we would like to state that this project was developed in Pycharm using Python 3.7, and a 'requirements.txt' file was handed with the code listing all the libraries (and versions) we used. **We would also like to state that all the code in this project was adapted from code and material that was given and taught to us during lessons and tutorials in the unit.**

1.1 Importing and Observing the Environment

The purpose of this assignment is to train an agent to play the Atari 2600 game Seaquest as optimally as possible and compare the performance of the agent when being trained on different input types (RAM, pixel images and both). Seaquest is a singleplayer (and multiplayer) game where the player uses a submarine to shoot enemies and rescue divers. We imported our training environment from the Atari library[1] from OpenAI Gym. We decided to import the 'SeaquestNoFrameskip-v4' since the default 'Seaquest-v0' environment has inbuilt stochastic frame skipping (randomly skips between 2-4 steps every time). We will use this environment to obtain both pixel and RAM inputs. The RAM inputs will be obtained when necessary using a wrapper (env.unwrapped._get_ram()). The environment is loaded as follows:

```
env_name = "SeaquestNoFrameskip-v4"
env = gym.make(env_name)
```

The training environment can provide us with two types of observations - Pixel and RAM. The first type of observation is a 210x160 RGB image, which is an array of shape (210, 160, 3). The second type of observation consists of 128 bytes which represents the RAM state of the Atari at any given game state. Given an observation, the agent is able to perform one of a total of 18 actions, or in other words, the size of the action space is 18. Based on the source code of the environment, the rewards given from training environment when performing an action are obtained from Atari 2600 VCS emulator running the Seaquest game. The following manual[2] provides a detailed description of the scoring system of the game. For instance, when the game begins, every killer shark and enemy sub is worth 20 points. However, to prevent instability on our models (and reduce the impact of varying rewards), we introduced reward clipping, which means that positive and negative actions will be rewarded with +1 and -1 respectively. When observing the environment's info dictionary, we are presented with a variable 'ale.lives' containing a value of 4 corresponding to the total amount of lives a player has before the game ends. Hence, each episode will be reset when the number of 'ale.lives' reaches a value of 0. It should be noted that a player is awarded an additional extra life each time the player scores 10,000 points.

1.2 Q-learning vs. Deep Q-learning

Traditional Q-learning is a value-based reinforcement learning algorithm which tries to teach an agent the best action to take given a current state (or observation). During training, a traditional Q-learning algorithm will generate a q-table of shape (state, action), where each row represents each possible state and each column represents each possible action [1]. The values of the q-table are initially 0 (or very low), and then updated to store q-values (the future reward or quality of a particular state and action) after each training episode. The agent will utilise the q-table to select the best action at a particular state based on it's q-value, and it should be noted that the goal of Q-learning algorithm is to learn a policy that maximizes the total reward. The Bellman equation below represents how the q-value of a state s and action a is updated in the q-table;

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma\ maxQ(s_1,a_1) - Q(s,a)]$$

where r is the reward, α and γ is the learning rate and discount rate, and $maxQ(s_1,a_1)$ is the maximum expected reward from the next state s_1 and action a_1. A traditional Q-learning algorithm would not be suitable for the game Seaquest as the observations of the game are too complex and high dimensional to be efficiently mapped in a q-table. As a result, it would be too difficult and computationally expensive to map the entire observation space in the q-table. A more efficient and suitable approach would be to use a Deep Q-Learning Network (DQN).

In Deep Q-Learning, we utilize a neural network to approximate q-values instead of a q-table, allowing us to explore a range of different neural models (convolutional, recurrent, etc.) according to our needs. Hence, the Deep Q Network is able to receive an input (or observation) of any shape and output the corresponding q-values of all possible actions. In Deep Q-Learning we update our network by minimizing mean squared error between our target q-value and our current q-output as follows;

[1] https://gym.openai.com/envs/atari

[2] https://atariage.com/manual_html_page.php?SoftwareLabelID=424

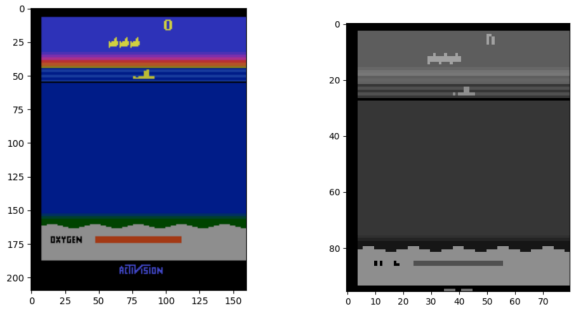
$$L(\theta) = E[(\underbrace{r + \gamma \max_{a_1} Q(s_1, a_1; \theta)}_{\text{Target}} - Q(s, a; \theta))^2]$$

where θ is a random weight that initializes the evaluation neural network. It should be noted that the value of the target q-value is computed depending on whether s is a terminal or non-terminal state. Compared to the Q-Learning algorithm used to update the q-table, the DQN will update the network based on the error between the predicted action value of the current state environment and the actual value of the next environment. In each episode of our training model, the agent will use ϵ - to determine how it should select the optimal action for a state (greedy approach is maximizing q-value, otherwise it carries out random selection). Then the agent will calculate the reward value and action to move to the next state. The parameter is updated by back propagation according to the loss function L(θ).

1.3 Pre-processing Pixels

Pre-processing the pixel observations was carried out by cropping the bottom section of each observation, sub-sampling, converting to grayscale (by calculating mean RGB values) and then normalising pixel values between the scale of 0-1. We also converted the values to type 'uint8' meaning 8 bit unsigned integer which is the smallest variable type which can be used to store the data, so that we take up less memory. Figure 1 shows an example of an observation before and after pre-processing. This section will proceed by discussing why each of these techniques were carried out, and what alternatives were they chosen over.

We cropped the bottom area of the image as it doesn't provide any useful features to the network, and as a result would take up additional memory in the network which could negatively affect training. We wanted to reduce the dimensions of our images as this would speed up training time and reduce the complexity of our networks - without negatively affecting performance. Reducing the dimensions of our observations could be carried out via sub-sampling, re-scaling or resizing. Sub-sampling involves removing every other row and column of the observation to create a half-sized image. We initially considered resizing or re-scaling each observation instead of sub-sampling so that we preserve more information in the observation, however, by doing so we found that sub-sampling the observation produced the same results, but was much faster.



(a) before (b) after

Figure 1: Input image before and after preprocessing.

Next, we chose to convert the observations to grayscale by averaging the RGB values as it was more efficient than using image pre-processing functions from libraries that use linear interpolation. We ensured that when using the averaging method, each feature was seen clearly and had a contrast from the surrounding background - particularly the divers, player and enemies. Finally, normalisation was carried out as neural networks tend to perform better using smaller data values, as these will create less bias when calculating weights. The code snippet used to crop, sub-sample and convert each observation to grayscale is shown below. Please note the observations are then normalised separately in the network model as shown in the first line of Figure 18 in Appendix B.

```
def prep_obs(obs):
    # cropping and sub-sampling
    img = obs[1:192:2, ::2]
    # convert to grayscale (0 and 255) and uint8
    img = img.mean(axis=2).astype(np.uint8)
    # reshape for tensorflow compatibility
    return img.reshape(96, 80, 1)
```

1.4 Implementing the Agent and Network

For the first task, a Double Deep Q agent based on a Convolutional Neural Network (CNN) was implemented. We initially started implementing our models using Keras, however, we ended up swapping back to tensorflow after discovering our models were running 2.5× faster using tensorflow. The difference between a Double Deep Q Network (Double DQN) and a traditional DQN is that a Double DQN uses two networks, only one of which (the online network) is updated at every training step, whilst the other network (the target network) is updated with the weights of the online network at constant step intervals. This modification has been shown to improve the stability of the learning algorithm significantly [2].

