Solving the Lunar Lander with Deep Reinforcement Learning

Tianda Fu
Department of Electrical and Computer Engineering
Queen's University
Kingston, ON, Canada
21tf5@queensu.ca

Juntao Lin
Department of Electrical and Computer Engineering
Queen's University
Kingston, ON, Canada
21jl80@queensu.ca

***Abstract*—The Deep Q-Network proposed by [illegible] became a benchmark and building point for deep reinforcement learning research. In this project, reinforcement learning methods are proposed to solve the lunar lander problem which is a 2-dimension box environment developed by OpenAI Gym. We implement a deep Q network as a baseline model followed by comparing the performance of the DQN model with Double DQN and Dueling DQN models. The simulation shows no significant difference between these models in their final score and converging time. We do observe that dueling DQN has a more stable performance compared with the baseline DQN and Double DQN.**

Keywords—Reinforcement Learning, Deep Learning, OpenAI Gym, DQN, Double DQN, Dueling DQN

I. INTRODUCTION

Model-free reinforcement learning has been applied to many challenging problems, especially when the environment is complex and hard to be modelled. Neural networks are good candidates for function approximation. Adapting deep learning techniques to the field of reinforcement learning introduces more alternatives while designing distinct policies and value functions [2]. With the proper feature engineering, training a neural network policy allows reinforcement learning agents to solve complex environments with a minimum set of state representations. Training robots and autonomous vehicles in the real-world environment can be very expensive. In the contrast, training a reinforcement learning agent in a simulated environment would be more cost-effective during the preliminary stage.

Q-Learning is a popular form of reinforcement learning that iteratively updates the Q-Table to improve the performance of the learning agent. One of the problems with Q-Learning is that the algorithm often overestimates the Q-Values. The overestimation becomes a problem when the overestimated are not uniformly distributed [3]. In such a case, the Q-Learning will pick the action with the highest value, thus overestimation accumulates quickly through iterations. In this paper, we explore different deep reinforcement learning models for a continuous control task. The performances of the models are validated through the LunarLander-v2 environment developed by OpenAI Gym [4]. The impact of overestimation will be determined during simulations. Our goal is to develop a single agent that uses neural networks to learn and solve the LunarLander-v2 environment.

II. RELATED WORK

A. DQN

Chen et al. proposed a Deep Q-Network (DQN) with experience optimization to solve Atari's Space Invaders environment. They concluded that DQN is sensitive to parameter tuning. Running DQN in different environments often requires a fine-tuning process on the parameters [5].

OwaisAli Chishti et al. designed an architecture that includes a CNN model and a DQN model. The CNN and DQN models are trained separately. The objective of the CNN model is to identify and locate the traffic signs in the source images. They drove the car around the local environment to collect more than 3000 data points to train the CNN model and achieved 73% validation accuracy. After detecting the traffic signs with the CNN model, the processed images are passed to the DQN model to train the reinforcement learning agent [6].

B. Double DQN

Q-Learning has a moderate performance while the environment is stochastic since the agent often faces the overestimation problem [7]. In 2010, Hasselt developed the Double Q-Learning algorithm that uses a double estimator approach to update the Q values for the next state [8]. In 2016, Hasselt et al. revaluated the overestimation problem of Q-Learning. They concluded that in practice the overestimation problem would become more severe. To address this problem, the Double DQN architecture was proposed and proved to be more reliable than the original DQN model [3].

Pan et al. published an actor learning-based multisource transfer double DQN. The authors combined transfer learning techniques with deep reinforcement learning to build a more generalized Double DQN model. The proposed architecture outperformed the baseline DQN and double DQN over multiple Atari games [9].

C. Dueling DQN

Wang et al. proposed the Dueling DQN model which contains two estimators to approximate the state value function and state-dependent action advantage function. The proposed model decouples the state value and action value, evaluated through all 57 Atari games. The result outperformed other RL

models in the Atari 2600 domain at the time this paper was published [10].

III. METHODS

Reinforcement learning algorithms can be divided into three categories: value-based, policy-based and actor-critic. The DQN represents the value-based algorithms that use one value function without a policy network.

According to the concept of Q-Learning [11], every state-action pair (s, a) corresponds to a value function Q(s, a). In theory, the value of any state-action pair can be determined through the lookup table (LUT) associated with the Q-Learning. However, when there are too many states or actions, the process of going through the LUT will be very inefficient, and the size of the memory might become a limitation. Therefore, a function approximation can be used to estimate the value function:

$$Q^{s,a,w} \approx Q_{\pi}(s,a) \quad (1)$$

In this way, the value function can also be estimated for states and actions that never seen by the model. In the DQN model, a deep neural network is used as the approximation function. If the environment is relatively simple, it is possible to use a linear function to approximate the Q-value.

