CS5079 - Assessment 1

Introduction to this Assessment

In the field of Artificial Intelligence, Reinforcement Learning (RL) is of special interest due to its ability to overcome the Bayes error rate; a supervised agent's capability will be dependent on the quality of the dataset it is fed. And thus, the agent cannot conceivably exceed the performance of a subject matter expert in the same task (for example, in the dog vs cat example, a human will always be near 100% accurate but not necessarily a CNN that's trained on data labelled by a human) (Ng, 2018). Thus, an advantage of RL is that it can overcome this limitation and achieve performance that is not within the reach of a human subject matter expert. Through more recent developments in both hardware and in software; interest in RL has resurged. Of note, Mnih et al. (Mnih et al., 2013) combined a CNN with an RL algorithm to train an agent to play seven vintage Atari 2600 games. The Seaquest Atari game was one of the platforms that the authors used to assess the performance of their agent. In this work, we will be presenting our proposed solutions to tasks 1, 2, and 3 of the first assessment in the CS5079 Applied Artificial Intelligence module. All tasks involve building and training an agent that is able to play the game according to the following rules: 1) The agent, the submarine, must not run off air 2) the agent needs to rescue divers and bring them to the surface before the air runs out. 3) the agent is free to defend itself from enemies by using torpedoes and evasive tactics to achieve goals 1 and 2.

The Seaquest game is simulated through the Arcade Learning Environment (ALE) (Bellemare et al., 2013). In our implementation we use Python's OpenAI Gym library to build the agent and the learning algorithm (Brockman et al., 2016). Furthermore, we make extensive re-use of the code used by Dr. Yun Bruno and code found in the GitHub repository associated with the text book: *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*

Task 1: Reinforcement Learning from the Screen Frames

1) First introduced in 1977, the Atari 2600 game console introduced a wide range of successful video games that were commercially successful throughout the 80's. The use of these legacy games to test AI agents is common in the literature. Many of the research work in the area of Reinforcement learning (RL) uses the games as testing modalities for new developed RL algorithms. Here we use the Seaquest V0 game as our testing modality. We begin by importing the OpenAI gym library to the Python workspace. OpenAI is a toolkit that enables developers to manipulate the behavior of the agents in the Atari games. Code 1.0 snippet is in Appendix.

Furthermore, we take note of the following: 1) The observation space of the game is an RGB image of size (210, 160) with pixel values ranging from 0 to 255. 2) The action space available to the agent is 18; that is, the agent is able to select from any of 18 unique actions. We gain access to this information via *code 1.1* (Appendix). The reward is the expected reward associated with a future action and can be accessed via the step function in the env class of openAI gym. This is shown in code 1.2 in the Appendix. Moreover, we print out the environment's info dictionary as shown in code 1.3. The info dictionary provides useful information for debugging the model, however, such information will not be used for training the RL agent. The Episode in the code defines each attempted game run, in other words, all the dynamics between states, actions and rewards taking place between the first and the last final state.

2) Classical Q learning algorithm states that an agent at any given state seeks to select future actions that are associated with the maximum possible reward among the range of available actions. This is formulated as follows (Bellman, 1966):

$$Q^{new}(s_t, a_t) \leftarrow Q^{old}(s_t, a_t) + \alpha\,[\, r(s_t, a_t) + \gamma . max(Q(s_{t+1}, a) - Q^{old}(s_t, a_t)\,]$$

Where Q^{new} is the new Q value updated via Q^{old} value from a previous state and action, the learning rate α which determine to what extent the newly gained information replaces the old information, r as the immediate reward for the action, γ as the discount rate that determines to what extent the reward in the future contributes to the Q function value behavior in the present, the max operator will choose the highest possible value of the Q function; that is the highest value that could be generated from selecting one action from the available range of immediate actions would be selected. Figure1.1 illustrates how the current and future actions of the agent are dependent on its current and past states (in turn, a function of observations and rewards of previous and future actions). Using the classical Q learning to train the Seaquest agent is impractical due to the large Q table that needs to be created and accessed. In contrast, using a Deep Learning as the modality through which the agent chooses the action is far more conducive for the purposes of this assessment.

3) Preprocessing the data is an essential step in many machine learning models. Decreasing the size of the input, whilst retaining important information will improve the performance of the model. The input image matrix to the network is shortened by deleting rows and columns that

are not relevant to the gameplay (side bar, logo, dead-space at the bottom). Following that, 128, the set's mean, is subtracted from each pixel value to create a range of [-128,127]. Finally, each pixel is divided by 128 to reduce the range to [-1,1] just before it's fed to the DQN network; a range that yields performance improvements in many DL models. The relevant code snippets can be found in the appendix code 1.4.

4) The implementation of the agent uses a double q-learning approach to learn from the experiences. The algorithm employs the Double Q-learning method in (Van Hasselt et al., 2015) in order to alleviate the effects of overestimation of the approximator function; partially caused by the use of layers rather than a traditional Q-table (Thrun and Schwartz, 1993). A thorough explanation of the model and the agent can be found in file Assessment1Task1withExperience.ipynb. In short, the agent uses the q-network to determine the q-values for the current state of the environment.

The agent employs the q-values to perform an action by choosing the action with the highest q-value. Alternatively, a random action is executed by the agent. The likelihood of performing the best action or a random action is dependent on the value of **epsilon** $\boldsymbol{\varepsilon}$. The closer epsilon ε is to the value of 1 the more likely a random action is chosen. The act of performing a random action is called exploration. This prevents the agent from remaining in a local optimum. With time, epsilon ε decreases (in this example towards 0.1) which makes the agent more reliant on the learnt best action. When using an action based on the q-values the agent is exploiting.

Next the effects of the action on the environment are obtained and used as the next state. This information, the prior state, the action, the reward and the game state are given to the agent to train the models, either directly or by storing it as an experience in the memory. The experiences are used to train the online neural network. The effects of using an algorithm with and without experience can be seen in subtask 1.5.

The online network is trained by retrieving the maximum next state's q-value from the target network. In order to calculate the long-term expected reward the Bellman equation is used.

$$Q(s_i,\ a_i) = r_{i+1} + \quad * max\ Q(s_{i+1})$$

The parameter **discount rate** $\boldsymbol{\gamma}$ controls the effect of uncertainty in future rewards. The further in the future the reward is the less it falls into account when calculating the current long-term reward. Because the model does not represent the Bellman equation (in the beginning) the values have to be adapted with the goal to approximate the Bellman equation for all states. To update the neural network's parameters the momentum optimizer is used. This optimizer uses a learning rate, momentum and employs Nesterov momentum.

