Reinforcement Learning Practical Final Project: Lunar Lander

Efe Görkem Şirin and Nihat Aksu

University of Groningen, University of Groningen PO Box 72 9700 AB Groningen The Netherlands

Abstract. This paper investigates the application of reinforcement learning (RL) techniques, specifically Linear Q-Learning and Deep Q-Networks (DQN), to address the Lunar Lander problem. The objective is to autonomously land a rocket on a predetermined pad, posing challenges in discrete actions and precise control. Despite discretized observations, Linear Q-Learning struggles with the Lunar Lander's complex dynamics. In contrast, DQN, leveraging deep neural networks, successfully learns non-linear approximations of the Q-value function. Experiments encompass various environmental conditions, including wind and turbulence, showcasing DQN's adaptability to increased complexity. Results highlight the scalability of solution approaches, with a deeper DQN architecture capturing more nuanced state-action relationships. Unexpectedly, the addition of wind and turbulence does not necessarily elevate the environment's complexity; a more generalized DQN model trained in a varied environment performs better. This study underscores the importance of selecting appropriate RL algorithms based on task intricacies and environmental characteristics, providing insights into the efficacy of RL in solving complex problems.

Keywords: Keywords: Reinforcement Learning, Lunar Lander, Linear Q-Learning, Deep Q-Networks (DQN), Autonomous Landing, Complex Dynamics, Neural Networks.

1 Introduction

Becoming an astronaut is a common childhood dream among the team, however, after realizing us becoming an astronaut is statistically improbable we decided to give our best efforts to take away astronauts' jobs using AI. Therefore we choose the lunar landing problem as a starting step.

Particularly in dynamic and unpredictable situations, Reinforcement Learning (RL) has become a potent paradigm for resolving complicated decision-making problems (Gadgil et al., 2020). To create an intelligent agent that can land a rocket on a predetermined pad on its own, we use RL approaches to take on the challenge of optimizing the Lunar Lander trajectory.

The Lunar Lander environment is grounded in Box2D physics, representing a classic problem in RL. The task involves guiding a rocket to a safe landing, where the landing pad is located at coordinates (0,0). The challenge lies in navigating discrete actions, including firing left, main, or right orientation engines, or taking no action. This discrete action space aligns with Pontryagin's maximum principle, suggesting that the engine should be either at full throttle or completely off.

There are two variations of the environment: continuous and discrete. The agent must learn exact engine control to accomplish a successful landing. By enabling more precise control over the throttle of the main engine and the lateral boosters, the continuous version adds even more complexity but we will be working with a discrete version of the environment in this paper.

In our project, we implemented linear Q-learning and Deep Q-Networks (DQN) independently to assess their performance in solving the Lunar Lander problem. Linear Q-learning, a classic reinforcement learning algorithm, serves as a benchmark for comparison, while DQN, a more advanced approach that incorporates neural networks, allows us to explore the benefits of deep learning in this complex task.

In this paper, we seek to find an answer to whether Linear Q Learning and DQN will be able to solve the task of landing the lunar lander. We predict that they both will be able to solve the task but predict that DQN will achieve higher scores due to their ability to capture more complex environments.

2 Methods

2.1 Environment

The essence of the problem is to navigate the lander to the landing pad using the thrusters on the lander. We use the environment provided by OpenAI Gymnasium, "LunarLander-v2"(Gymnasium Documentation, n.d.).

The lunar lander environment is deterministic. The interactions in the environment are based on the physics simulation, therefore, they are consistent given the same initial conditions and actions. The initial position of the lander is constant, which is the center concerning the x-axis and the top concerning the y-axis. The lander starts with a random initial force applied to its center of mass. The wind is generated using the function:

$$\tanh(\sin(2k(t+C)) + \sin(\pi k(t+C)))$$

where k is set to 0.01, and C is sampled randomly between -9999 and 9999. Turbulence_power dictates the maximum magnitude of rotational wind applied to the craft. We used a wind value of 15 and a turbulence value of 1.5.

The scoring system for the lander's performance varies based on its proximity to the landing pad, its speed, its angle of tilt, and its leg contact with the ground. Specifically, the score increases or decreases depending on whether the lander is closer or further from the landing pad, and it similarly adjusts based on the lander's speed, with slower movements increasing the score. The score is negatively impacted the more the lander tilts away from a horizontal orientation. For each leg that makes contact with the ground, the score is increased by 10 points. However, using the side engines and the main engine detracts from the score, with deductions of 0.03 and 0.3 points for each frame they are firing, respectively. The episode's outcome further influences the score, adding 100 points for a safe landing or subtracting 100 points for a crash. For an episode to be deemed a success, the score must reach at least 200 points. The reward is dependent only on the current state and action therefore it is an episodic problem. We limit each attempt to 1000 steps to make it a finite horizon problem.

We have a continuous state and observation space. The state is an 8 dimensional vector of the coordinates of the lander in x and y dimensions, its linear velocities in x and y dimensions, its angle, its angular velocity, and two boolean values that represent whether each leg is in contact with the ground or not. The action space however is discrete. The agent has the option to do nothing, fire the left orientation engine, fire the main engine, and fire the right orientation engine; they are numbered 0, 1, 2, and 3 respectively. The environment is fully observable because all necessary state information is known at every frame.