A. DQN

Reinforcement learning is an iterative process. Each iteration must solve two problems: giving a policy evaluation and updating the policy according to the value function.

As mentioned above, DQN uses a neural network to approximate the value function. The input of the neural network is the state s, and the output is Q(s, a), $\forall a \in A$ (action space). After calculating the value function through the neural network, DQN uses the $\epsilon-$greedy [12] strategy to output the action. The connection between the value function network and the $\epsilon-$greedy strategy is as follows:

1) The environment will give an observation.

2) The agent obtains all Q(s, a) about the observation according to the value function network.

3) Using $\epsilon-$greedy to select an action and make a decision.

4) The environment will give a reward and the next observation after receiving the current action.

5) Updating the parameters of the value function network according to the reward, and then go to the next step.

Step 1 to 5 will be repeated until an effective value function network is trained for the DQN model.

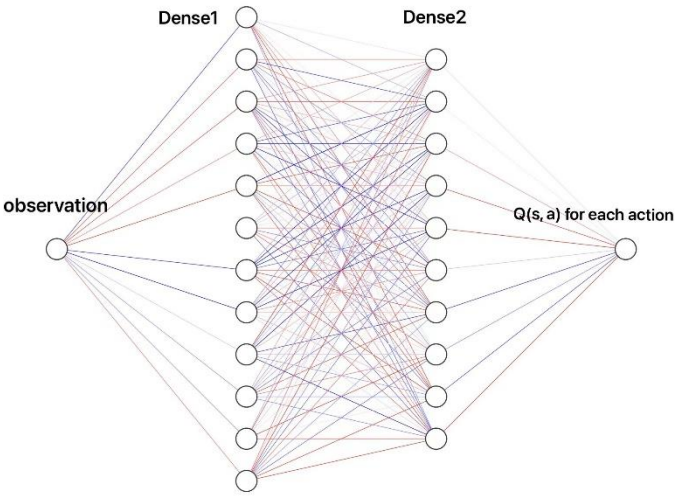

Fig. 1. The structure of the proposed value function network

To update the parameters in the neural network shown in Fig. 1, the loss function is defined as:

$$L(\theta) = E[(R + \gamma \cdot max_{a'} Q(s', a'; \theta^{-}) - Q(s, a; \theta))^2] \quad (2)$$

It is a residual model, like the least-squares method, the square of the difference between the actual value and the predicted value. The predicted value is Q(s, a; ω), which is the output of the neural network.

DQN is a model-free reinforcement learning algorithm that requires sampling to get the actual value rather than getting the value by using Bellman equation, since the model of the environment is unknown. Apart from that, DQN would update its value function only depending on the next step which is very similar to SARSA.

Both Q-Learning and DQN are off-policy reinforcement learning. A main feature of original DQN is that both the target value and the predicting value would be updated for each iteration, which means they are used the same network. One problem brought by this is that every time the neural network updated, the targets used to update the neural network will change accordingly, which will easily lead to the non-convergence of parameters. While in supervised learning, labels are stabled and will not change with parameter updates.

Therefore, DQN introduces a target Q network [13] on the basis of the original Q network, that is, the network used to calculate the target. It has the same structure as the Q network, and the initial weights are also the same, but the Q network is updated every iteration, and the target Q network is updated every once in a while. The target of DQN is:

$$Q_{target} = R_{t+1} + \gamma \cdot max_{a'} Q(S_{t+1}, a'; \theta^{-}) \quad (3)$$

Denote by $\omega-$ that it is slower to update than the weights ω of the Q network.

Compared with Q-Learning, DQN has the following improvements: approximating the value function through a convolutional neural network, using the target Q network to update the target, and including the experience replay feature.

Unlike supervised learning where the training data are mostly independent, in the proposed DQN model, the training data are observations from the environment that are sequential, from one time step to the next. Updating the parameters of the neural network with such sequential data might be problematic. Therefore, an experience playback function is added to the DQN model to address this problem. The DQN model uses a memory block to store the experience data. Every time the DQN model performs an update, data will be extracted from the memory block for updating the weights. By doing that, the network can break the association between the data [14].

B. *Double DQN*

Double DQN improved based on the DQN model. The model structures of double DQN and DQN share many features in common. The only difference is how to calculate the target function. The target function of DQN and double DQN are described on the equations 4 and 5 below.

$$Y_t^{DQN} \equiv R_{t+1} + \gamma \cdot max_a Q(S_{t+1}, a; \theta_t^-) \quad (4)$$

$$Y_t^{DDQN} \equiv R_{t+1} + \gamma \cdot Q(S_{t+1}, argmax_a Q(S_{t+1}, a; \theta_t), \theta_t^-) \quad (5)$$

The difference between these two target functions is that the optimal action selection of double DQN is based on the parameter θ of the Q network currently being updated, whereas the optimal action selection in DQN is based on the parameter θ in target-Q network.