1.4.1 Building the CNN Architecture

The architecture of the CNN model is based on the work done by Mnih et al. [2], where a model with an identical architecture was trained to play a set of 49 Atari 2600 games. In the aforementioned work, it was reported that the model managed to surpass the level of a professional human games tester in 29 out of the 49 games, though it is worth noting that their model performed 75% worse than the professional human level in question in Seaquest specifically, likely due to the relatively complex nature of the game compared to some of the other games which were learned.

The structure of our CNN model is shown in Figure 2 below. Our CNN model consists of 3 convolutional layers, each with a Rectified Linear Unit (ReLU) activation function as ReLU generally performs best with CNN layers and is considered the primary choice. The first convolutional layer has 32 filters with an 8x8 kernel size and a stride of 4, whilst the second convolutional layer has 64 filters with a 4x4 kernel and stride of 2, and the third and final convolutional layer has 64 filters with a 3x3 kernel and a stride of 1. The output from this third convolutional layer is then flattened and fed into a dense layer with 512 units also having ReLU activation functions. Finally, the output from this dense layer is fed into the final output dense layer with 18 outputs with linear activation functions, one for each possible action in Seaquest's action space. The code used to create this model can be seen in Figure 18 in Appendix B.

Hyperparameter	Value	Description
Learning Rate	0.00025	Determines how much the optimizer adjusts the model's weights at every learning step.
Discount Rate	0.95	Determines how much our Q-Learning update is concerned with the future potential rewards of a state as opposed to the immediate rewards. A value of 0 means that the algorithm will only be concerned with immediate rewards.
Maximum and Minimum ϵ	1, 0.1	The ϵ value determines the likelihood of the agent to explore, i.e. take a random action, when training as opposed to exploiting, i.e. taking the perceived best action. It begins at an initial (maximum) value and decays over a number of frames (determined by the Final Exploration Frame hyperparameter) during training to a minimum value.
Final Exploration Frame	100,000	The frame at which the ϵ value stops decaying. The value will decay at a steady rate from the first frame until this frame is reached.
Replay Buffer Size	100,000	The maximum amount of most recent frames stored in the buffer which is used for Prioritized Experience Replay. A batch of these (of a size determined by the Batch Size parameter) is taken at every learning iteration.
Batch Size	32	The number of experiences taken from the replay buffer at every training iteration.
Priority Scale	0.8	The rate at which the importance of an experience affects its likelihood to be sampled. A value of 0 would mean each experience has equal chance of being sampled regardless of priority.
Maximum Episodes	500	Number of episodes the agent is trained for.
Copy Steps	10,000	The number of steps between each update of the target network.
Frame Skip Rate	4	The agent selects a new action and the model is trained every k amount of steps, where k is the frame skip rate. During the steps in between, the previous action is repeated and the model is not trained.

Table 1: Description and values for hyperparameters used in our experiments

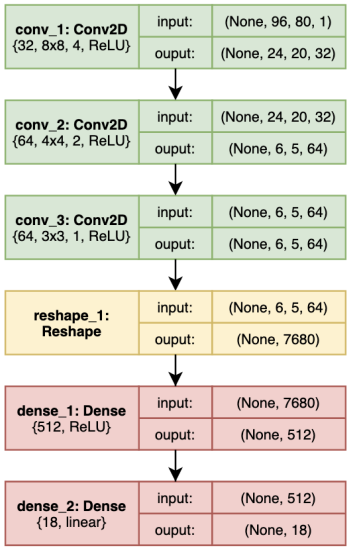


Figure 2: Network architecture of the model trained on Pixel input.

It is worth noting that unlike the research [2, 3] we referred to when building our model, we did not opt to preprocess our frames into a square shape. This is because the aforementioned research mentioned that the frames were cropped to a square shape solely due to a particular GPU implementation of 2D convolutions [4] which they used in their experiments. Since we did not work with this particular implementation, we instead opted to reshape the frames into a form which better represents the necessary area of the frame for playing the game.

1.4.2 Building the Agent

With regards to building the agent, we created a class model called 'QLearningAgent' which holds all necessary functions including an initialisation (init) function, a function to construct the previously described CNN model (create_model), a function to train the agent and update the q-values of the network (train), and a function to return an action for a given observation (get_action). We included and experimented with several optimisations to our agent to assist it in learning such as loss clipping, frame skipping, reward clipping, prioritized replay learning, soft vs. hard target network updates and varying the epsilon decay function. Please note we implemented prioritized experience replay as opposed to simple experience replay as research [5] has shown it outperformed DQN's with uniform replay on 41 out of 49 games, one of which is Seaquest. Please refer to Figure 21 and 22 in Appendix B corresponding to the init, get_action, and train functions implemented for the agent class. I will briefly describe the contents of each of these methods in the next few paragraphs, followed by how we implemented our optimisations.

In the **'init' function** we instantiate two models (the main and target network) which will be updated during training, as well as the operations needed to update the target network weights based on the main network. We implemented both hard and soft target networks, however we trained our final network using hard updates as this was supported by most research we followed [1, 2], as well as our own experimentations. In this function we also defined our error and loss functions - which is calculated by the difference between our current q-value and our target q-value (error) and then squaring it (loss). We also introduced loss clipping into our loss function using Huber loss[3] to avoid exploding gradients. Moreover, we took the account of the importance weight values from prioritized replay learning into our loss function as a research [6] discovered the design of prioritized sampling methods should not be considered in isolation from the loss function. Moreover, this would remove the bias created from prioritized replay learning as bias is introduced when the agent chooses more prioritized experiences (determined by the priority rate). Finally, we also define an optimizer to minimize our loss, and keep track of the number of steps our model is training for which would be used keep track when to update our target network and save checkpoints during training.

In the **'get_action' function**, our agent calculates the q values for a particular state (using the main network)

[3]https://www.tensorflow.org/api_docs/python/tf/compat/v1/losses/huber_loss

and then chooses whether to select a q-value using a greedy policy (selecting the optimal q-value) or by random choice based on the value of epsilon (ϵ). It should be noted that the value of epsilon is also repeatedly updated (to decay) each time this function is called based on the number of steps it already took. Moreover, the 'max()' function is used to ensure that the value of epsilon doesn't fall below a value of 0.1.

In the **'train' function**, the agent adds a new experience to the experience buffer and then extracts an experience batch from the experience buffer based on a priority scale of 0.8. Please note that the class used to manage the functionalities of the experience replay buffer is taken from the class material that was presented in one of our lectures. The class 'PrioritizedReplayBuffer' manages the experience buffer, and assigns each experience in the buffer a priority based on it's absolute loss value (and an offset to avoid priority from being 0 in some cases). The class then assigns a probability that an experience is chosen based on it's assigned priority and a priority scale (where 1.0 means experiences with highest losses have full priority and 0.0 represents simple experience replay where experiences are chosen at random). It also calculates the importance weights of the experiences to be used in the models loss function as previously mentioned to counter the bias introduced by the priority scale variable.

