CS 7642 Project 2: Lunar lander

Ying Huang yhuang836@gatech.edu

Git Hash: 71bd9f89e43ec2a86a8c47efaad49fc7f2e2c0ad

***Abstract*—This report discusses deep Q-learning and its implementation in solving a states-partially-observable problem, "Lunar Lander". In a lot of real-life simulating problems the full state is not available to the agent, and the generalization of the state will be very necessary. This report discusses the function approximation using an artificial neural network in reinforcement learning, which is known as deep neural network.**

***Keywords*—Deep Q-network, function approximation, artificial neural network**

I. Lunar Lander

Lunar Lander is a sophisticated problem in reinforcement learning, the agent should be trained to know how to land on the landing pad properly. Unlike the learning problems in previous assignments which are discrete MDP, the Lunar Lander has continuous states, which are represented by 8 variables:

(x, y, x˙, y˙, θ, θ˙, legL, legR)

The state variables x, y are the horizontal and vertical position, and x˙, y˙ are the horizontal and vertical speed. θ, θ˙ are the angle and angular speed of the lander. legL, legR are the binary values to show whether the left or right leg of the lander is touching the ground.

To solve this problem, the simple Q-table would be insufficient. After reading the papers, I implemented the deep Q-network (DQN), which is a combination of reinforcement learning and artificial neural network, to estimate the optimal Q function in continuous state space. The Q function will be represented by an artificial neural network, whose weights are trained to produce the optimal Q value for each action.

II. Q-Learning and Function Approximation

In MDP Q-learning, the approximation value of the actions in each state is stored in the Q-table. However, for continuous states, we need function approximation to generalize the value-state function. The parameterized functional form is using a weight vector $w \in \mathbb{R}^d$, and we can write $\hat{v}(s,w) \approx v_\pi(s)$ for a states s and given weights w. The function $\hat{v}$ can be a linear function or a nonlinear function like the polynomial function. But it also can be a function that is produced by an artificial neural network, in which the weights will be the weight for each node in the network.

To get the function approximation right, the function should update the function weight by shifting its prediction value to the target value. Let us refer to an individual update by the notation $s \to u$, where s is the state updated and u is the update target that s 's estimated value is shifted toward.[1] In reinforcement learning, the function update should happen while the agent is interacting with the environment, that's why it is critical to have an algorithm that can efficiently learn from the incrementing data. Also, it is worth mentioning that the target function may not be stationary. The target function will change over time, so the function approximation should also shift to different targets each time.

III. Prediction Objective ($\overline{VE}$)

As we assumed that in reinforcement learning there are more states than weights. So sometimes when we change the weights to fit one state, we might move far away from other states. We need to decide which decisions are more important than other ones. In Sutton's paper, a state distribution $\mu(s)$ is used to represent how much we care about that state. $\mu(s) > 0$, $\sum_s \mu(s) = 1$. By the error in a state s we mean the square of the difference between the approximate value $\hat{v}(s,w)$ and the true value $v_\pi(s)$. Weighting this over the state space by μ, we obtain a natural objective function, the *mean square value error*, denoted $\overline{VE}$:[2]

$$\overline{VE}(w) = \sum_{s \in S} \mu(s)\,[v_\pi(s) - \hat{v}(s,w)]^2$$

The square root of $\overline{VE}$ represents the error between the approximate values and the true value. However, we are not completely sure that $\overline{VE}$ is the right measurement for reinforcement learning because reinforcement learning has non-stationary target function. The function that minimizes the $\overline{VE}$ is not necessary the best function for solving the problem. But so far we haven't found a better alternative measurement.

IV. Artificial Neural Network

The artificial neural network is a popular method for function approximation, especially in reinforcement learning. A multi-layer neural network consists of a lot of interconnected nodes/perceptrons. Perceptron has some similar properties to neurons, which is where the neural network name comes from. The image below shows a generic feedforward artificial neural network:

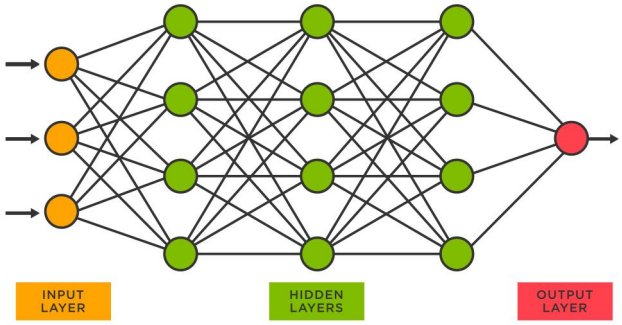

***Figure 1:** A feedforward ANN with 3 input units, 1 output unit, and 3 hidden layers.*

There are two types of ANN, recurrent ANN, and feedforward ANN. Recurrent ANN has loops in the network, which is not our focus in this report. Figure 1 shows the generic feedforward ANN. The information in a feedforward neural network only moves in one direction---from the input units to the output unit.