The reason for this change in updating the double DQN is that traditional DQN usually has an overestimation problem [15] in predicting Q values.

Since double DQN is based on the parameters of the current Q network each time, which is not based on the parameters of target-Q like DQN. So, when calculating the target value, it will be a little bit smaller than the original. Because the target value is calculated through the target-Q network, in DQN, the action with the largest Q value was originally selected according to the target-Q parameter, but now the Q value calculated after replacing the selection with DDQN must be less than or equal to the original Q value. This reduces the overestimation to a certain extent, making the Q value closer to the true value.

C. *Dueling DQN*

Dueling DQN is also an improvement based on the DQN model. The only difference between the two models happens in the last layer of the DQN's neural network. In the original DQN, the last layer is a fully connected layer. After this layer, n Q values (n represents the number of selectable actions) are the output. However, Dueling DQN does not directly train to obtain those Q values [16]. It obtains two indirect variables V (state value) and A (action advantage) through training and then the summation of those two variables expresses the final Q value.

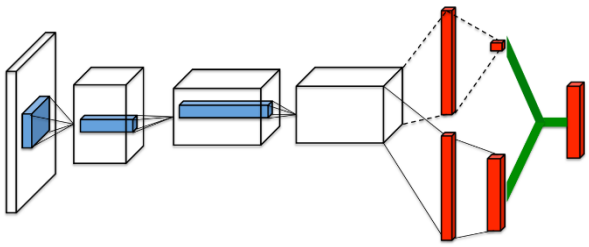

Fig. 2. The structure of Dueling DQN [10]

A common structure of Dueling DQN is shown in Fig. 2 above. In the proposed Dueling DQN model, the model uses fully-connected (FC) layers instead of convolution layers. we can still separate the network into three parts:

1) Part 1: Like ordinary DQN, this part is used to process and learn data.

2) Part 2: Calculating the state value which is the average value estimated by the network.

3) Part 3: Normalizing the action value to keep the mean of the action values at 0.

The equation of state value V and the action advantage A are calculated as shown in equations 6 and 7.

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi(s)}[Q^{\pi}(s,a)] \quad (6)$$

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \quad (7)$$

V represents the average expectation of the Q value in the current state s (considering all optional actions). A represents how much the Q value exceeds the expected value when choosing action a. Adding the state value V and the action advantage A together leads to the actual Q(s, a). The state value V is the current estimation of the state. A is the adjustment for the current state value. Therefore, the model is designed in this way to allow the neural network to have a basic judgment on the given state s, and then make corrections according to different actions on this basis.

When DQN increases the estimated state value in a given state, there is no compensation for this increase and might result in overestimation during iterations of training. Such an increase in Dueling DQN will not be a problem. When the network updates, the network will preferentially update the state value so that the sum of the action values must be 0. Rather than updating the Q value for a single action, the model adjusts the Q values of all available actions in the current state. This way, the model can update more values in fewer iterations.

IV. TUNING DQN MODEL

This section describes the tuning process of the proposed model. The impact of each tunable parameter will be discussed later in the section. The DQN model is trained as a stable baseline for the other proposed models. All the proposed models share a similar set of parameters. Table 1 summarizes the hyperparameters used in the models.

Name	Symbol	Variable Name	Description
Gamma	γ	gamma	The discount factor for future reward.
Epsilon	ε	Epsilon	Epsilon value used in ε -greedy method that decreases after each episode.
Final Epsilon Value		Epsilon_end	Final value for epsilon after episodes of training.
Epsilon Decay Rate		epsilon_dec	Parameter decides how fast does the epsilon value decrease after each episode.
Memory Size		mem_size	Memory size of the replay buffer. Indicating the maximum number of states will be stored by the agent.
Batch Size		batch_size	The number of examples saved in the replay buffer, before training the agent.
Learning Rate	α	lr	Learning rate used in the Adam optimizer.
Episodes		n_games	The total number of episodes in which the agent will be trained.

Table 1: Hyperparameters in the proposed models.

The training process starts with the DQN model. The initial DQN model uses a neural network that has 3 FC layers; each layer includes 128 neurons.

The simulation result using the initial set of parameters is summarized in Table 2. The agent did not approach a positive score in the LunarLander-v2 environment. We suspect the epsilon decayed too fast, so the agent could not explore the environment enough.

Parameters		
gamma=0.99	Epsilon=2.0	mem_size=1e5
batch_size=32	epsilon_end=0.01	lr = 0.001
n_games=300	epsilon_dec=1e-3	
FC (128) -> FC (128) -> FC (256) -> Output		
Result		
Convergence: No	Final Score: -800	

Table 2: Initial parameters used in the DQN model.