5) In the training process different parameter settings were explored. These can be observed in the table down below.

Training steps	Epsilon ε	Discount rate γ	Learning rate α	Momentum	Nesterov	Experience
1,000,000	Linear*	0.99	0.001 and 0.0001	0.95	Yes	Yes and No

*The epsilon was used with the values from the tutorial *(`max(0.1, 1 - (0.9/2000000) * step)`)*. Note, that in order to have a continuous value from 1 to 0.1 we would either have to change the denominator or increase the training steps. When using 1,000,000 steps the minimum value of epsilon will be 0.5. With this approach we will have quite a high likelihood of exploration even in the later state of the training.

There are many other possible parameter settings. For example, a change in epsilon ε, e.g. making the decay exponential or reward based may allows the agent to learn more for a longer period of time. This would be especially interesting when letting the agent train for more than 1,000,000 steps. However, evaluating the agents' performances with the aforementioned settings already reveals some interesting results which are discussed below.

Using a learning rate of 0.001 and no experience results in the rewards that can be seen in figure 1.2. In contrast figure 1.3 shows the agent with the same learning rate but employing experience replay.

From the graphs we can infer that both agents successfully increase their reward. This can be best observed in the right-hand side of the figure where we are taking the average over the last 100 episodes. The agent without experience has the best average reward around a value of 115, whereas the agent with experience managed to increase its average reward to almost 200. Thus, the agent with experience achieves a higher reward due to the actions it performs. Another interesting thing to observe is how many steps each episode lasted for. This can be seen in figure 1.4 and shows that the agent with experience survives longer on average than the agent without experience. The strategies used by the agents can be seen in the folder video.

Because the agent using experience learns multiple times from the same experience it might be worth evaluating the model with a smaller learning rate. The results of the agent with experience replay and a learning rate of 0.0001 can be observed in figure 1.5. We can see that the reward is considerably higher than in the other parameter setting. The maximum average reward is around 275. Additionally, the convergence seems more stable as the average reward almost monotonically increases. In figure 1.6 we can observe that the number of steps per episode is clearly increasing over time as well.

One interesting observation regarding all performed experiments was that the loss did not really decrease but rather seemed to increase. This is seen in figure 1.7 which shows that phenomenon for the agent with experience replay and a learning rate of 0.0001.

Bonus - Frame Skipping

Adding a Frame Skipping capability to the code of the experience replay improves execution speed by over 100%. For 1000 episodes, we find that when using Frame Skipping, execution time is 5.4 seconds as opposed to 10 seconds for conventional replay. Code and a brief discussion can be found in the file - *Assessment1Task1withExperienceBonus.ipynb*

Task 2: Reinforcement Learning with RAM

1) Similar to task 1, we here use the DQN algorithm to train the Seaquest agent. However, we here utilize the RAM states of the virtual console as an input rather than using an RGB image. Using the RAM as an input for the Seaquest game instead of an image is beneficial. Using the RAM's 128 bytes representation of the environment, instead of a grey-scale image, makes the training faster due to a smaller size input. However, using the RAM input skips the screen input which may cause irretrievable information. From figure 2.1, the Seaquest Atari environment is called to train the agent by importing “Seaquest-ram-v0”, after unwrapping the environment, a 128 byte array is used as an input to the DQN where each byte has a range between 0 and 255. In figure 2.2 we show the code and a plot of bytes against their values at the start of the game.

Moreover, in figure 2.2 we use the wrapped method to extract the values for each byte, the graph is plotted using the matplotlib library. Here we pre-processed the input to the DQN by filtering only the most relevant RAM inputs to the agent. The code in figure 2.3, filters out absolute RAM values that are greater than a threshold of 100. All other values below the threshold are set to zero. Only 29, out of 128, inputs remain. Descriptively, we note that the maximum input value is 255, the minimum is 0 and the mean is 55.7 with a standard deviation of 91.5 (figure 2.3).

2) The code for these tasks can be found in the *Assessment1Task2.ipynb* files. There are two versions for the different learning rates used for this task. The first modification made to accommodate the Seaquest-ram-v0 environment was to be able to handle RAM input rather than an image - this meant that the image preprocessing method was no longer required, so this was removed from the code. Secondly, the shape of the RAM output is very different to the image output, with the shape of the environment being an array size [1, 128], compared to the image input of 210×160 RGB frames. To normalise the RAM outputs to [0, 1] range, the output state values are divided by 255. Moreover, the structure of the network was changed from a CNN to a dense neural network, based on the architecture proposed by Bonilla et al (2016) for designing a neural network for RAM Atari environments, with four dense layers that use a ReLU activation function. The code for the Q-network for RAM is seen in figure 2.4, and has been implemented with Tensorflow v1, in preparation for developing a hybrid model in task 3. The resulting network layers are a reshape layer, which transforms the input into a [1,128] tensor, with the next three layers being the ReLU activated dense layers, which keep the same shape. The next layer is the ReLU activated dense layer, which transforms the output to [1,6]. This is finally passed to the output layer. Figure 2.4 in the appendix provides the relevant code extracts.

3) The agent was trained over 1,000,000 training steps twice, first with a learning rate of 0.001, which resulted in 1,314 episodes, and the second with a learning rate of 0.0001, which resulted in 1,201 episodes. These were the same parameter values used in task 1 in order to be able to establish performance improvements or deteriorations between task 1 and 2 agents.

As shown in figure 2.5, both agents saw an increase in reward through the episodes. The agent with a learning rate of 0.001 peaked at around 400 for reward, and the agent with a learning rate

of 0.0001 peaked around 700, showing that the lower learning rate was able to achieve higher rewards. Figure 2.6 shows averaged rewards for both agents for every 100 episodes, which shows that the average rewards towards the end of the training period improved for both agents, with the lower learning rate achieving higher reward values. However, the lower learning rate agent did experience a large decrease towards the end - this is potentially due to the small number of episodes that would be in the last sample, and the averaging algorithm used. Figure 2.7 shows that for both agents, the number of steps per episode increased during the training period, with both agents starting around 750 steps per episode. The agent with the 0.001 learning rate averaged over 1,000 steps per episode at the end of the training period, and the agent with the 0.0001 learning rate averaged around 1,200 steps per episode.