Additionally, it should be noted that we also implemented **frame skipping** (from scratch) and reward clipping to improve our agent's training. Frame skipping (with a frame skip rate of 4) was implemented in the training process (which will be described in Section 1.5), where the model repeats an action for 3 steps and then acquires a new action and trains the model every 4^{th} step. It should be noted that the default environment uses stochastic frame skipping (skips 2-4 frames randomly each step) so to properly implement frame skipping we used the 'SeaquestNoFrameskip-v4' version. Frame skipping allows us to speed up the training process and skip steps that wouldn't be significant to training, in fact, Sygnowski and Michalewski [1] achieved rather satisfactory results using a frame skip rates as high as 30 in Seaquest, although this was not the case in some other games. Reward clipping was also implemented in the training process, and involved clipping reward values so that negative reward values would be -1 and positive reward values would be 1. This is done in order to avoid excessively high reward values interfering with the training of the neural model since such models usually work with low weights, hence large values could cause instability in the model. The implementation for frame skipping and reward clipping is shown in Figure 23 in Appendix B, highlighted in the comments.

1.4.3 Choosing Hyperparameters

The hyperparameters used to train our model were highly influenced by the hyperparameters used by Sygnowski and Michalewski [1], which were in turn influenced by the aforementioned work of Mnih et al. [2, 3]. We opted towards taking a closer likeness to the hyperparameters chosen by Sygnowski and Michalewski [1] as opposed to those given by Mnih et al. [2] due to the fact that the latter's experiments involved training for 50 million frames, which equates to roughly 38 days of game experience. On the other hand, the experiments conducted by the former lasted between 1 and 3 days each. Due to the limited timeframe provided for this assignment, it would not be feasible to train for excessively long periods of time and hence we opted to use similar hyperparameters to those used in previous research which were trained for a relatively shorter duration. Table 1 shows a detailed description of the purpose of each hyperparameter as well as their values for our experiments.

1.5 Training and Evaluating the Agent

Our agent was trained using the aforementioned hyperparameters. The training was done on a laptop with an Nvidia GeForce RTX 2070 GPU with 8 GB of GDDR6 VRAM, as attempts to run the process on Google Colaboratory[4] resulted in the execution timing out due to the long training time.

1.5.1 Training Process

To train our final models, we instantiate an agent class then loop through a total of 500 training episodes. Figure 23 in Appendix B illustrates the code corresponding to this training process. At the beginning of each episode we reset the training environment, and at the end of each training episode we display the total number of training steps that were performed and the total reward of the episode. During each episode, we loop through an infinite number of training steps until the agent loses all of it's lives (i.e. done from the 'step()' function is set to True). At each step in the episode we keep track of the total train steps that were taken to keep track of when to regularly update our target network and save checkpoints of our models. At every step we perform the last chosen action from the 'get_action' function previously described, and use it to acquire the next state, reward and terminator flag (done). It should be noted that the next state is pre-processed using the 'prep_obs' function described in Section 1.3, and the reward is clipped using 'np.sign()' function. At the end of the episode the next state is then set to the current state. Since we implemented frame skipping (at a frame skip rate of 4), our agent will retrieve a new action and train the model using the most recent state (and values from get_action) every 4 steps.

1.5.2 Evaluation

The total reward per episode and average reward per episode during the training of our frame based agent can be seen in Figures 8 and 11 respectively. As can be seen from the first figure, the results obtained by the agent are rather inconsistent throughout the entirety of the training process. Despite this, it can be seen that the results were more inconsistent towards the beginning of the training process as opposed to later on, with much more occurrences of 0 rewards towards the beginning of training along with some occurrences of very high rewards. On the other hand, later on in the process the amount of 0 rewards appears to drop considerably, having no episodes with 0 rewards observed at all in the last 200 training episodes.

When looking at Figure 11, one can note that the average reward per episode slightly improves over the course of the training process. Although it is very inconsistent within the first 100 episodes, due to the small sample size and high exploration rates at the start of training, the average reward per episode can be seen to follow a relatively steady pattern of improvement for the rest of the training process, finally reaching an average reward per episode of 5.148 by the end of training.

[4] https://colab.research.google.com/

2. REINFORCEMENT LEARNING WITH RAM

2.1 Pre-processing RAM

The RAM data for the Atari environment consists of a list containing 128 bytes of data represented by unsigned 8 bit integers (0-255), which represents the state of the Atari machine's RAM in any given game state. This type of data is naturally much more abstract to the human eye when compared to screen input from the game itself. Nonetheless, it might be possible for an agent to learn to play the game using this input. In fact, previous research [1] has shown rather satisfactory results for learning Seaquest based on RAM input when compared to models learning from screen input [3].

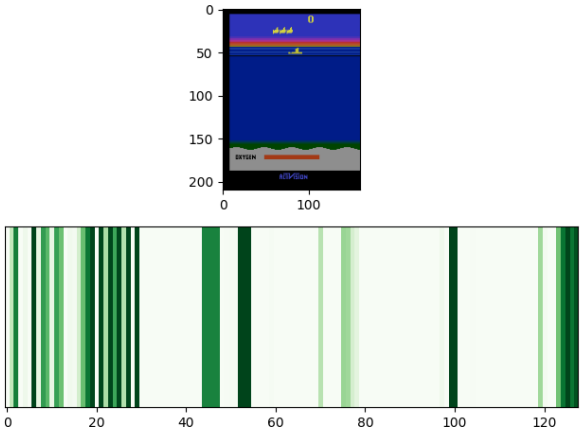


Figure 3: Visualisation of RAM state in initial state of the game.

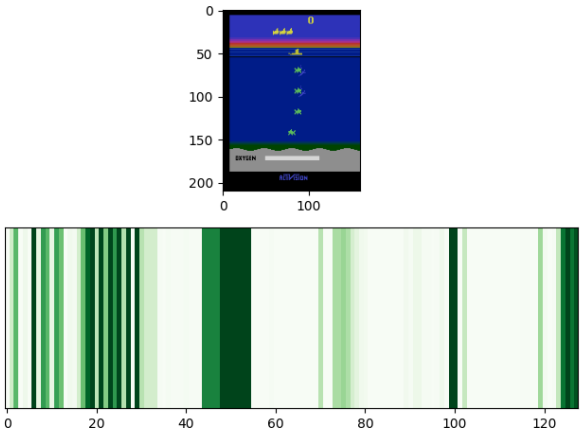


Figure 4: Visualisation of RAM state in state of the game with enemies and divers on screen.

Figures 3-5 show a visualisation of the RAM state of the game alongside its corresponding frame representation. The visualization is in the form of a heatmap where white represents a value of 0 in the respective RAM unit whilst darker green colours represent higher values. Figure 3 shows the state of the RAM in the initial state of the game, whilst Figure 4 shows the state in a scenario with enemies and divers in the game space, and the player still in the initial position. Finally, Figure 5 shows the RAM state in a scenario where there are no enemies or divers in the game space and the player is firing whilst at the bottom of the screen. As can be seen from the visualisations in these figures, the most important RAM cells for this game, i.e. the ones that change the most between the different states, appear to be those within the range of RAM units 40-60, 70-80 and 90-110.

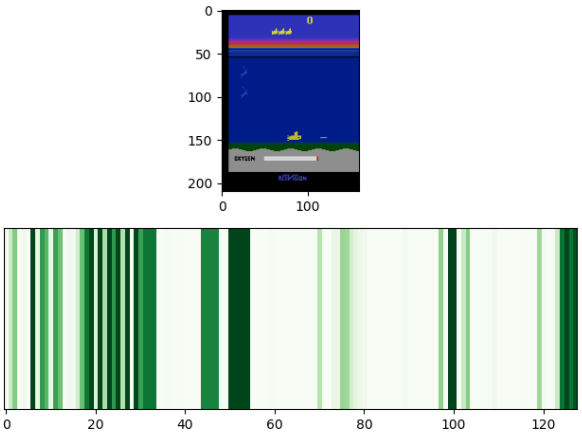


Figure 5: Visualisation of RAM state in state of the game with player at the bottom of the screen and firing.