2.2 Q-Learning

Q-learning is a model-free reinforcement learning algorithm used to inform an agent on how to act optimally in a given environment by learning the value of actions in different states. It is not dependent on an environment model and can deal with issues involving random transitions and rewards without requiring modifications. The fundamental principle of Q-learning is the estimation of the optimum action-value function, which provides the predicted utility of choosing the best course of action in a given condition and then acting accordingly.

The algorithm maintains a table (Q-table) where each entry $Q(s, a)$ represents the estimated value of taking action a in state s. The Q-values are updated iteratively using the Bellman equation:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

where α is the learning rate, r is the reward received after taking action a in state s, s' is the subsequent state after action a is taken, γ is the discount factor that balances the immediate and future rewards, and $\max_{a'} Q(s', a')$ represents the estimated maximum future reward from the next state s'.

Because Q-learning is off-policy, it gains knowledge about the optimum policy's worth without regard to the agent's behaviors. This promotes the exploration of the state-action space by enabling the agent to learn from exploratory acts that might not necessarily be included in the present policy. The Q-table converges to the optimal action-value function over time as the agent updates the Q-values and investigates the surroundings. This enables the agent to choose the action with the greatest Q-value in each state to choose the best course of action.

Because of its ease of use and efficiency in discrete, finite Markov Decision Processes (MDPs), this algorithm is a cornerstone approach in the field of reinforcement learning, with applications ranging from robotic control to gameplay.

2.3 Linear Q-Learning

Instead of keeping a table for every conceivable state-action pair, linear Q-learning is a variation of the Q-learning technique that uses a linear combination of characteristics generated from the state-action pairings to approximate the action-value function, $Q(s, a)$. This method works especially well in settings where it is not practicable to maintain a discrete Q-table, such as those with a vast or continuous state space. The Q-value for a state-action pair in linear Q-learning is expressed as follows:

$$Q(s, a) = \theta^T \phi(s, a)$$

where θ is the weight vector and $\phi(s, a)$ is the feature vector that is obtained from the state s and action a. The selection of characteristics is crucial and has a big influence on how effective and efficient the algorithm is.

Iteratively updating the weights θ in response to variations between estimated Q-values and observed rewards is the learning process. In a linear setting, the update rule is:

$$\theta \leftarrow \theta + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]\phi(s, a)$$

where α is the learning rate, r is the immediate reward, γ is the discount factor, s' is the subsequent state, and $\max_{a'} Q(s', a')$ represents the maximum estimated reward for the next state s' over all possible actions a'. By minimizing the prediction error, this rule modifies the weights θ, gradually improving the approximation of the Q-function.

2.4 Deep Q-Networks (DQN)

By combining deep neural networks with Q-learning, Deep Q-Networks (DQN) offer a breakthrough in reinforcement learning, allowing agents to learn optimum policies in high-dimensional state spaces straight from unprocessed sensory inputs. DQN, developed by DeepMind, uses Convolutional Neural Networks (CNNs) to overcome the drawbacks of conventional Q-learning and linear function approximation techniques (Mnih et al., 2013). This allows for the automatic extraction and learning of features from complicated inputs. The main innovation of DQN is its ability to use a deep neural network that accepts the state as input and outputs Q-values for all possible actions to approximate the action-value function:

$$Q(s, a)$$

where s is a state and a is an action.

DQN uses several crucial strategies to enhance and stabilize the learning process. It first employs experience replay, which stores experiences in a replay buffer at each time step (referred to as transitions (s, a, r, s'). The network is updated using random mini-batches from this buffer, which smoothens variations in the data distribution and lessens correlations in the observation sequence. Second, DQN generates the Q-value targets using a different target network. To reduce the possibility of feedback loops between the Q-values and the target values, which could result in unstable learning dynamics, the weights of this network are periodically updated with the weights of the learning network.

When a DQN is trained, its goal is to minimize the difference in loss between the target and predicted Q-values. The target Q-values are calculated by taking the action's observed reward and discounting the highest future reward that the target network predicts. The DQN can learn policies efficiently from high-dimensional inputs thanks to this learning process, which makes it suitable for a variety of sophisticated tasks, such as robotic control and playing Atari games.

Because of their unique benefits in managing the discrete action spaces and continuous state spaces characteristic of this task, Linear Q-learning and Deep Q-Networks offer attractive solutions for the Lunar Lander. Thanks to its ease of use and effectiveness, linear Q-learning works especially well in settings where the state space may be discretized. This method makes an excellent choice for initial experiments, environments with limited computational resources, or where interpretability of the learned policy is important because it can be implemented simply and quickly converge to a reasonable policy for safely landing the spacecraft.

However, DQN does not require human feature engineering since it uses deep neural networks to automatically extract and learn features from the continuous state space of the Lunar Lander environment, even with its complicated dynamics. The ability to navigate and land on the lunar surface while under the influence of gravity, while maintaining precise control over movement, is a critical skill for the Lunar Lander challenge. Stable and effective learning even in the high-dimensional state spaces is made possible by the employment of experience replay and a target network in DQN, which separately handle the problems of correlation in sequential input and non-stationarity of targets.