Combining this with the increase in rewards, we can say that the agents did improve in terms of rewards over the course of the training. The agent with the lower learning rate was earning higher rewards, and was able to play the game for a larger number of steps per episode, showing that a lower learning rate achieved better performance. However, due to time constraints, the number of training steps was kept relatively low, at 1,000,000 steps. Compared to work by Sygnowski and Michalewski (2016), who ran their models for 1-3 days, this was not a timescale that was feasible for this assessment, and this is one area of potential refinement for the agent. The learning rate could also have been further reduced, with a longer timescale and more steps, to refine the model, as well as further adjusting parameters in the creation of the Q-network - increasing the number of layers, and including dropout in the dense neural network. The key improvement would be to increase the number of steps, which will also increase the number of runs that the agent can attempt, to provide the model with more experience of the game, and how it can optimise its score.

4) As for now, two different agents are implemented, where the first agent's input was only the information from on the screen frames and the second agent is trained and evaluated using the RAM only as input. This task objective is to compare and conclude the results of both agents trained using the different methods.

Looking at figures 2.8 and 2.9, for both learning rates, both types of game output show improvement in the average number of rewards per episode. For a learning rate of 0.001, the screen output had peak rewards of around 200, whereas the RAM outputs had a peak of just over 110. The RAM model also showed much slower improvement, compared to the screen model. For the lower learning rate of 0.0001, the difference in performance between the screen and RAM outputs becomes more pronounced, with the screen outputs gaining rewards of over 275, compared to RAM outputs reaching 160. This shows that the screen output models have a more stable learning pattern, with better performance observed, and that the screen models are faster at learning compared to the RAM ones.

Looking at figure 2.10, it can be seen that the agent is being trained on 1 million episodes for each method, by looking at the Screen Frame Input, it is visualised that when more runs occur, the loss is increasing gradually when reaching the final state. Unlike the loss per run visualised on the agent with the RAM input, the loss changes steadily from the first till the last episode by

increasing and decreasing over time. This indicates that the RAM input is more reliable when looking at the loss rate.

To conclude, when the agent is trained using the Screen Frame input, using all the figures given above the results are superior, where the rewards obtained are greater and the loss decreases gradually to be reduced less than the RAM Input.

Task 3: Reinforcement Learning by Mixing Screen and RAM

1) The screen frames are retrieved from the Seaquest-v0 environment. To retrieve the RAM the following method is used to get the associated ram values from the screen frame environment: unwrapped._get_ram().
The model consists of 3 convolutional layers that receive the preprocessed screen frame as input. The output of the last convolutional layer is flattened and passed through a dense layer with 512 units with relu as activation function. The output of that layer is concatenated with the preprocessed ram input and passed through 3 dense layers with 128 units each with relu activation followed by one dense layer with 6 units. The output layer of the neural network is another dense layer. The overall architecture has been inspired by the paper Learning from the memory of Atari 2600 (Sygnowski and Michalewski, 2016). The number of units and layers are the same used in task 1 and 2 to increase the comparability between the different approaches.
The code can be found in the file *Assessment1Task3.ipynb*.
The experiment setting was as follows:

Training steps	**Epsilon ε**	**Discount rate γ**	**Learning rate α**	**Momentum**	**Nesterov**	**Experience**
1,000,000	Linear*	0.99	0.001 and 0.0001	0.95	Yes	No

*The epsilon was used with the values from the tutorial *(`max(0.1, 1 - (0.9/2000000) * step)`).* Note, that in order to have a continuous value from 1 to 0.1 we would either have to change the denominator or increase the training steps. When using 1,000,000 steps the minimum value of epsilon will be 0.5. With this approach we will have quite a high likelihood of exploration even in the later state of the training.

2) The following is a discussion of the experiments' results. In figure 3.1 we observe the rewards per episode and the average reward over 100 episodes for the agent with a learning rate of 0.001. The reward is increasing over time to around 120 after 1,000,000 steps. Figure 3.2 shows the rewards for the agent using a learning rate of 0.0001. We observe a similar trend for the reward after 1,000,000 steps.

Another interesting factor to observe is the duration of survival. Hence, figure 3.3 shows the number of steps per episode over an average of 100 for both learning rate settings. We observe that for both trained agents the number of steps increased on average with the number of steps. When comparing figure 3.3 to the previous figures 3.1 and 3.2, we note that, on average, the longer the agent survives on average, the higher the average reward.

In summary, we observe that in both experiments the average reward increased above a value around 120. The graphs indicate that with both learning rates the training will probably yield

better results with a higher number of steps as the best performance was achieved around the 1 million steps mark with an upward trend.

3) The table below is used to compare the results of all the used methods when including different learning rates and average number of steps:

	Number of episodes	Learning Rate	Results
Screen Frame Input	1×10^6	0.001 & 0.0001	An examination of the out runs in task 1 when an image was used as input, we observe that the rewards and runs positively correlate. An increase in the runs yields an increase in the rewards.This pattern can be clearly seen in figure 2.6. The figure shows that better results are obtained with the learning rate drops from 0.001 to 0.0001
RAM Input	1×10^6	0.001 & 0.0001	A further examination between the two inputs indicates that the starting values of rewards at a larger learning rate, 0.001, are smaller than when the smaller learning rate of 0.0001 is used; however the reward rate displays less variance, is more stable and reaches a higher final reward than when the latter is used. In contrast, using a smaller learning rate yields higher starting rewards; however the reward rate drops gradually with the episodes.
Combination Input	1×10^6	0.001 & 0.0001	When combining both inputs together, the rewards per run and steps per episode steadily increase throughout the training cycle. We take note here that when the learning rate was decreased by 10 folds to 0.0001; it is observed in the output that the values of the independent inputs are higher than the value of the combined one.

In conclusion, the evidence in our figures shows that all methods perform better when the learning rate is set to 0.0001. Moreover, combining both inputs to the deep learning layer yields a worst performance when measured against training time, rewards gained in fewer steps over few game plays (episodes or runs) when trying individual inputs. When viewed from a computational cost perspective, using RAM states as an input can be viewed as a better method. The computational time is faster and the results are better in terms of rewards. However, the caveat here is that although using Screen Frame input is faster and more efficient it is susceptible to causing irrecoverable loss of information; this is likely the main factor behind the significant performance

improvement witnessed in the final output using the Screen Frame input than the RAM Input and the combined.