Due to the relatively low dimensionality of the RAM state representation when compared to the pixel frame data, it was not deemed necessary to apply any dimensionality reduction techniques. Nonetheless, the data was still normalised for the same reasons mentioned previously in Section 1.3.

2.2 Implementing the Agent and Network

When developing our agent to learn the game from RAM inputs, we used the exact same agent as detailed in Section 1.4 with the same algorithms (get_action, init, train) and optimisations (loss clipping, reward clipping, frame skipping, experienced replay learning, etc.). The only difference between the agent class in this task and the previous task is the architecture of the underlying neural network in the create_model function, as well as the type of observation the agent handles (now a one dimensional input of size 128 rather than a 2D input of size (96, 80)). Since we are working with one dimensional input, a dense (fully connected) neural network was used as opposed to the convolutional model used for Task 1.

The architecture of our model consists of 4 dense layers, each with 128 units and a ReLU activation function, followed by a dense output layer with one unit for each possible action in the action space and a linear activation function, just like the output layer in our model for Task 1. This network architecture was inspired by the research conducted by Sygnowski and Michalewski [1], who experimented with this architecture and a similar one with 2 less dense layers, and obtained significantly better results using this architecture. A visualization of this architecture can be seen in Figure 6. Please note the code used to build this architecture is found in Figure 19 in Appendix B.

It is also worth noting that before commencing the training of the final models, we conducted informal experiments with less episodes to analyze the performance of different optimisers, namely Adam, RMSProp and Momentum. We found that Adam performed better in our informal experiment, and hence opted to use it for our final models even though previous research [1, 2, 3]

appeared to favour RMSProp for this particular application. We are unsure if this is due to the fact that the Adam optimizer [7] was relatively new at the time and perhaps less known, or whether the researchers had some other reason to favour RMSProp, but as already mentioned, we opted for the Adam optimizer due to the results of our informal experiments. The Adam optimizer probably outperformed RMSProp during experimentation due to the fact that it is an improvement to the RMSprop optimizer as it has slightly more momentum (since it also stores the exponentially decaying average of past gradients), hence, it is less likely to end up in a local minima.

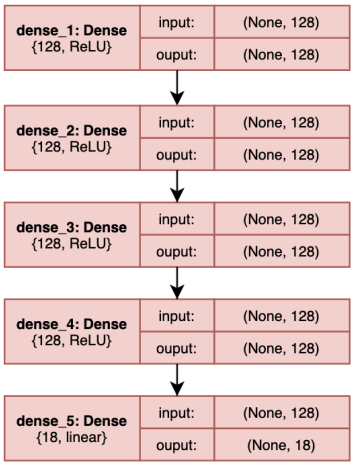


Figure 6: Network architecture of the model trained on RAM input.

2.3 Training and Evaluating the Agent

The agent was trained under the same experimental setup as the agent in Task 1, using the same hyperparameters as shown in Table 1. The total reward per episode and average reward per episode in training can be seen in Figures 9 and 12 in Appendix A respectively.

By observing Figure 9, one can immediately notice that all the way to the last 40 episodes of the training, episodes with 0 reward can still be found. The highest individual rewards per episodes appear to be obtained in the 80-140 episode range, with the peaks obtained in this range never being reached again throughout the rest of the training process. In fact one can note from Figure 12 that the peak average reward seems to be achieved around the 140 episode mark, when ignoring the initial spike which can be attributed to coincidence caused by the low sample size and high exploration rate at this early stage of training. After this point, the average reward fluctuates slightly throughout the rest of the training process but then suffers a net loss throughout. Training finishes at a mean reward per episode of 4.960 after a total of 500 episodes.

Since there appears to be a trend of decrease in average reward per episode after the initial part of the training process, one can reasonably be inclined to believe that some tuning to the experimental setup should be carried out, either in the form of changes to the preprocessing, hyperparameters or network architecture (maybe trying other model architectures such as a LSTM model). Whilst this is not entirely unreasonable to believe, it is also key to remember that agents previously trained for this game were trained for very long periods of time and still did not achieve results surpassing human capabilities [2]. Since the decrease in average reward is not of an overwhelming magnitude, the results cannot be considered conclusive evidence of a flaw in the experimental setup itself. However, perhaps adopting a more complex model architecture could improve the training of the network. Another possibility of refining this model would be to implement frame stacking (where a total number of n frames or observations) is fed to the agent at a time during training instead of just one observation, as carried out in research we observed [2, 3]. This will allow the agent to pick up certain features which it might not be able to learn from a single frame such as the direction of movement between items and the location of certain objects which might not appear on all frames.

2.4 Comparing performance - Pixel vs. RAM

When comparing the results between the training of the pixel frame based model and the RAM based model, a number of disparities can be noted. Figures 14 and 16 in Appendix A can be used to help visualise the differences between the results of these two models.

Firstly, one can notice from Figure 14 that the RAM model appears to suffer much more from numerous episodes having very low (or 0) reward values, and appears to be more inconsistent overall throughout. Moreover, it can be seen from Figure 16 that the average reward per episode in the initial stage of the training process fluctuates much more violently for the RAM model when compared to the pixel model.

After this initial stage of training, when both models' average reward per episode appears to begin stabilizing at around 50 episodes (due to less exploration), the RAM model appears to perform better for the next 250 episodes or so. Despite this, the overall performance of the RAM model appears to begin falling off at around 140 episodes as mentioned previously whilst the frame based model is still at a relatively steady pattern of improvement at this point.

The frame based model appears to finally surpass the average reward per episode of the RAM based model at around 300 episodes into the training process, at which point it stays on top for the rest of the process and finishes the 500 episodes with a mean reward per episode of 5.148 compared to the RAM model's 4.960.

Although previous research [1] stated that their RAM models performed better than certain frame based models [3], these models were created under different experimental conditions, whilst our models trained under the exact same conditions (parameters, training process and optimisations) indicated a different result. This is not a highly unexpected result as one would expect the pixel game frames to contain more features than the RAM due to their higher dimensionality, and hence gives the agent more features to learn from.

3. REINFORCEMENT LEARNING BY MIXING SCREEN AND RAM

3.1 Implementing the Agent and Network

As was the case with the previous task, we used the same agent class as described in detail in Task 1, with the same method calls and optimisations to maintain a consistent basis for experimentation when evaluating the 3 models to each other. The only difference between the agent class of Task 1 and the agent class created for this task is the underlying neural network architecture (in this case we have a mixed neural network), and the

number and type of observations passed into the network (in this case we have two types of observations, one of size 128 (RAM input) and one of size (96, 80) (pixel input)). It should also be noted that the pre-processing carried out for each type of observation for this agent is identical to pre-processing we carried out for them in previous two tasks (described in Section 1.3 for pixel input and 2.1 for RAM input). When creating the neural network, we wanted to maintain as much as we could from the original architectures of the models we used for the previous 2 tasks. The final architecture for this model can be seen in Figure 7. Moreover, the code used to construct this model in tensorflow is shown in 20 in Appendix B.

It is worth noting that we kept all the models for the convolutional part of the model but removed half the layers from the dense part of the model. This was done as after conducting some informal experimentation, as we felt that training the model with both of the original architectures concatenated in their entirety resulted in a model which was too computationally expensive to train. Hence, we opted to keep the pixel model in its entirety and reduce the amount of layers in the dense model for the following reasons:

1. The pixel based model performed better than the RAM based model in previous experiments when comparing them individually, hence, leading us to believe there is more meaningful information to be learned from the pixel data.

2. There is a significant difference between the properties of the different layers of the pixel based model, whilst all the layers of the RAM based model except for the output layer are the identical.