With the neural network, we can use state features as inputs, and pass them to a few hidden layers, finally producing 4 action values as the output.

V. Experience replay & Iterative update

Function approximation is generally a relatively difficult task in reinforcement learning. Especially in the lunar lander problem, the training data are sequences of observations, any changes may change the action selection and changes the weights in neural network nodes. Thus, we can address these problems with two key ideas. First, we used a biologically inspired mechanism termed experience replay that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution. Second, we used an iterative update that adjusts the action-values(Q) towards the target.[3]

The experience replay makes the neural network repeatedly trained by limited data, which is not what I used to do in supervised learning. And the nature paper explained that this algorithm is not efficient for large neural networks, which is a pitfall I fell into. Without knowing how large should a large neural network be, and in my past experience in neural networks I always use more than 2 hidden layers and each has hundreds of nodes, at first, I set my neural network to be 8 inputs, first hidden layer of 256 nodes, second hidden layer of 128 nodes, and 4 outputs. And the result is the agent will learn things super slow, it can't reach to a high average reward even after a thousand iterations. Finally, I set the network structure to be that 80 nodes in the first hidden layer and 40 nodes in the second hidden layer and solve the lunar lander problem.

When we train the Q-network, the loss function in iteration i is:

$$L_i(\theta_i) = E_{s,a\sim\rho(.)}[(y_i - Q(s,a;\theta_i))^2]$$

$$L_i(\theta_i) = E_{(s,a,r,s')\sim U(D)}[(r + \gamma max_{a'} Q(s',a';\theta_i') - Q(s,a;\theta_i))^2]$$

So in this process, I create a neural network for selecting the actions in each state and predicting the value of state $Q(s,a;\theta_i)$, and a optimal neural network which reflects the current true value $r + \gamma max_{a'} Q(s',a';\theta_i')$. To balance the exploration and exploitation, I use the ε-greedy strategy for action selection, the algorithm with pick the $max_{a'} Q(s',a';\theta_i')$ with probability $1-\varepsilon$ and select a random action with probability ε.

It's tricky to set a satisfying ε. I find the best way is to have a decaying ε. With iteration counts goes up, ε should decay and have less random action, but it should always keep some randomness for exploration. Another pitfall for me is that, I set the e function to be $\varepsilon = decay\_factor^{iter}$. When iteration is very big, ε decays to almost 0, it won't explore new situations. So by the end, my agent learned how to balance itself with left/right engines, but it didn't learn how to slow down when approaching the ground, and it crashed almost every time in the end.

So I tried more complicated ε decay function:

$$\varepsilon = \frac{10}{(sin(iter)+1.1)*iter+1}$$

The ε graph is shown below in Figure 2. I thought it's a good way to decay ε because the ε goes big and small periodically. It can optimize the neural network when ε is close to 0 and explore random actions when ε is bigger. However, when I try this ε function, the computing of ε is too complicated and expensive that it slows down the performance.

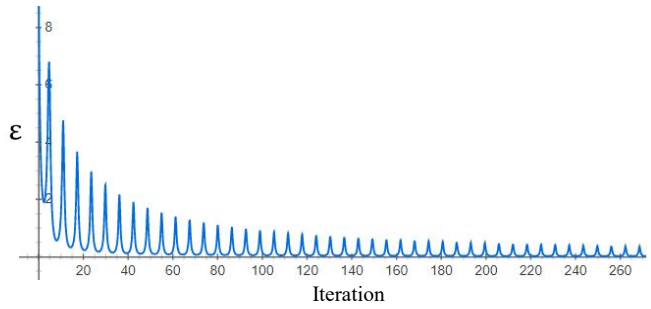

Figure 2: *Function graph of* $\varepsilon = \frac{10}{(sin(iter)+1.1)*iter+1}$.