References

- Bellemare, M.G., Naddaf, Y., Veness, J. and Bowling, M., 2013. The arcade learning environment: An evaluation platform for general agents. Journal of Artificial Intelligence Research, 47, pp.253-279.
- Bellman, R., 1966. Dynamic programming. Science, 153(3731), pp.34-37.
- Bonilla, E., Zeng, J., and Zheng, J., 2016. Asynchronous Deep Q-Learning for Breakout with RAM inputs
- Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W., 2016. Openai gym. arXiv preprint arXiv:1606.01540.
- Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.
- Ng, A., 2017. Machine learning yearning. URL: http://www. mlyearning. org/(96).
- Schaul, T., Quan, J., Antonoglou, I. and Silver, D., 2015. Prioritized experience replay. arXiv preprint arXiv:1511.05952.
- Sygnowski, J. and Michalewski, H., 2016. Learning from the memory of Atari 2600. In Computer Games (pp. 71-85). Springer, Cham.
- Thrun, S. and Schwartz, A., 1993, December. Issues in using function approximation for reinforcement learning. In Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum.
- Van Hasselt, H., 2010. Hasselt. Double q-learning. Advances in Neural Information Processing Systems, 23, pp.2613-2621.
- Van Hasselt, H., Guez, A. and Silver, D., 2015. Deep reinforcement learning with double q-learning. arXiv preprint arXiv:1509.06461.

Appendix

Task 1

```
import gym
env = gym.make("Seaquest-v0")
```

Code 1.0 snippet for task 1.1. Here, openAI gym is imported to the Python workspace and the Seaquest-v0 game is loaded as the environment.

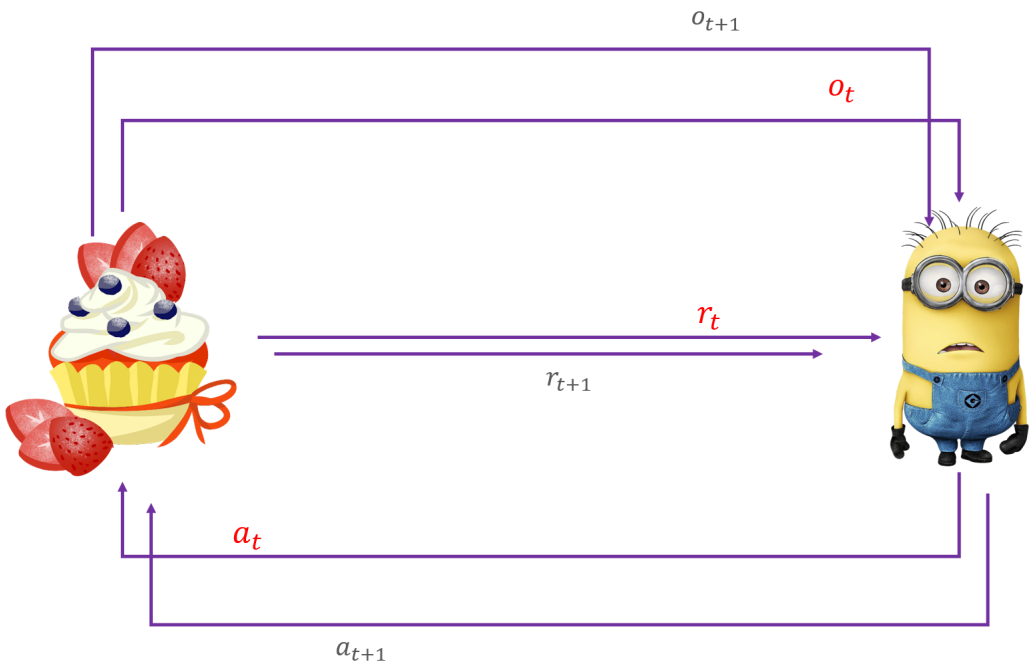

Figure 1.1: The agent receives an observation o_t from the environment. The agent executes an a_t action based on an expectation of reward from the environment r_t. The received reward and the next observation o_t alters the state of the agent such that the agent executes action a_{t+1} and so on.

```
print("Observation space:", env.observation_space)
print("Action space:", env.action_space)
for i in env.get_action_meanings():
  print(i)
```

Code 1.1 Python code for displaying the environment's observation space, the action space available to the agent and the actions available.

```
next_observation, reward, done, info = env.step(action)
```

Code 1.2 We pass the action to the step function in the environment to gain access to the resulting observation (t+1), resulting reward.

```
actions = np.arange(0,17)
obs = env.reset()
res = 0
info = []
for action in actions:
    _,_,_,res = env.step(action)
    info.append(res)
print(info)
```

Code. 1.3 Code for printing out the info dictionary at the beginning of the game before any runs.

```
def preprocess_observation(observation):
    img = observation[1:192:2, ::2] #This becomes 96, 80,3
    img = img.mean(axis=2) #to grayscale (values between 0 and 255)
    img = (img - 128).astype(np.int8) # normalize from -128 to 127
    return img.reshape(96, 80, 1)
```

```
prev_layer = X_state / 128.0 # scale pixel intensities to the [-1.0, 1.0] range.
```

Code 1.4: Code snippets for preprocessing the screen data

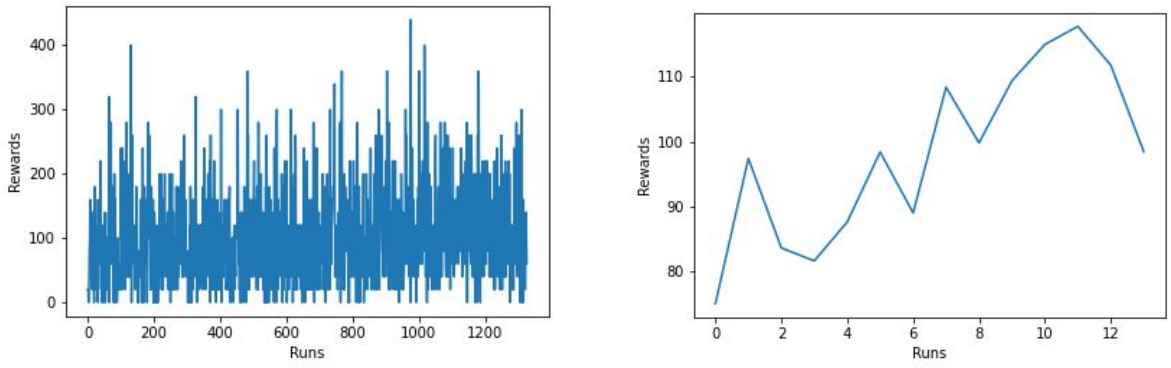

Figure 1.2: No experience, lr=0.001, Rewards per episode (left), Averaged rewards over 100 episodes (right)