As the epsilon decay rate lowered to 1e-4, the agent could explore more states and achieved a better score. The result shows that in the first 150 games, the agent was mostly exploring, and the model was approaching convergence. After 160 games, the agent would exploit 99% of the time. The agent's performance dropped suddenly after training for 200 games.

Parameters		
gamma=0.99	Epsilon=2.0	mem_size=1e5
batch_size=32	epsilon_end=0.01	lr = 0.001
n_games=300	epsilon_dec=1e-4	
FC (128) -> FC (128) -> FC (256) -> Output		
Result		
Convergence: No	Final Score: -480	

Table 3: Simulation with a slower decay rate for epsilon.

Since lowering the epsilon decay improved the performance, the final epsilon value increased from 0.01 to 0.1 in the next simulation. In the first 160 games, the agent would mostly explore. After 160 games, the agent would explore at 10% chance and exploit at 90% chance. Fig. 2 indicates the agent's performance collapsed shortly after finishing the exploration stage. This collapse was observed through multiple simulations, even with different sets of parameters.

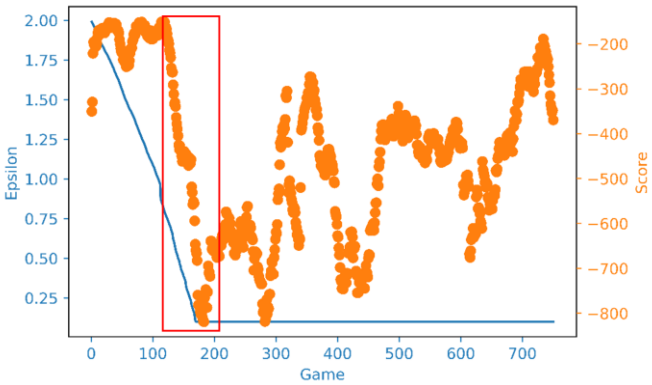

Fig. 3. Training curve with 0.1 final epsilon, performance collapsed around game 200

To address the sudden drop in agent's performance, the memory increased from 1e5 to 2e5, and the final epsilon value increased from 0.1 to 0.3. We did two simulations with the same sets of parameters to determine whether the model has a consistent performance. The simulation results are shown in Fig. 4 and Fig. 5, the regions where the collapses happened are highlighted. Although changing the epsilon value in the epsilon-greedy strategy has a positive impact on the model's performance, the model does not have a consistent behavior.

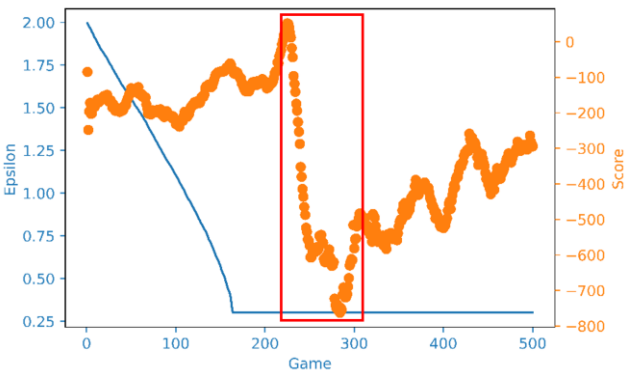

Fig. 4. Training curve with increased final epsilon value and memory size

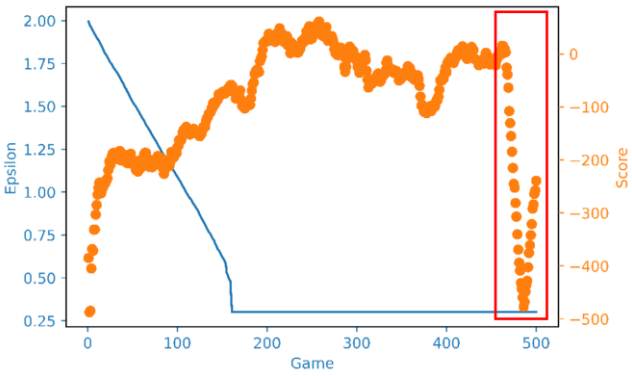

Fig. 5. Repeating the same simulation to check for consistency

Adjusting the epsilon and its decay rate did not solve the collapse problem in the agent's performance, so we take a closer look at gradients propagated in the multilayer perceptron (MLP) network. When the agent picks up a random move that caused a sudden change in the gradient, the Adam optimizer used in the MLP network might not be able to handle it. A significant change in gradient is very likely to result in an exploding gradient problem. In such a case, the MLP network would have an undesired update that interrupts the previously trained weights. In consequence, the DQN model would have a large update as well that can overwhelm the agent.