3. Previous research [1], which dealt with similar mixed models also reduced layers from their individual models. Despite this, it is worth noting that our final model architecture for this task differs from both of the architectures displayed in the aforementioned research.

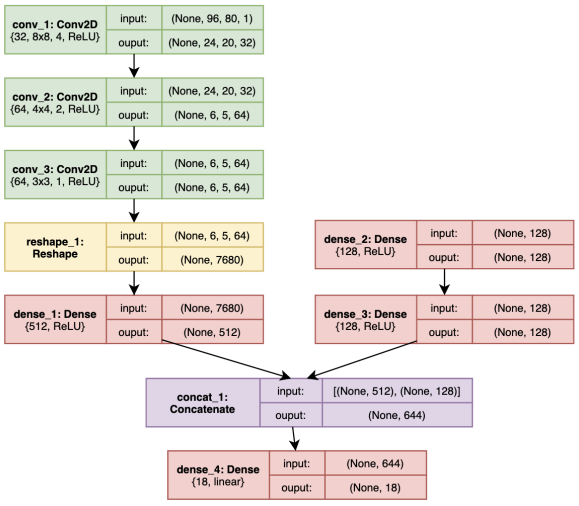


Figure 7: Network architecture of the model trained on both RAM and pixel input.

In terms of the optimizer used when training our model, we conducted some small informal experiments similar to what was done for the previous two tasks and concluded that the same Adam optimizer used for the previous networks would then be used for this one based on the results of these experiments, as well as those of the previous experiments.

3.2 Training and Evaluating the Agent

We used the same training process as the previous two tasks, described in Section 1.5.1 to train and evaluate our agent. Similarly, the same experimental setup detailed by the hyperparameters in Table 1 was used for training our final model, which allows us to compare all the models on at an equal scale. The training results of our agent is shown in Figures 10 and 13 in Appendix A.

From Figure 10 one can immediately note that the largest variance in total reward obtained between episodes throughout the training process is seen in the first 120 episodes. Further along the training process, it can be seen that the model begins obtaining more consistent results. Unfortunately, this change is not necessarily for the better as many low reward values are still obtained, and there are much less instances of higher reward values later on in the training process.

The aforementioned observations can be further backed by the average reward per episode plot shown in Figure 13. From this plot we can see that after the initial exploration stage, i.e. from around episode 80 onward, there seems to be a trend of improvement in performance for the next 60 episodes or so. Unfortunately, after this point the graph shows quite a consistent decrease in performance for the rest of the training process, finally finishing at a mean reward per episode of 4.810 after a total of 500 training episodes.

3.3 Comparing performance - All Models

Looking at Figures 15 and 17 in Appendix A, one can note from the first figure that when looking at the highest rewards obtained by each algorithm, the mixed model obtained higher results in its best cases (during early training). Despite this, the average result per episode shows the mixed model having a worse performance than both of the individual models by the end of the training process. This aligns with the conclusions made by Sygnowski and Michalewski, who said that "Mixing screen and RAM did not lead to an improved performance comparing to screen-only and RAM-only agents" [1], even though we used a different model architecture than they used in both of their mixed input experiments.

From Figure 17 one can note that all of the models appear to begin a relatively stable pattern of change in average performance at around 140 episodes into the training progress. Whilst the pixel input model begins a trend of improvement from this point until the end of the training process, the other two models begin following a downward trend from this point onward, with the mixed model declining in performance at a significantly higher rate. In fact, the mixed model begins with the highest average reward per episode between all of the models at episode 140 and ends up with the worse average reward per episode by episode 500.

The average reward per episode of the 3 models appears to converge at around 300 episodes into the training process, after which the pixel only model surpasses the other two models. Despite this, one can note that the difference between the three models in terms of their average reward per episode is not too high, with a difference of less than 0.4 average reward per episode between the best (pixel only) and worst (mixed pixel and RAM)

models at the end of the training process. Moreover, whist performing the worst in terms of average overall performance, the mixed pixel and RAM model obtained a higher maximum reward when comparing the results of the best episodes of each model, which was also a metric used by Sygnowski and Michalewski [1] when comparing their models.

Overall from the results observed it is hard to make a truly conclusive statement on the performance of the different models since the experiments were constrained to a very limited timeframe. This made it difficult to experiment with hyperparameters and model architectures as much as we would have liked whilst still training for a significant amount of episodes. Our experimental results would lead us to conclude that the model using frame input performs the best on average whilst the mixed model performs better in the best cases, but we would need to conduct further experiments going forward to draw truly conclusive results.

4. CONCLUSION

In this report we highlighted the process behind implementing and training 3 different types of Double DQN models for learning the Atari 2600 game Seaquest, which was based on a mix of research and experimentation. Through our experiments we observed that the frame based model had the best overall performance whilst the mixed frame and RAM model had the best performance when looking at the best case episodes in the training process. Due to time constraints, the training process was relatively short when compared to previous research efforts and the amount of experiments conducted to tune our models was also less than we would have liked, but some interesting results were still produced from the experiments conducted.

4.1 Future Improvements

Going forward from the experiments highlighted in this report, a number of further optimizations could be made to potentially achieve better or at the very least new results.

One example of such optimizations is the application of frame stacking, i.e. combining skipped frames into one input and using this to train the network, as describe in Section 2.3. Such an optimization could be applied to all of our models to potentially allow for more features to be learned such as the direction in which things are moving, at the cost of a higher computational overhead in training. This has shown to give good results in previous research [2, 3] and should definitely be applied in any future experiments where the higher computational overhead can be afforded.

Another such optimization which could be implemented in future work is the population of the memory buffer before training commences using a random agent for a specified number of steps, as was done in previous research [2, 1]. This could potentially help speed up the initial stages of the learning process.

Dynamic frame skipping [8] approaches were also shown to achieve much better results when learning Seaquest as well as other Atari 2600 games. These approaches involve training a network with a larger action space so that it also learns how many frames to repeat an action for as opposed to setting this to a constant value (4 in our case).

Finally, in an attempt to improve the results obtained by the mixed frame and RAM agent, an extra dense layer could be added to the model architecture of the agent between the concatenation of the two individual models and the output layer to learn patterns in the concatenation of these two models before feeding into the output layer. This could potentially result in better results being obtained by this model since it showed promise in its best case scenarios but failed to achieve consistently good results when compared to the individual models.

5. REFERENCES

[1] J. Sygnowski and H. Michalewski, "Learning from the memory of atari 2600," in *Computer Games*, pp. 71–85, Springer, 2016.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[5] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[6] S. Fujimoto, D. Meger, and D. Precup, "An equivalence between loss functions and non-uniform sampling in experience replay," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[7] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[8] A. S. Lakshminarayanan, S. Sharma, and B. Ravindran, "Dynamic frame skip deep q network," *arXiv preprint arXiv:1605.05365*, 2016.

APPENDIX

A. FIGURES

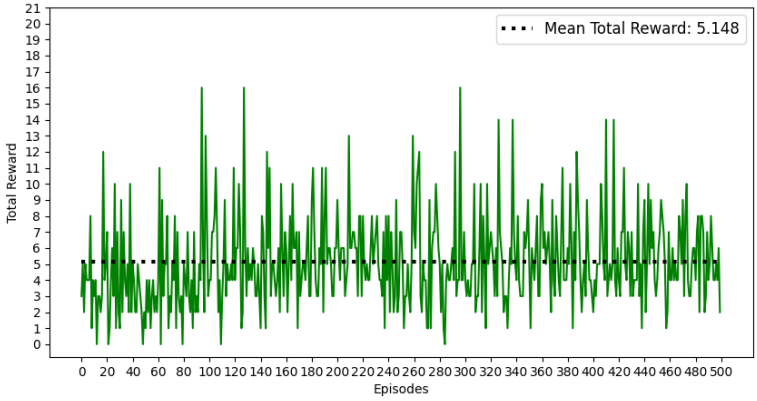


Figure 8: Total Reward per Episode obtained by the Pixel model during training.