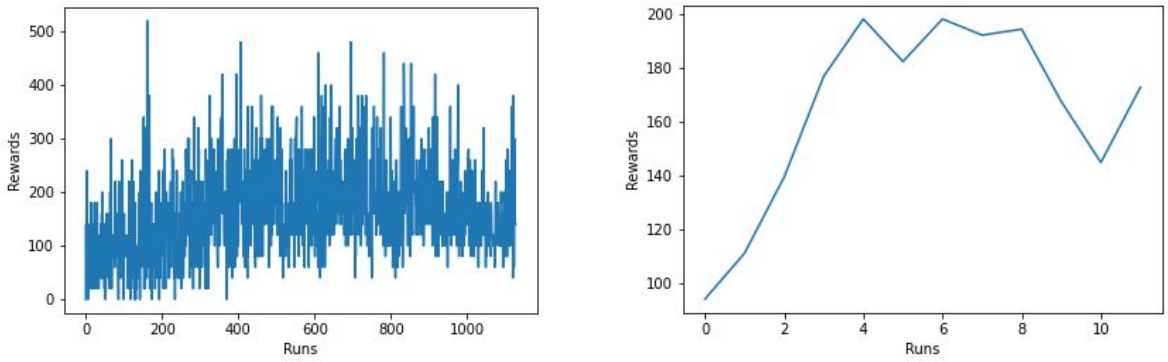

Figure 1.3: Experience, lr=0.001, Rewards per episode (left), Averaged rewards over 100 episodes (right)

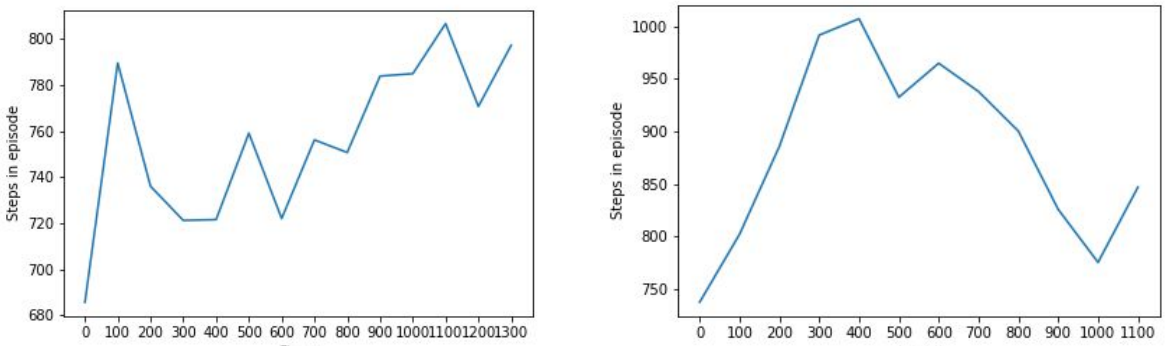

Figure 1.4: No experience, lr=0.001, Steps in episode (avg) (left), Experience, lr=0.001, Steps in episode (avg) (right)

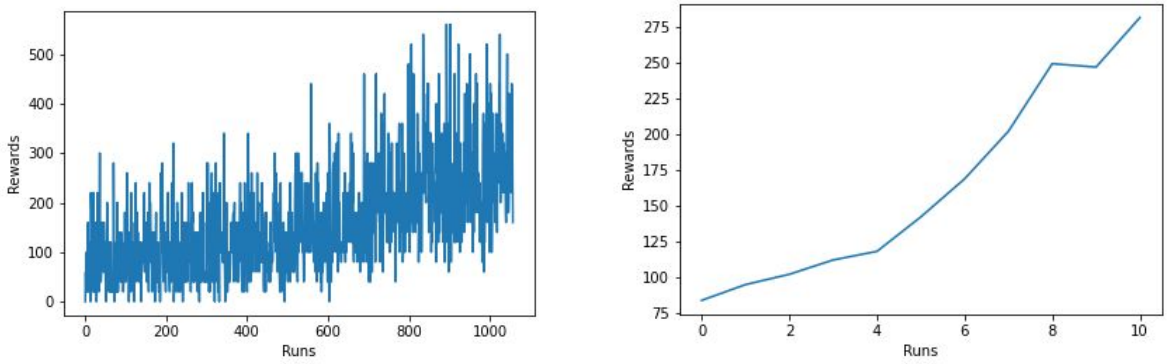

Figure 1.5: Experience, lr=0.0001, Rewards per episode (left), Experience, lr=0.0001, Averaged rewards over 100 episodes (right)

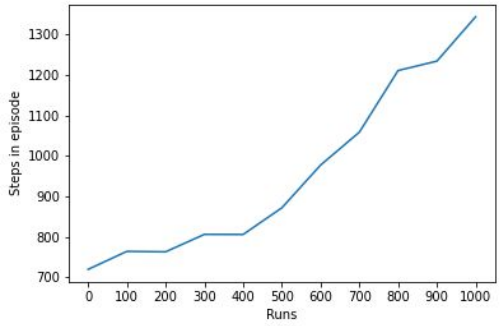

Figure 1.6: Experience, lr=0.0001, Steps in episode (avg)