After some tests, I finally set the ε decay function to be $\varepsilon = \max(0.01, 0.994^{iter})$. The ε will decay slowly as more training episodes happen, but it will be at least 0.01, which means it always has one percent chance to explore. This function is used in my algorithm to solve the lunar lander problem, and I will discuss about the ε decay function later in the report. The ε decay chart is shown below (Figure 3).

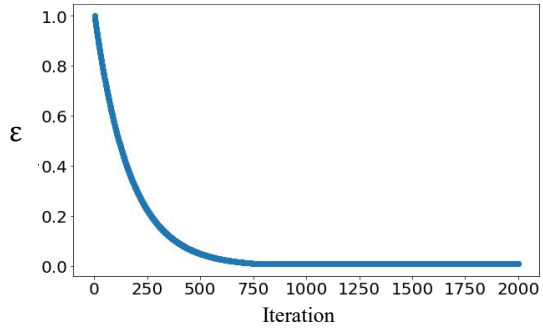

Figure 3: *Function graph of* $\varepsilon = max(0.01, 0.994^{iter})$

VI. Algorithm and Experiments

To implement the DQN strategies and solve the lunar lander problem, I implemented the algorithm in the paper "Human-level control through deep reinforcement learning", the algorithm is shown below (Figure 4).

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 4: Deep Q-learning with experience replay[4]

The algorithm is very straightforward, but I had countless pitfalls during its implementation. Some problems are pure mistakes like I should have updated Q every C steps, instead, I updated Q every C episodes, which lead to some progress at the first but the performance declined after. I also adjusted the parameters so many times, including neural network learning rate, the way of ε decay, replay memory D size, minibatch size, Q-learning discounting factor γ, and the hidden layer size of the neural network. My observation is that these parameters are related to the overall deep Q-network's learning rate and the maximum performance it can reach. My presumption is that the learning process should be slow and stable, let the Q follows the Q_optimal, otherwise they can easily diverge and won't improve the performance.

So finally I had one successful trial, making things slow and stable, I set the discount factor = 0.99, learning rate = 0.0003, epsilon decay factor = 0.994, memory replay storage size = 5000000, mini-batch size = 64, and the neural network has 2 hidden layers which have 80 and 40 nodes. The rewards for each training episode while training the agent is shown below (Figure 5):

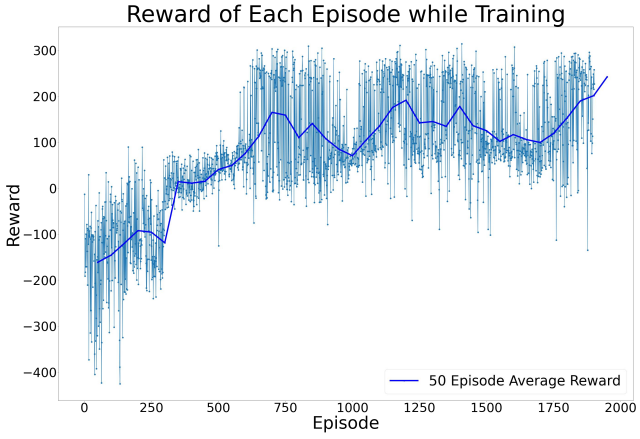

Figure 5: Reward of each episode while training the agent

The thinner line represents the reward per episode while the agent is under training. The reward curve is not a smooth line, instead, it goes up and down, it can be -100 this episode and goes 250 in the next episode. But we can observe the overall trend of the reward is improving with the increase of episodes. To observe the trend better, I plotted the average reward for every 50 episodes (the thicker line). The average reward curve shows the reward increases from -180 to 240 with the episode increasing, sometimes the reward decreases but it keeps bouncing back. We can draw the conclusion that deep Q-learning helps the agent to learn how to solve a problem, and it improves the performance in a limited time. The improvement is not a smooth line, it can have decreasing reward and it learns from the experience and improve later.

After training the agent, I performed 100 consecutive episodes on the trained agent and plotted the reward per episode below (Figure 6). The average reward for the 100 episodes is 205, we can observe that most rewards are near or above 200, and only a few rewards are under -100. Unlike a regular MDP, the trained model isn't guaranteed to always have a good performance. My assumption is that the lunar lander needs consecutive correlated actions to get a good reward, a bad action choice can influence the following actions and lead to a crash. Based on that, I believe that if the agent is trained more time, which improves the accuracy of each action selection, its performance will keep improving.