To address the problem that the agent's experience is interfered by an unexpected update, we make the agent learn slower, therefore the MLP model could be more robust to sudden changes in gradient. To slow down the training process, the DQN model's performance is evaluated across smaller learning rates, discount factors, and batch sizes.

Parameters		
gamma=0.6	Epsilon=2.0	mem_size=2e5
batch_size=10	epsilon_end=0.3	lr = 5e-4
n_games=1000	epsilon_dec=8e-5	
FC (128) -> FC (128) -> FC (256) -> Output		
Result		
Convergence: No	Final Score: 0 ~ -50	

Table 4: Simulation with a smaller learning rate, discount factor, and batch size.

Adding batch normalization between the FC layers helps the deep neural network model to train in a faster and more stable manner [17]. While the batch size is set to a small number, adding batch normalization after the FC layers makes the DQN model diverge from the starting point. To fix the diverging problem, the batch size increased from 10 to 64. With batch normalization added to the MLP model, the agent is trained with 2000 games.

The result shows the DQN model did not converge to a final policy. The score achieved by the agent fluctuated between 100 and -100. The agent achieved one of the highest scores in around 500 episodes. However, training after 500 episodes were just fluctuations, the agent did not learn any useful experience.

Parameters		
gamma=0.6	Epsilon=2.0	mem_size=2e5
batch_size=64	epsilon_end=0.3	lr = 5e-4
n_games=2000	epsilon_dec=8e-5	
FC (128) -> FC (128) -> FC (256) -> Output		
Result		
Convergence: No	Final Score: 100 ~ -100	

Table 5: Simulation with batch normalization added to the MLP model.

Adding batch normalization to the MLP network solved the collapse problem observed in Fig. 4 and Fig. 5, but batch normalization also introduced a fluctuation problem in which the agent would not converge to a final policy. To make the DQN model more stable, gradient clipping will be tested next. Gradient clipping smooths the gradient used in the backpropagation and accelerates the process for the deep learning model to approach convergence [18]. Clipping the gradient by norm ensures every weight in the MLP network is clipped, therefore the norm of the weights will not exceed a specific value. The norm of the gradient clipping is set to 1 for the next simulation.

The result shows that gradient clipping did not stabilize the DQN model. In contrast, adding gradient clipping exaggerated the fluctuation problem. After 500 games of training, the agent's performance fluctuated between -150 and -300.

Parameters		
gamma=0.6	Epsilon=2.0	mem_size=2e5
batch_size=64	epsilon_end=0.3	lr = 5e-4
n_games=500	epsilon_dec=8e-5	
FC (128) -> FC (128) -> FC (256) -> Output		
Result		
Convergence: No	Final Score: -150 ~ -300	

Table 6: Simulation with batch normalization and gradient clipping added.

Gradient vanishing and exploding problems often occur as the depth of the neural network increases. Since batch normalization and gradient clipping did not help to stabilize the DQN model, we decide to reduce the number of FC layers. In the previous setup, the MLP network had 3 FC layers and 1 output layer. Now the number of FC layers is reduced to 2.

Within 400 games of training, the agent learned to score between 25 and -75. As the MLP network's depth decreases, the gradient changes in the backpropagation become less dramatic, but the training curve follows a zig-zag path that implies the model is still unstable. This result is captured in Fig. 6. Other than the number of FC layers, there should be other factors that dominate the DQN model's behaviors of being inconsistent.

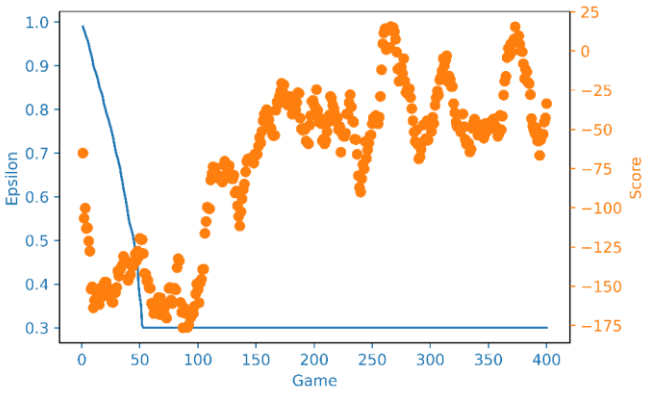

Fig. 6. Training curve with increased final epsilon value and memory size

Gradient was not the main issue causing the agent's inconsistent behavior, the following simulations are focused on the epsilon value used in the epsilon-greedy method. Having a high exploration rate helps agents to learn the environment faster, especially during the early training. The benefit of having a high exploration rate decreases as the agent becomes more experienced [19]. Previously the exploration rate was set to 30% to help the agent to explore more often. However, the final exploration rate must be lowered, so the agent can learn more consistent behaviors. In the next simulation, the final exploration rate lowered to 1%.