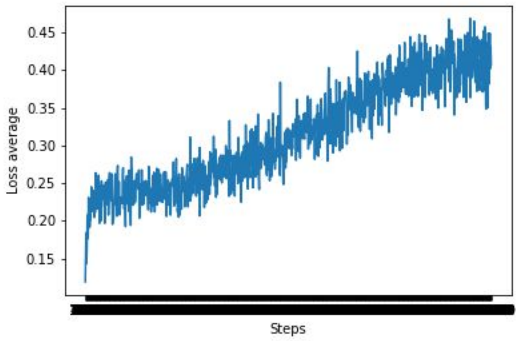

Figure 1.7: Experience, lr=0.0001, Loss averaged over 1000 steps

Task 2

```
[ ]   1 env = gym.make("Seaquest-ram-v0")
      2 env.reset()
```

```
array([  0,  67, 204,   0,  15,   0, 255,  26, 175, 144,  24, 170, 132,
         0,  12,   6,  50, 134, 212, 253,   0, 253,  86, 253, 164, 253,
        80, 254,   0, 254,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0, 200, 200, 200, 200,   0,   1,   2,   3,
       255, 255, 255,   0,   0,   0,   0,   3,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,  76,   0,   0,   0,   0, 101,  96,  48,
        32,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,  13,   0, 255, 255,   0,   0,   0,
         1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,  96,   7,   0,   0, 180, 215, 254, 214, 244], dtype=uint8)
```

Figure 2.1: Array including the RAM values of Seaquest

```
observation_image, reward, done, info = env.step(0)
observation_ram = env.unwrapped._get_ram()
env = env.unwrapped
env.reset()

plt.title('RAM Inputs Visualization',size=12)
plt.xlabel('Bytes (0-127)')
plt.ylabel('Array Values (0-255)')
plt.plot(observation_ram)
plt.show()
```

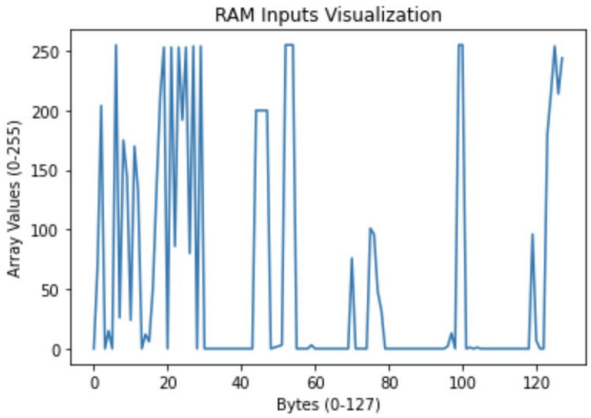

Figure 2.2: Visualising the value of each RAM byte.

In [10]:
```
temp=[]
temp_new = []
for i in range(0,127):
  if observation_ram[i]> 100:
    temp.append(observation_ram[i])
    temp_new.append(observation_ram[i])
  else:
    temp.append(0)

print(temp)
np.array([temp_new]).shape
```

```
[0, 0, 204, 0, 0, 0, 255, 0, 175, 144, 0, 170, 132, 0, 0, 0, 0, 134, 212, 253, 0, 253, 0, 253, 164, 253, 0, 254, 0, 254, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 200, 200, 200, 200, 0, 0, 0, 0, 255, 255, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 101, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 255, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 174, 215, 254, 214]
```

Out[10]: (1, 29)

In [11]:
```
print("Maximum Value:    ", observation_ram.max())
print("Minimum Value:    ", observation_ram.min())
print("Mean height:       ", observation_ram.mean())
print("Standard deviation:", observation_ram.std())
```

```
Maximum Value:     255
Minimum Value:     0
Mean height:        55.7421875
Standard deviation: 91.57922168928302
```

Figure 2.3: Displaying and Choosing most significant inputs

```python
def q_network(X_state, name):
    prev_layer = X_state/255 # scale pixel intensities to the [-1.0, 1.0] range.
    initializer = tf.variance_scaling_initializer()
    with tf.variable_scope(name) as scope:
        prev_layer = tf.reshape(prev_layer, shape=[1, 128])
        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)
        hidden = tf.layers.dense(prev_layer,6,
                                     activation=tf.nn.relu,
                                     kernel_initializer=initializer)
        outputs = tf.layers.dense(hidden, env.action_space.n,kernel_initializer=initializer)
    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 outputs, trainable_vars_by_name
```