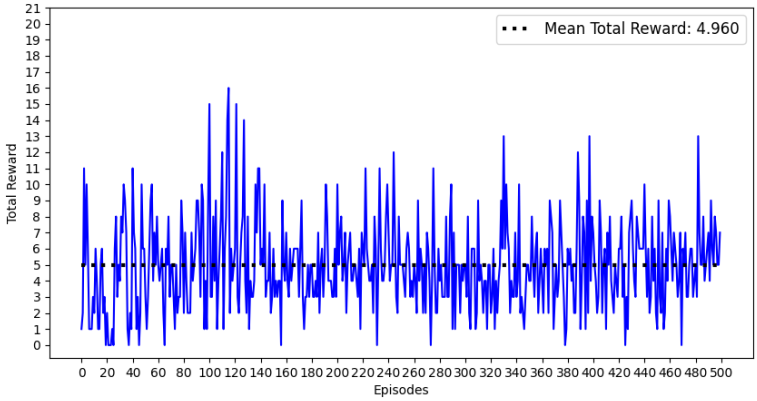


Figure 9: Total Reward per Episode obtained by the RAM model during training.

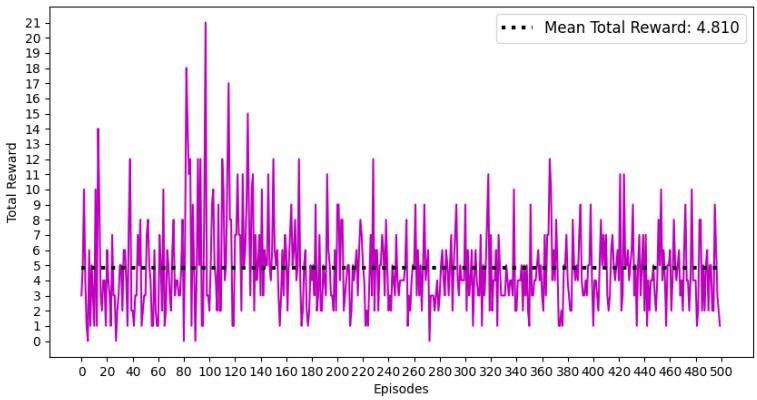


Figure 10: Total Reward per Episode obtained by the Pixel and RAM model during training.

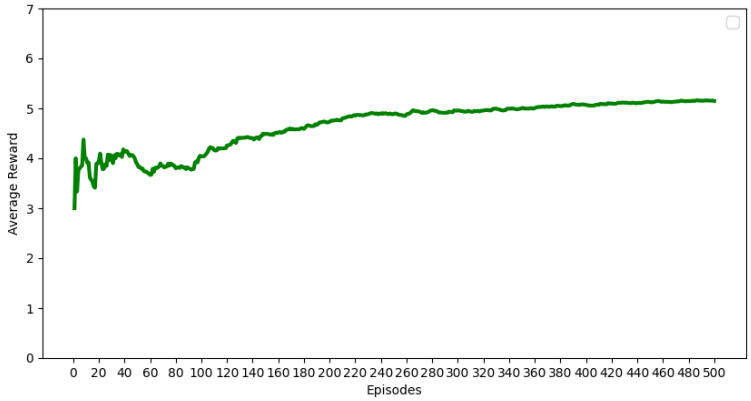


Figure 11: Average Reward per Episode obtained by the Pixel model during training.

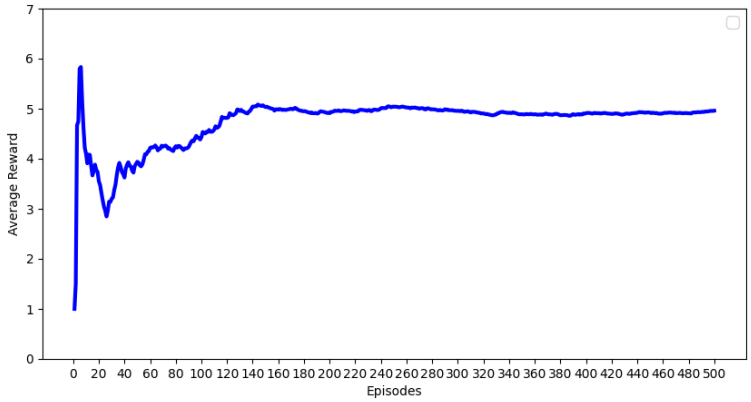


Figure 12: Average Reward per Episode obtained by the RAM model during training.

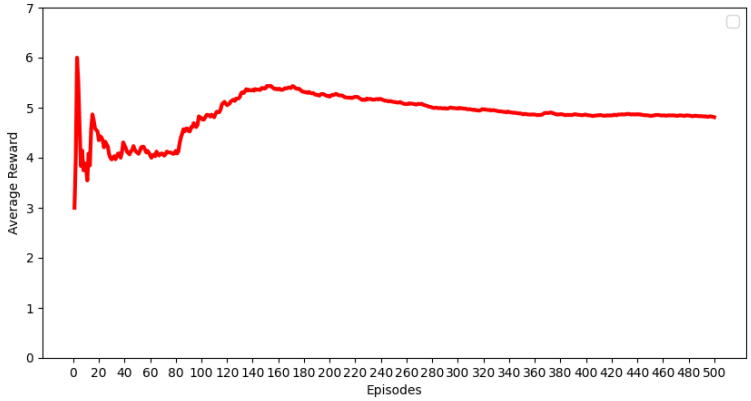


Figure 13: Average Reward per Episode obtained by the Pixel and RAM model during training.

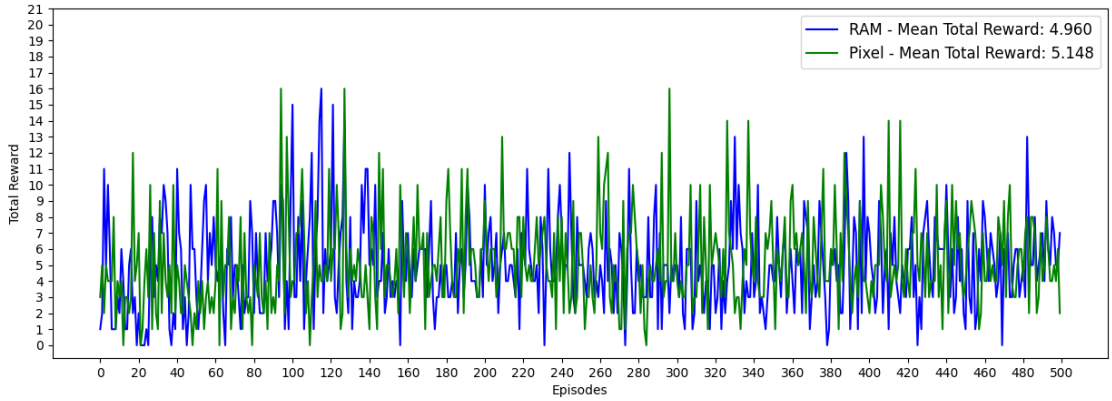


Figure 14: Total Reward per Episode obtained by the pixel and RAM models during training.