Decreasing the final exploration rate while having a small batch size makes the model diverge. To accommodate the lowered exploration rate, batch size increased from 10 to 100. Within 150 games of training, the agent converged to a final policy that scored -100.

Lowering the final exploration rate from 30% to 1% makes the DQN model more stable throughout the training session. The zig-zag pattern is no longer observed in the training curve. We believe lowering the final exploration rate helps to mitigate the inconsistent performance of the DQN model.

Parameters		
gamma=0.6	Epsilon=1.0	mem_size=2e5
batch_size=100	epsilon_end=0.01	lr = 5e-4
n_games=500	epsilon_dec=8e-4	
FC (128) -> FC (128) -> Output		

Result	
Convergence: Yes	Final Score: -100

Table 7: Simulation with reduced final exploration rate and increased batch size.

The gamma value has a very important role in training and updating the MLP network. Different gamma values are evaluated for the DQN model, the rest of the parameters remain the same as summarized in Table 7. Setting the gamma value to less than 0.9 makes the DQN model less stable, whereas setting the gamma to 1 causes the DQN model to diverge within the first 50 episodes. Increasing the gamma value from 0.99 to 0.999 makes the DQN model diverge as well. We believe this divergence is caused by overestimation. Increasing the gamma value from 0.9 to 0.99 helped the DQN model to converge to a much better policy. The agent was trained for 300 games and reached a final score of around 230.

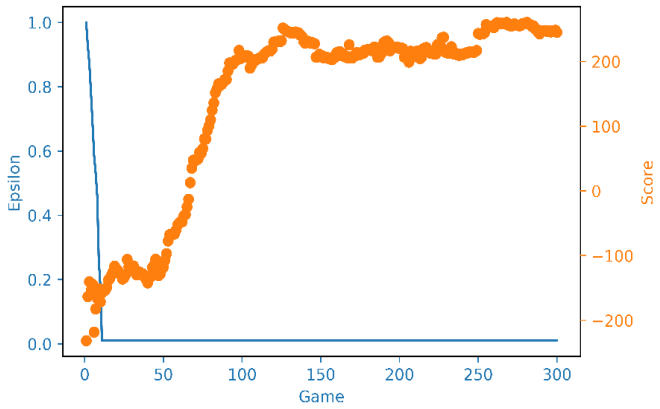

Fig. 7. Training curve with gamma set to 0.99

V. Experiment Result

In this section, the experiment results are presented for each of the proposed models.

A. DQN

The proposed DQN model uses the parameters summarized in Table 8. In addition, simulation results shown in Fig. 7 and Fig. 8 use the same set of parameters. The only difference between the two simulations is one setting for the Adam optimizer. Momentum is calculated in the Adam optimizer while performing the gradient descent. Momentum allows the optimizer to build inertia in the direction that the gradient descent approaches, therefore the optimizer can overcome the noises encountered while descending the gradient [20]. When the gradient changes frequently in the DQN model, having high inertia will have a negative impact on the optimizer's performance.

The momentum of the Adam optimizer is reduced by increasing the momentum decay from 0.99 to 0.5. In 800 games of training, the DQN model converged to a final policy that scored around 270.

Parameters		
gamma=0.99	Epsilon=1.0	mem_size=2e5
batch_size=100	epsilon_end=0.01	lr = 5e-4
n_games=900	epsilon_dec=8e-4	
FC (128) -> FC (128) -> Output		
Result		
Convergence: Yes	Final Score: 270	

Table 8: Simulation with reduced final exploration rate and increased batch size.

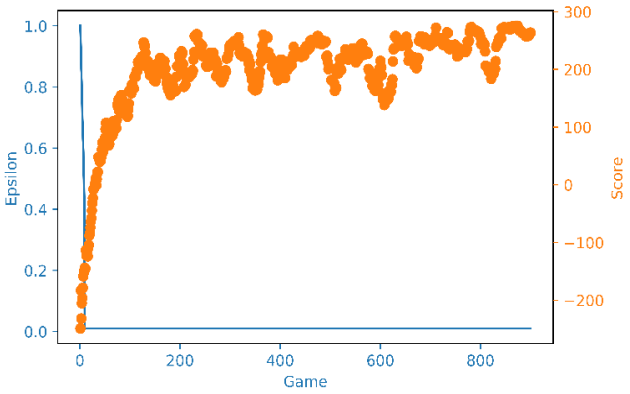

Fig. 8. Training curve for the final DQN model

B. *Double DQN*

The proposed Double DQN model uses the parameters summarized in Table 9. In the beginning, the gamma was set to 0.9 and the epsilon decay was 8e-5. In that scenario, the model can only converge to around -100. No matter how many steps the agent was trained, the reward could not be higher than 0. The agent might be trapped in a local minimum and cannot move out to achieve a higher score.