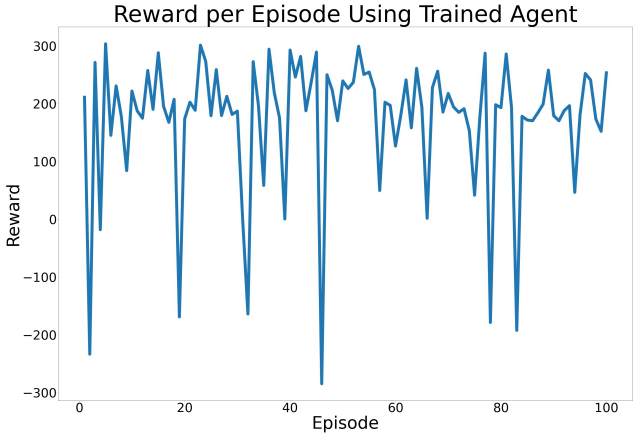

Figure 6: Reward of consecutive 100 episodes using trained agent

VII. Hyperparameters and Performance

I tried a lot of parameter combinations when I try to solve the lunar lander problem. From those experiences, my guess is that the neural network learning rate and epsilon decay factor influence the learning speed of the agent, and mini-batch size and memory replay size influence the agent's maximum performance in limited training time.

To prove my thought, I did some experiments with the hyperparameters. While I test each single parameter, other parameters are the same as what I described in part VI:

A. Learning Rate (used in neural network loss function backpropagation)

I plotted the reward curves using different NN learning rates within 2000 episodes (Figure 7). The learning rate I experimented are 0.00015, 0.0003, 0.00045, 0.0006. If we focus on the 0 to 500 episode range, we can observe that the reward of 0.0006 learning rate(LR) improves the fastest, the

0.00015 learning rate(LR) curve improves the slowest, the performance improvement speed is 0.0006 LR > 0.00045 LR > 0.0003 LR > 0.00015 LR. The bigger the neural network learning rate is, the faster the performance improves.

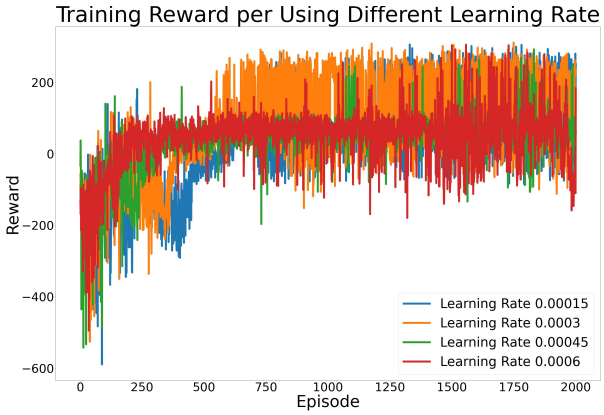

Figure 7: *Training reward using different NN learning rates*

But when the curves reach to 750 to 200 episode range, the 0.0006 LR curve seems to have less high reward compared to others. To observe the curves more clearly, I plotted the average reward(per 50 episodes) chart in Figure 8.

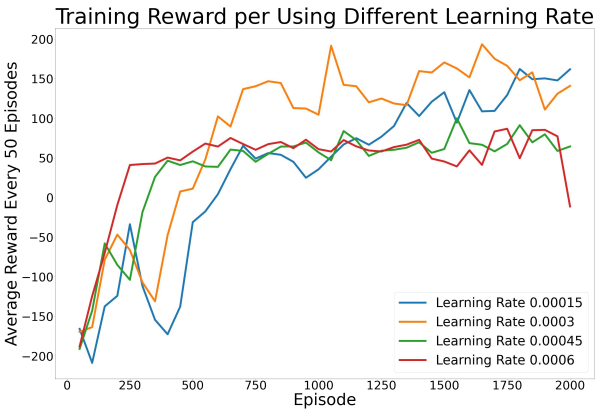

Figure 8: *Average training reward for 50 episodes using different NN learning rates*

From Figure 8 we can see the 0.0006 LR curve improves fast but it also hits the turning point fast and starts to improve slowly. The 0.00045 LR curve behaves similar to the 0.0006 LR, except it stops the fast growth later. The 0.0003 LR overall has the best performance, it has the highest average reward. Interestingly, the 0.00015 LR curve doesn't have the highest reward but its tendency is still growing. In episode 2000, the 0.00015 LR has the highest reward, if I trained more episodes, the 0.00015 may be the best parameter choice.