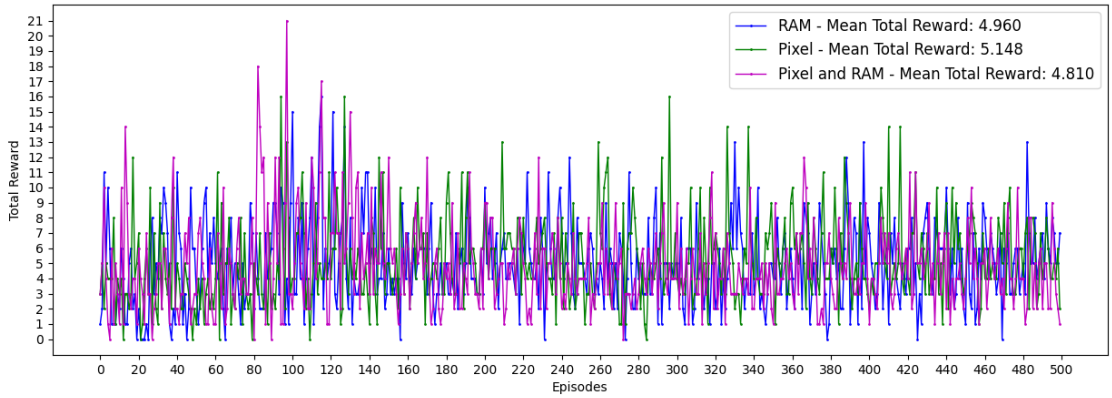


Figure 15: Total Reward per Episode obtained by all the models during training.

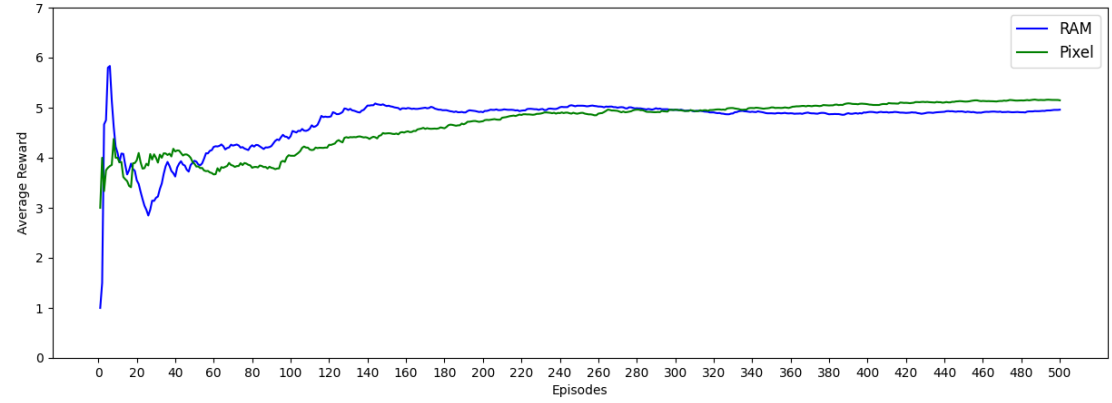


Figure 16: Average Reward per Episode obtained by the pixel and RAM models during training.

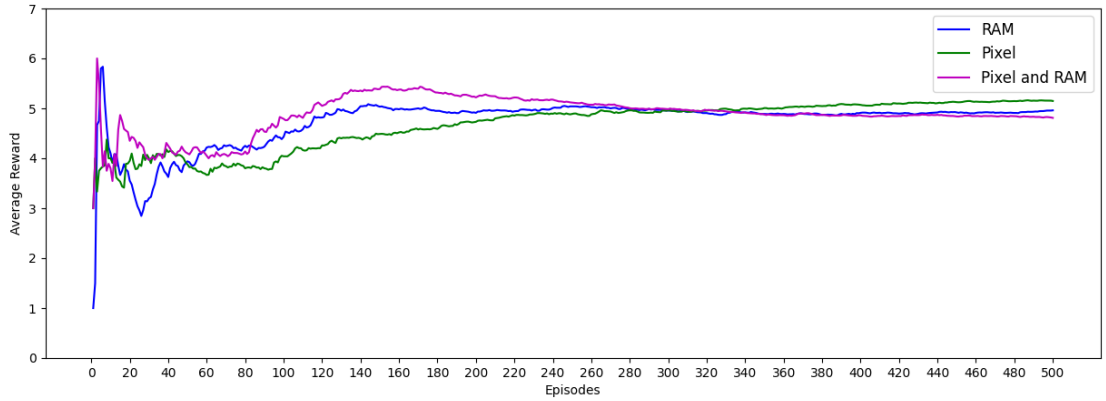


Figure 17: Average Reward per Episode obtained by all the models during training.

B. CODE SNIPPETS

```
"""
CREATING THE CNN NETWORK
"""
def create_model(self, X_state, name):
    prev_layer = X_state / 255.0  # scale pixel intensities to the [0, 1.0] range.
    initializer = tf.variance_scaling_initializer()

    with tf.variable_scope(name) as scope:
        prev_layer = tf.layers.conv2d(prev_layer, filters=32, kernel_size=8, strides=4, padding="SAME",
                                      activation=tf.nn.relu, kernel_initializer=initializer)
        prev_layer = tf.layers.conv2d(prev_layer, filters=64, kernel_size=4, strides=2, padding="SAME",
                                      activation=tf.nn.relu, kernel_initializer=initializer)
        prev_layer = tf.layers.conv2d(prev_layer, filters=64, kernel_size=3, strides=1, padding="SAME",
                                      activation=tf.nn.relu, kernel_initializer=initializer)
        flatten = tf.reshape(prev_layer, shape=[-1, 64 * 12 * 10])
        hidden = tf.layers.dense(flatten, 512, activation=tf.nn.relu, kernel_initializer=initializer)
        output = tf.layers.dense(hidden, self.action_size, kernel_initializer=initializer)

    # create a dictionary of trainable vars by their name
    trainable_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope=scope.name)
    trainable_vars_by_name = {var.name[len(scope.name):]: var for var in trainable_vars}
    return output, trainable_vars_by_name
```

Figure 18: Creating the model trained on Pixel input.

```
"""
CREATING THE DENSE NETWORK
"""
def create_model(self, X_state, name):
    prev_layer = X_state / 255.0  # scale pixel intensities to the [0, 1.0] range.
    initializer = tf.variance_scaling_initializer()

    with tf.variable_scope(name) as scope:
        # Same as big_ram model from https://arxiv.org/pdf/1605.01335.pdf
        prev_layer = tf.layers.dense(prev_layer, 128, activation=tf.nn.relu, kernel_initializer=initializer)
        prev_layer = tf.layers.dense(prev_layer, 128, activation=tf.nn.relu, kernel_initializer=initializer)
        prev_layer = tf.layers.dense(prev_layer, 128, activation=tf.nn.relu, kernel_initializer=initializer)
        prev_layer = tf.layers.dense(prev_layer, 128, activation=tf.nn.relu, kernel_initializer=initializer)
        output = tf.layers.dense(prev_layer, self.action_size, kernel_initializer=initializer)

    # create a dictionary of trainable vars by their name
    trainable_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope=scope.name)
    trainable_vars_by_name = {var.name[len(scope.name):]: var for var in trainable_vars}
    return output, trainable_vars_by_name
```