After testing different sets of parameters, a relatively good configuration was found in which the gamma has been changed from 0.9 to 0.99 and epsilon decay was changed from 8e-5 to 0.995. Meanwhile, the neural network used in this model was a two-layer fully-connected network with 256 neurons in each layer. To get the global maximum reward, Adam is the optimizer since it has a better performance at descending gradient.

The final result of the Double DQN model is shown in Fig. 9, the model converged in 200 games and achieved a final score of 260.

Parameters		
gamma=0.99	Epsilon=1.0	mem_size=2e6
batch_size=64	epsilon_end=0.01	lr = 5e-4
n_games=500	epsilon_dec=0.995	
FC (256) -> FC (256) -> output		
Result		
Convergence: Yes	Final Score: 260	

Table 9: The final parameters used in the Double DQN model

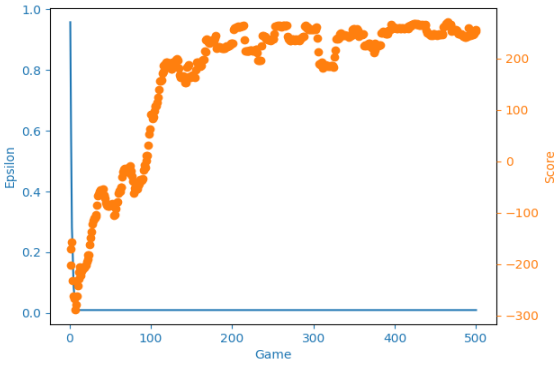

Fig. 9. Traing curve for final Double DQN model

C. *Dueling DQN*

The proposed Dueling DQN model uses the parameters summarized in Table 10. In Dueling DQN, there are two MLP networks, one for estimating the state value, and the other one for evaluating the action advantage. To simplify this model, the two MLP networks share the same structure.

The two MLP networks have two FC layers followed by an output layer, each FC layer has 128 neurons. The gamma value started at 0.9, but the reward always converged to -100. This behavior was also observed in DQN and Double DQN model, when the gamma values were set to 0.9.

During the tuning steps, the following gamma values were tested: 0.9; 0.99; 0.995 and 0.999. Multiple simulations were done to test different learning rate, batch size, epsilon decay rate, and the number of training episodes. The best result obtained is shown in Fig. 10 that the Dueling DQN model converged in around 350 steps and reached an average reward of 270.

Parameters		
gamma=0.99	Epsilon=1.0	mem_size=2e5
batch_size=64	epsilon_end=0.01	lr = 5e-4
n_games=500	epsilon_dec=1e-3	
MLP1: FC (256) -> FC (256) -> output MLP2: FC (256) -> FC (256) -> output		
Result		
Convergence: Yes	Final Score: 270	

Table 10: The final parameters used in the Dueling DQN

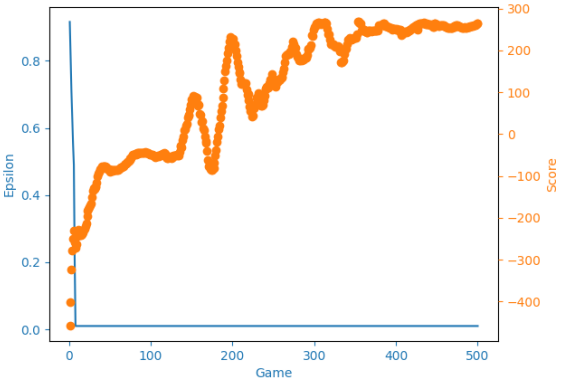

Fig. 10. Traing curve for the final Dueling DQN model

Model	Final Reward
DQN	~270
DDQN	260
Dual-DQN	270

Table 11: Comparison of different models

VI. Conclusion

This project proposes to use the DQN, Double DQN, and Dueling DQN model to solve the LunarLander-v2 environment. All three models converged into a final policy. Both the DQN and Dueling DQN score near 280, and the Double DQN model scores around 260. In the terms of final score and convergence time, the three models have similar performances. Once the models converged to an optimal policy, the Dueling DQN model consistently hits the score of 270, whereas the DQN and Double DQN models still have fluctuation on the training curves. The Double DQN and Dueling DQN mitigates the overestimate problem, which makes these models more stable than the baseline DQN architecture.

VII. Member Contributions

All members met regularly to discuss the project. A general breakdown of each member's contributions is in the table below.