This experiment shows that the neural network learning rate can influence the agent's learning result. The higher the learning rate is the faster the performance starts to improve. However, the improvement starts fast and also ends fast. The low learning rate curve seems to be able to improve more with more training time.

However, we should consider both performance and time cost in problem-solving, the 0.0003 LR is the best parameter in limited 2000 episodes.

B. Epsilon decay factor (used in e-greedy Q-learning)

Epsilon decay factor is the 0.994 in my epsilon decay function $\varepsilon = max(0.01, n^{episode})$, (n = 0.994). When this factor is very close to 1, the epsilon will decay very slowly. So I plotted the reward curves using different epsilon decay factors within 2000 episodes (Figure 9). The factor samples are 0.984, 0.989, 0.994, 0.000.

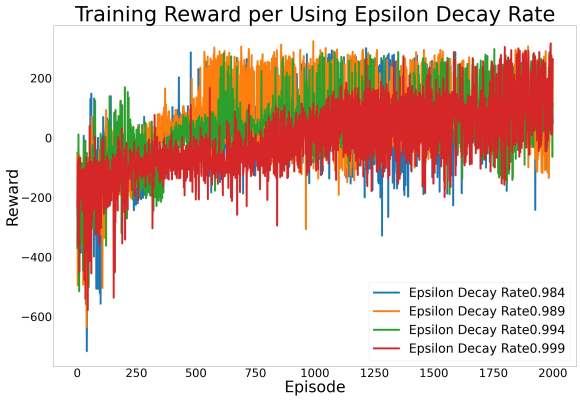

Figure 9: *Training reward using different epsilon decay rates*

Clearly, the performance of each curve improves in different speeds, which are n = 0.984 > n = 0.989> n = 0.994> n = 0.999. This can be explained as the slow decay epsilon let agent have more random actions instead of the best actions. The n = 0.999 curve improves slowly which is because it still has a big probability to do the random actions.

To observe the curves' tendencies better, I also plotted the average reward per 50 episodes (Figure 10). We can see that even though the curves behave differently at the first, they kind of tangle at the end, it's hard to pick the best and the worst within 2000 episodes. But the n = 0.989 is clearly the best parameter if the training time is limited to 650 episodes.

In the 2000th episode, the n = 0.999 curve has the highest reward, and n = 0.984 has the lowest reward. My explanation is that the slowly decaying epsilon can have more opportunities to explore, to learn from the random actions and the random situations, that's why the slow decay factor may have a higher reward with more training time.

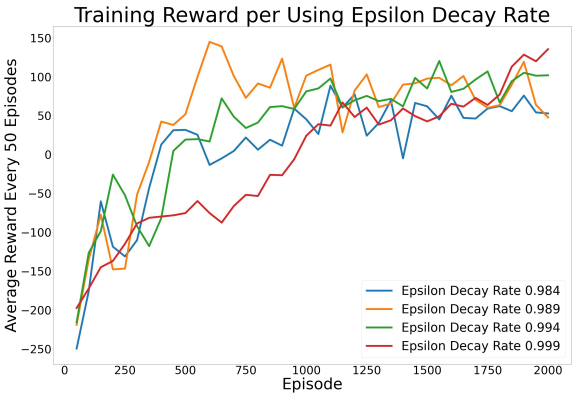

Figure 10: *Average training reward for 50 episodes using different epsilon decay rates*

C. *Minibatch Size (number of training case)*

The minibatch is a batch of data that gets sampled from the memory replay, which is used from training the deep learning network weight. For every C step, one minibatch will be selected. In this experiment, I used C = 4 for all agents. I experimented on 4 different minibatch sizes, 32, 64, 96, and 128. I plotted the reward curves using different minibatch sizes within 2000 episodes (Figure 11).

To observe the result better, I also plotted the average reward for every 50 consecutive episodes (Figure 12).