Figure 19: Creating the model trained on RAM input.

```
"""
CREATING THE MIXED NETWORK
"""
def create_model(self, X_state_pixel, X_state_ram, name):
    prev_layer = X_state_pixel / 255.0  # scale pixel intensities to the [0, 1.0] range.
    ram_layer = X_state_ram / 255.0  # scale pixel intensities to the [0, 1.0] range.
    initializer = tf.variance_scaling_initializer()

    with tf.variable_scope(name) as scope:
        prev_layer = tf.layers.conv2d(prev_layer, filters=32, kernel_size=8, strides=4, padding="SAME",
                                      activation=tf.nn.relu, kernel_initializer=initializer)
        prev_layer = tf.layers.conv2d(prev_layer, filters=64, kernel_size=4, strides=2, padding="SAME",
                                      activation=tf.nn.relu, kernel_initializer=initializer)
        prev_layer = tf.layers.conv2d(prev_layer, filters=64, kernel_size=3, strides=1, padding="SAME",
                                      activation=tf.nn.relu, kernel_initializer=initializer)
        flatten = tf.reshape(prev_layer, shape=[-1, 64 * 12 * 10])
        final_cnn = tf.layers.dense(flatten, 512, activation=tf.nn.relu, kernel_initializer=initializer)
        ram_layer = tf.layers.dense(ram_layer, 128, activation=tf.nn.relu, kernel_initializer=initializer)
        ram_layer = tf.layers.dense(ram_layer, 128, activation=tf.nn.relu, kernel_initializer=initializer)
        concat = tf.concat([final_cnn, ram_layer], 1)

        output = tf.layers.dense(concat, self.action_size, kernel_initializer=initializer)

    # create a dictionary of trainable vars by their name
    trainable_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope=scope.name)
    trainable_vars_by_name = {var.name[len(scope.name):]: var for var in trainable_vars}
    return output, trainable_vars_by_name
```

Figure 20: Creating the model trained on Pixel and Ram input.

```
"""
CREATING THE AGENT
"""
class QLearningAgent():
    def __init__(self, env):
        self.action_size = env.action_space.n
        self.observation_size = (96, 80, 1)
        self.learning_rate = 0.00025  # higher for experience replay
        self.discount_rate = 0.95
        self.checkpoint_path = "seaquest_pixel.ckpt"  # where to save model checkpoints
        self.min_epsilon = 0.1  # make sure it will never go below 0.1
        self.epsilon = self.max_epsilon = 1.0
        self.final_exploration_frame = 100000
        self.loss_val = np.infty  # initialize loss_val
        self.error_val = np.infty
        self.replay_buffer = PrioritizedReplayBuffer(maxlen=100000)  # experience buffer
        self.tau = 0.001

        tf.reset_default_graph()
        tf.disable_eager_execution()

        # observation variable - takes shape 96 by 80
        self.X_state = tf.placeholder(tf.float32, shape=[None, 96, 80, 1])
        # create two deep neural network - one for main model one for target model
        self.main_q_values, self.main_vars = self.create_model(self.X_state, name="main")
        self.target_q_values, self.target_vars = self.create_model(self.X_state, name="target")

        # update the target network to have the same weights as the main network
        self.copy_ops_hard = [targ_var.assign(self.main_vars[targ_name]) for targ_name, targ_var in self.target_vars.items()]
        self.copy_ops_soft = [targ_var.assign(targ_var * (1. - self.tau) + self.main_vars[targ_name] * self.tau) for targ_name, targ_var in self.target_vars.items()]
        self.copy_online_to_target = tf.group(*self.copy_ops_hard)  # group to apply the operations list

        # we create the model for training
        with tf.variable_scope("train"):
            # variables for actions (X_action) and target values (y)
            self.X_action = tf.placeholder(tf.int32, shape=[None])
            self.y = tf.placeholder(tf.float32, shape=[None])
            self.importance = tf.placeholder(tf.float32, shape=[None])

            self.q_value = tf.reduce_sum(self.main_q_values * tf.one_hot(self.X_action, self.action_size), axis=1)

            # calculate error and loss function
            self.error = self.y - self.q_value
            self.loss = tf.reduce_mean(tf.multiply(tf.losses.huber_loss(self.y, self.q_value, reduction='none'), self.importance))

            # global step to remember the number of times the optimizer was used
            self.global_step = tf.Variable(0, trainable=False, name='global_step')
            self.optimizer = tf.train.AdamOptimizer(self.learning_rate)
            # tell optimizer to minimize loss, the function will also add +1 to global_step at each iteration
            self.training_op = self.optimizer.minimize(self.loss, global_step=self.global_step)
```

Figure 21: Creating the Agent class.

```
""" ------- CHOOSING AN ACTION -------"""
def get_action(self, state):
    q_values = self.main_q_values.eval(feed_dict={self.X_state: [state]})

    # slowly decrease epsilon
    self.epsilon = max(self.min_epsilon, self.max_epsilon - ((self.max_epsilon - self.min_epsilon)/
                                                             self.final_exploration_frame)*self.global_step.eval())

    if np.random.rand() < self.epsilon:
        return np.random.randint(self.action_size)  # choose random action
    else:
        return np.argmax(q_values)  # optimal action

""" ------- TRAINING -------"""
def train(self, experience, batch_size=32, priority_scale=0.0):
    self.replay_buffer.add(experience)  # add experience to buffer

    # extract an experience batch from the buffer
    (state, action, next_state, reward, done), importance, indices = self.replay_buffer.sample(batch_size, priority_scale=priority_scale)

    # compute q values of next state
    next_q_values = self.target_q_values.eval(feed_dict={self.X_state: np.array(next_state)})
    next_q_values[done] = np.zeros([self.action_size])  # set to 0 if done = true

    # compute target values
    y_val = reward + self.discount_rate * np.max(next_q_values)

    # train the main network
    importance = (importance**(1-self.epsilon)).reshape((importance.shape[0],))
    feed = {self.X_state: np.array(state), self.X_action: np.array(action), self.y: y_val, self.importance: importance}
    _, self.loss_val, self.error_val = self.sess.run([self.training_op, self.loss, self.error], feed_dict=feed)
    self.replay_buffer.set_priorities(indices, self.error_val)
```

Figure 22: Creating the train and get_action function for the Agent class.

```
agent = QLearningAgent(env)
episodes = 500  # number of episodes
copy_steps = 10000  # update target network (from main network) every n steps
save_steps = 10000  # save model every n steps
list_rewards = []
frame_skip_rate = 4

with agent.sess:
    for e in range(episodes):
        state = prep_obs(env.reset())
        done = False
        total_reward = 0
        losses = []
        i = 1  # iterator to keep track of steps per episode - for frame skipping and avg loss
        action = 0  # do nothing at start - no states yet
        while not done:
            step = agent.global_step.eval()

            # get a new action every X frames (frame skipping)
            if i % frame_skip_rate == 0:
                action = agent.get_action(state)

            # get outcome of action - perform last action for X steps
            next_state, reward, done, info = env.step(action)
            next_state = prep_obs(next_state)
            reward = np.sign(reward)  # in reward clipping all positive rewards are +1 and all negative is -1

            # train model every X frames (frame skipping)
            if i % frame_skip_rate == 0:
                agent.train((state, action, next_state, reward, done), priority_scale=0.8)

            state = next_state
            total_reward += reward

            # regulary update target DQN - every n steps
            if step % copy_steps == 0:
                agent.copy_online_to_target.run()

            # save model regularly - every n steps
            if step % save_steps == 0:
                agent.saver.save(agent.sess, agent.checkpoint_path)

            i += 1
        print("\r\tEpisode: {}/{},\tStep: {}\tTotal Reward: {}".format(e + 1, episodes, step, total_reward))
        list_rewards.append(total_reward)
        pickle.dump(list_rewards, open("results/pixel_seaquest_test.p", "wb"))

plt.plot(list_rewards)
plt.show()
```

Figure 23: Running the training environment.