Member	Contributions
Tianda Fu	- Implementing DDQN and Dueling DQN models with Python. - Comparing the result of the three models. - 50% of the report.
Bob Lin	- Implementing the DQN model with Python as the baseline. - Tuning the parameters for all three models. - 50% of the report.

Table 12: Contributions of each group member

VIII. References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness and M. G. Bellemare, " Human-level control through deep reinforcement learning," *Nature*, p. 518(7540):529–533, 2015.

[2] K. Arulkumaran, M. P. Deisenroth, M. Brundage and A. A. Bharath, "Deep Reinforcement Learning: A Brief Survey," *IEEE Signal Processing Magazine*, vol. 34, pp. 26-38, 2017.

[3] H. van Hasselt, A. Guez and D. Silver, "Deep reinforcement learning with double Q-learning," *Proc. Association for the Advancement of Artificial Intelligence*, pp. 2094-2100, 2016.

[4] G. Brockman, V. Cheung, L. Pettersson, J. Schneide, J. Schulman and J. Tang, et al., "Openai gym," 2016.

[5] Y. Chen and E. Kulla, "A Deep Q-Network with Experience Optimization (DQN-EO) for Atari's Space Invaders," *Advances in Intelligent Systems and Computing*, vol. 927, 2019.

[6] S. OwaisAli Chishti, S. Riaz, M. BilalZaib and M. Nauman, "Self-Driving Cars Using CNN and Q-Learning," in *IEEE 21st International Multi-Topic Conference (INMIC)*, 2018.

[7] C. Schulze and M. Schulze, "ViZDoom: DRQN with prioritized experience replay double-Q learning & snapshot ensembling," arXiv:1801.01000, 2018, [online] Available: https://arxiv.org/abs/1801.01000.

[8] H. Hasselt, "Double Q-learning," in *Advances in neural information processing systems 23*, 2010.

[9] J. Pan, X. Wang, Y. Cheng and Q. Yu, "Multisource Transfer Double DQN Based on Actor Learning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, pp. 2227-2238, 2018.

[10] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot and N. Freitas, "Dueling Network Architectures for Deep Reinforcement Learning," in *The 33rd International Conference on Machine Learning*, 2016.

[11] C. J. C. H. Watkins and P. Dayan , "Q-learning," *Machine Learning*, vol. 8, pp. 279-292, 1992.

[12] M. Tokic, "Adaptive ε-Greedy Exploration in Reinforcement Learning Based on Value Differences," in *KI 2010: Advances in Artificial Intelligence, 33rd Annual German Conference on AI*, Karlsruhe, Germany, September 21-24, 2010.

[13] J. Fan, Z. Wang, Y. Xie and Z. Yang, "A Theoretical Analysis of Deep Q-Learning," *Proceedings of the 2nd Conference on Learning for Dynamics and Control, PMLR 120:486-489*, 2020.

[14] L. Lv, S. Zhang, D. Ding and Y. Wang, "Path Planning via an Improved DQN-Based Learning Policy," in *IEEE Access, vol. 7, pp. 67319-67330*, 2019.

[15] W. Zhang, J. Gai, Z. Zhang, L. Tang, Q. Liao and Y. Ding, "Double-DQN based path smoothing and tracking control method for robotic vehicle navigation," *Computers and Electronics in Agriculture,*, vol. 166, p. 104985, 2019.

[16] Y. Zhao, Z. Wang, K. Yin, R. Zhang, Z. Huang and P. Wang, "Dynamic Reward-Based Dueling Deep Dyna-Q: Robust Policy Learning in Noisy Environments," *AAAI*, vol. 34, no. 05: AAAI-20 Technical Tracks 5, pp. 9676-9684, Apr. 2020.

[17] S. Santurkar, D. Tsipras, A. Ilyas and A. Madry, "How Does Batch Normalization Help Optimization?," *Advances in Neural Information Processing Systems 31*, pp. 2483-2493, 2018.

[18] J. Zhang, T. He, S. Sra and A. Jadbabaie, "Why gradient clipping accelerates training: A theoretical justification for adaptivity," *Proc. Int. Conf. Learn. Represent. (ICLR)*, pp. 1-21, May 2019.

[19] A. Nair, B. McGrew, M. Andrychowicz, W. Zaremba and P. Abbeel, "Overcoming Exploration in Reinforcement Learning with Demonstrations," in *IEEE*

International Conference on Robotics and Automation (ICRA), 2018.

[20] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *3rd International Conference for Learning Representations*, 2015.

[21] B. Guo, X. Zhang, Q. Sheng and H. Yang, "Dueling Deep-Q-Network Based Delay-Aware Cache Update Policy for Mobile Users in Fog Radio Access Networks," *IEEE Access*, vol. 8, pp. 7131-7141, 2020.