Figure 2.4: The dense neural network for the RAM environment

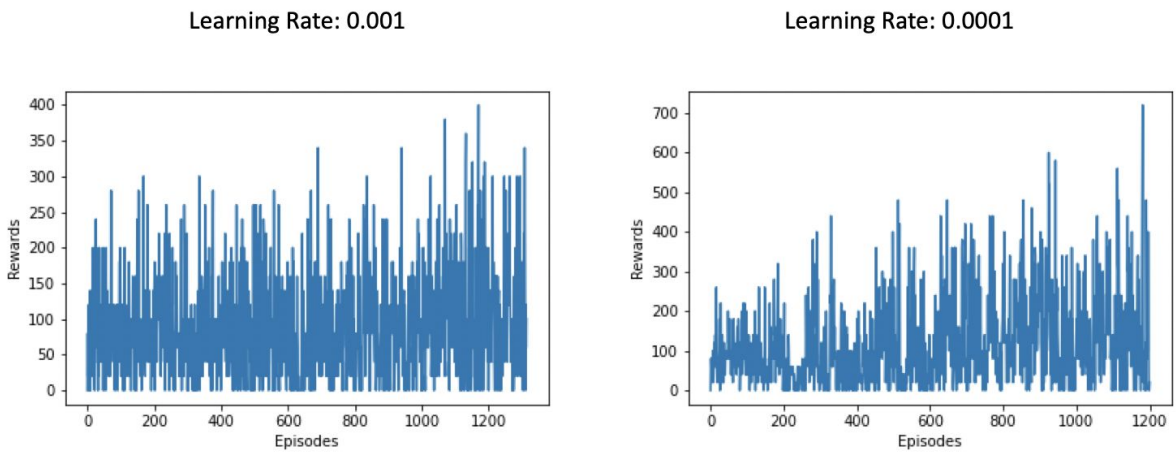

Figure 2.5: The reward for each episode

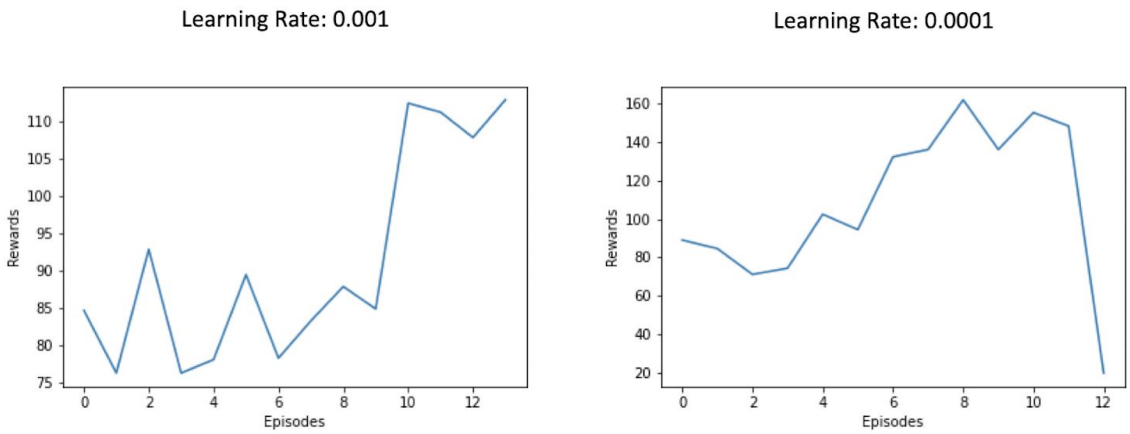

Figure 2.6: The averaged graph showing the rewards for every 100 episodes

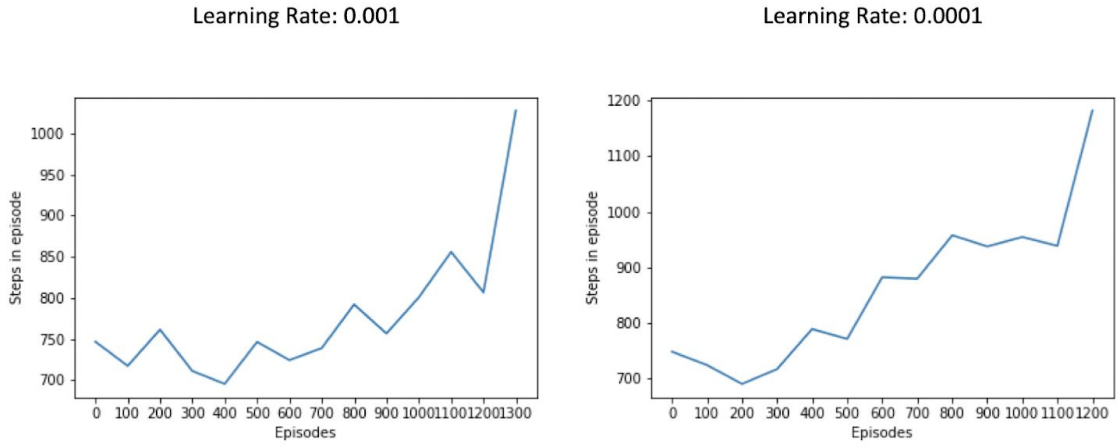

Figure 2.7: The number of steps in each episode, averaged over every 100 episodes

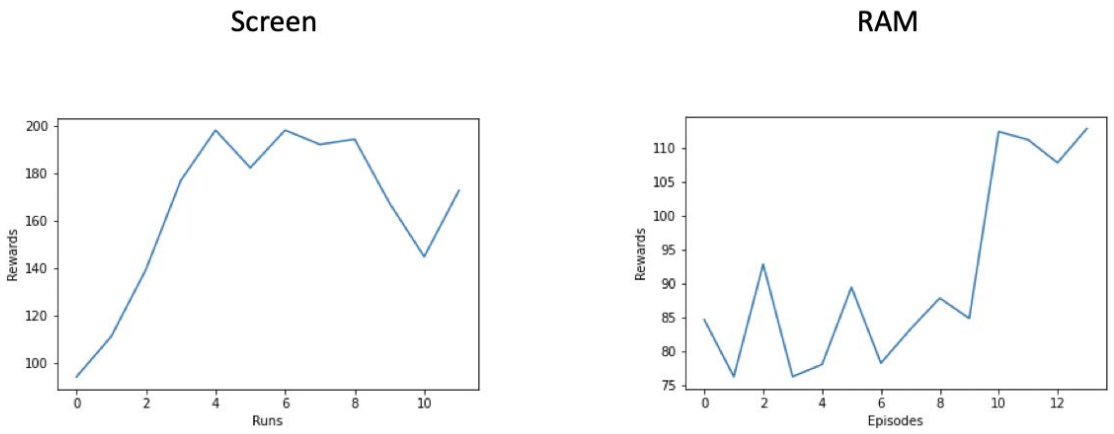

Figure 2.8: Rewards vs Runs graph for both methods (lr=0.001)

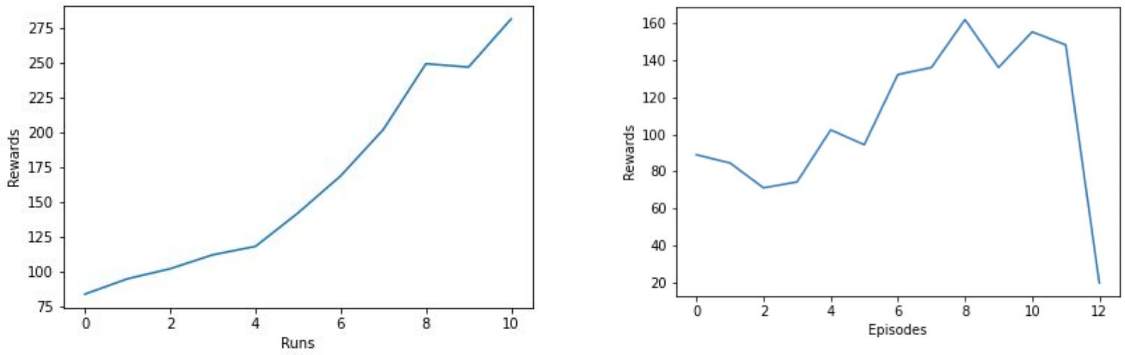

Figure 2.9: Rewards vs Runs graph for both methods (lr=0.0001), Screen (left), RAM (right)