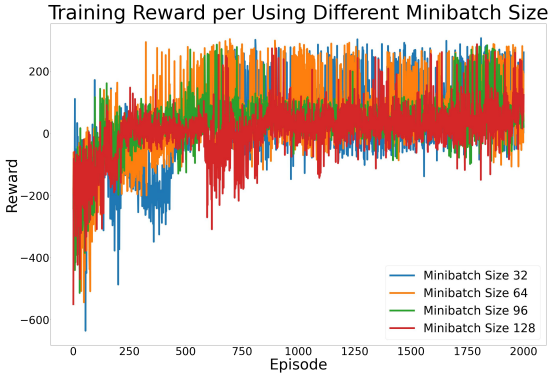

Figure 11: *Training reward using different minibatch sizes*

We can observe that when the episode is small, the 4 curves highly overlap together. As the more episode experienced, the 32-size curve had a big decrease in performance, which can be because the agent is trained to overfit an uncommon situation. With a larger minibatch size, there is less chance of overfitting. As the episode goes bigger, the rewards tend to be stable, while the size 128 has the lowest reward and size 64 has the highest reward. We can draw the conclusion that the minibatch size is not bigger the better. The size should be considered with the replay memory size and the neural network learning rate, learning too much from the current batch can lead to updating the weight too much.

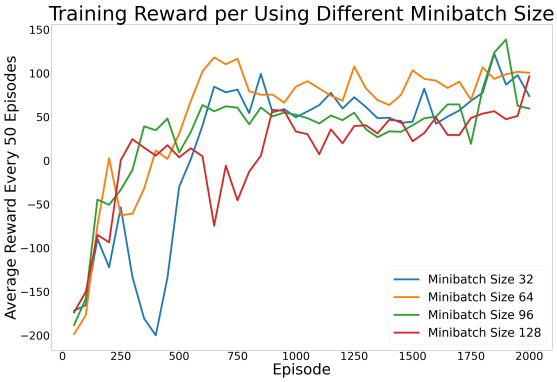

Figure 12: *Average training reward for 50 episodes using different minibatch sizes*

D. *Memory Replay Size (number of stored recent training samples, minibatch samples from the memory replay)*

The memory replay is the place that stores the most recent training example, where the minibatch is sampled from. A larger memory replay size can store more recent examples, which can influence the learning result. I plotted the reward curves using different memory replay sizes within 2000 episodes in Figure 13. The sizes I experimented with are 32, 64, 96, and 128. The average reward for 50 consecutive episodes is shown in Figure 14.

We can observe that the curves overlap at the beginning when the episode is low, which means the different memory play sizes won't impact the learning speed of the agent. But as the episode goes up, we can observe that memory size 50,000 has the lowest rewards, instead memory size 5,000 and 5,000,000 have the highest rewards. The memory replay size can impact a lot on the best performance that an agent can be trained to do.

Based on this experience, within 2000 episodes, the best memory replay size is 5,000. My explanation for this is that a small memory replay size can represent the most recent training examples, and the agent can be trained to handle the recent scenario which helps to improve the performance.

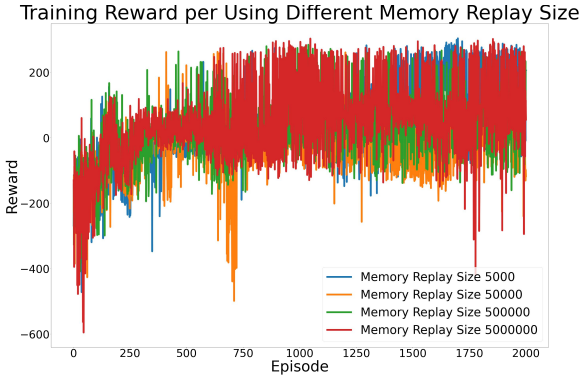

Figure 13: *Training reward using different memory reply sizes*

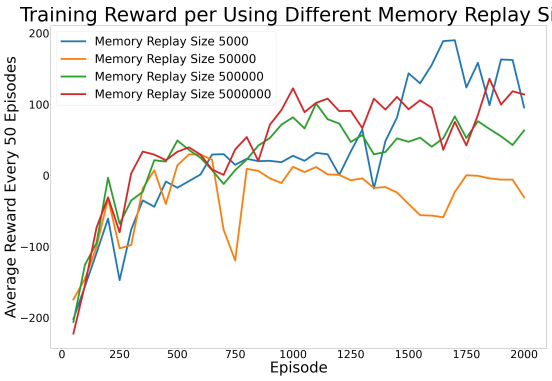

Figure 14: *Average training reward for 50 episodes using different memory replay sizes*

REFERENCES

[1] Sutton, R. S., Barto, A. (2018). Reinforcement learning: An introduction.. Cambridge, MA: The MIT Press.

[2] Sutton, R. S., Barto, A. (2018). Reinforcement learning: An introduction.. Cambridge, MA: The MIT Press.

[3] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015). https://doi.org/10.1038/nature14236

[4] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015). https://doi.org/10.1038/nature14236