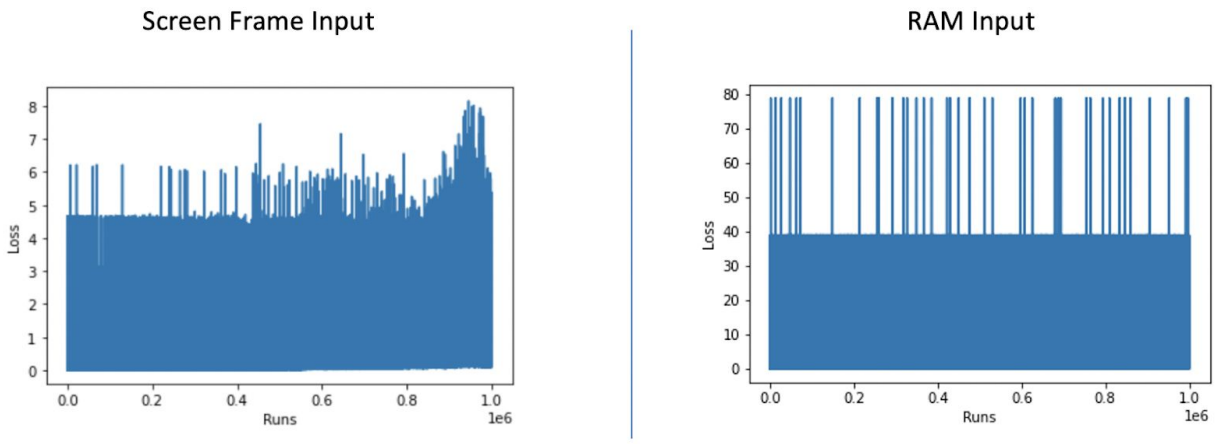

Figure 2.10: Loss vs Runs graph for both methods

Task 3

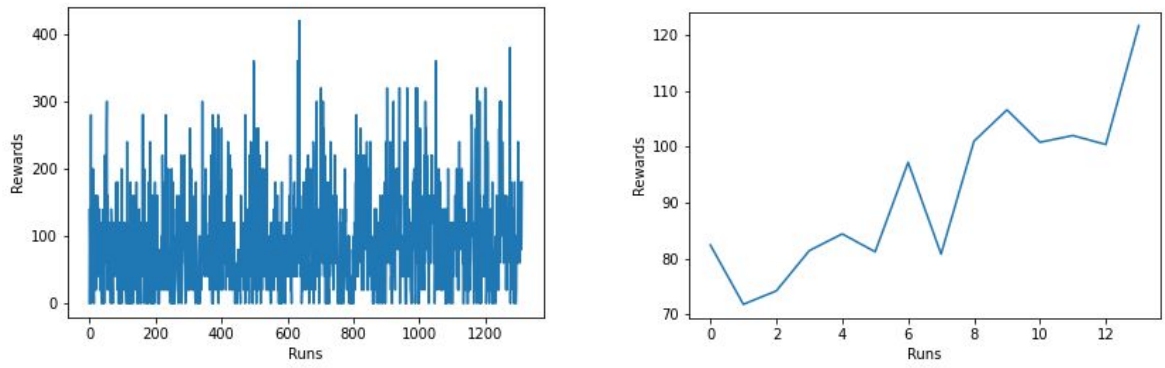

Figure 3.1: Rewards per episode (left), Averaged rewards over 100 episodes (right) (Learning rate: 0.001)

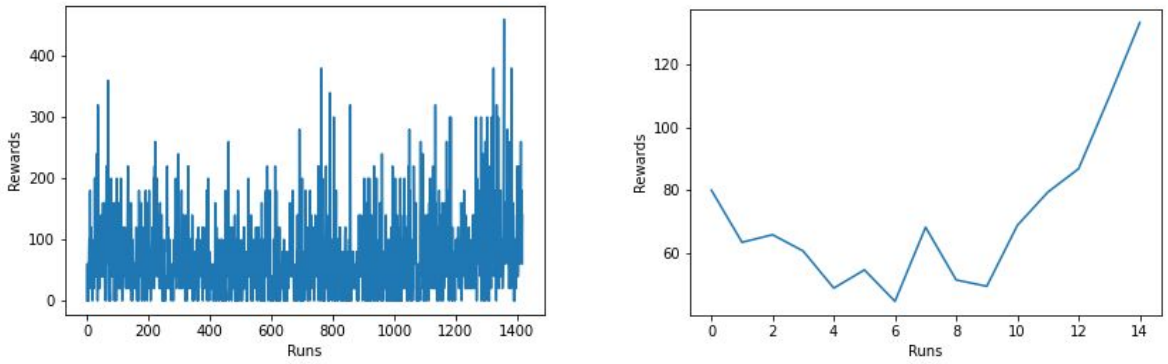

Figure 3.2: Rewards per episode (left), Averaged rewards over 100 episodes (right) (Learning rate: 0.0001)

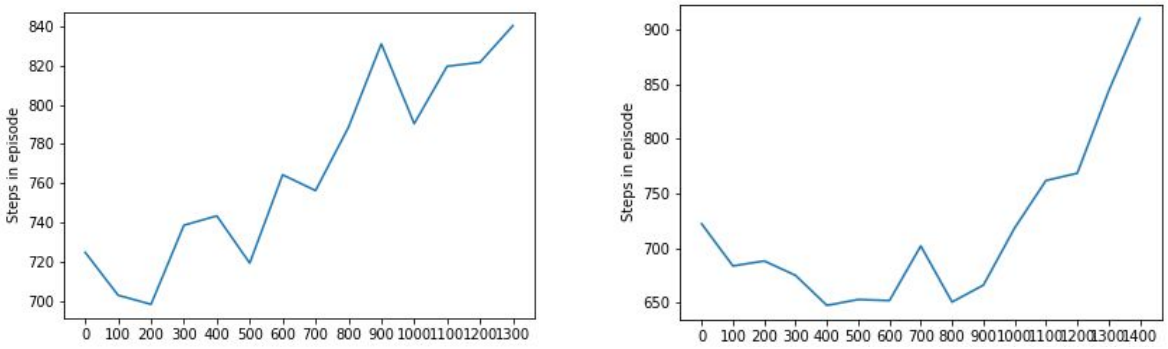

Figure 3.3: lr = 0.001: Steps in episode (avg) with (left), lr= 0.0001: Steps in episode (avg) (right)