Through processes such as epsilon-greedy strategies, both algorithms are skilled at managing the exploration-exploitation trade-off, a crucial part of learning in reinforcement learning. This guarantees that the agent will be able to gradually identify and strengthen the best landing tactics. The decision between DQN and Linear Q-learning ultimately comes down to the environment's complexity, the necessity for feature abstraction, and the particular need for computing efficiency. Linear Q-learning provides a less complex, but still powerful, alternative for environments where the dynamics can be captured with less complexity. DQN's ability to handle high-dimensional inputs and extract features makes it an especially powerful choice for the Lunar Lander environment, where mastering the dynamics is crucial to success.

2.5 Loss Function

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Since Mean Squared Error (MSE) is a good way to quantify the difference between the target and predicted Q-values, it is frequently used as a loss function in Q-learning, including both Deep Q-Networks (DQN) and Linear Q-learning. A clear, quantitative indicator of how well the agent's Q-value predictions match the observed rewards plus the discounted future rewards is provided by the Mean Squared Error (MSE), which is calculated as the average of the squares of the discrepancies between the predicted and actual values. This is important for reinforcement learning tasks, where the objective is to learn an optimal policy by minimizing the error in the value function estimation.

Because MSE is quadratic, it penalizes larger mistakes more than smaller ones. This feature is useful in the Lunar Lander setting, because the agent may crash or miss the landing pad if major deviations from ideal behavior occur. These events might have considerable negative consequences. MSE pushes the learning algorithm to concentrate on decreasing these significant errors by punishing these huge errors more severely, which results in more robust learning.

Fundamentally, Q-learning may be understood as an attempt to regress the predicted rewards for action-state pairings in a regression issue. Since MSE is a common regression loss function, it is a good fit for estimating Q-values, or predicted future rewards.

2.6 Exploration Strategy

$$\text{Choose action } a \begin{cases} \text{Explore: select with probability } \epsilon \\ \text{Exploit: select the best-known with probability } 1-\epsilon \end{cases}$$

The epsilon-greedy exploration strategy is a straightforward but efficient way to find a balance between exploration and exploitation. It operates by arbitrarily choosing the most well-known action with probability 1- ϵ, a little positive number, and actions with probability ϵ (epsilon). Using this method enables the agent to experiment with the surroundings to learn about its dynamics and identify the best course of action. Simultaneously, it leverages the agent's present understanding to consistently arrive at optimal options. The epsilon-greedy approach was selected for the Lunar Lander scenario due to its efficacy in managing the exploration-exploitation trade-off in settings that necessitate cautious navigation and uncertain decision-making. The goal of the Lunar Lander mission is to safely land a spacecraft on a landing pad while using the least amount of fuel possible. The epsilon-greedy approach is a sensible option for this challenging assignment since it allows the agent to experiment with different landing techniques to effectively finish the mission while steadily enhancing its performance as it gains experience.

2.7 Experiments With Linear Q Learning Agent

We planned to experiment in conditions where the complexity of the environment increases. Initially, we start with the basic environment with no wind and turbulence. Then as we successfully train an agent to find a solution we will transition to the harder environment. We begin with the continuous observations and if the agent fails to learn we will discretize the observations and repeat the experiment.

2.8 Experiments With DQN Agent

Similarly, we planned to experiment in conditions where the complexity of the environment increases with added wind and turbulence. We experimented with neural networks with increasing complexity. We start with 2 layers consisting of 64 neurons and increase the number of layers until the agent learns.

2.9 Hyperparameters

For both agents, the initial timestep is set to 0, indicating the start of the learning process. The batch size, crucial for the mini-batch training approach, is fixed at 64, allowing for a balance between training speed and memory utilization. The discount factor (gamma), set at 0.99, emphasizes the importance of future rewards, enabling the agents to prioritize long-term gains over immediate rewards. The soft update parameter (tau), at a value of 0.001, ensures gradual updates to the target network, facilitating stable learning. The learning rate, a critical hyperparameter for the optimization algorithm, is maintained at 0.0001, supporting the convergence of the learning process by adjusting the magnitude of weight updates. Furthermore, the network update frequency is set to every 4 timesteps, balancing between the computational efficiency and the need for timely network updates.

Additionally, the training process incorporates an exploration-exploitation strategy through an epsilon-greedy policy, with an epsilon decay rate of 0.995. This approach allows the agent to progressively focus more on exploitation by reducing the epsilon value, starting from an initial value of 1.0 down to a minimum of 0.0. This gradual reduction in exploration ensures that the agent sufficiently explores the environment in the early stages of training, gradually shifting towards exploiting the learned policy as it becomes more confident in its decisions. The careful calibration of these hyperparameters plays a pivotal role in the learning efficacy and overall performance of both Linear and DQN agents in navigating complex environments and achieving optimal decision-making.

3 Results

In our first experiment with Linear Q Learning, in Figure. 1 we can see that the agent did not learn based on the learning curve. Also, it failed to land in all

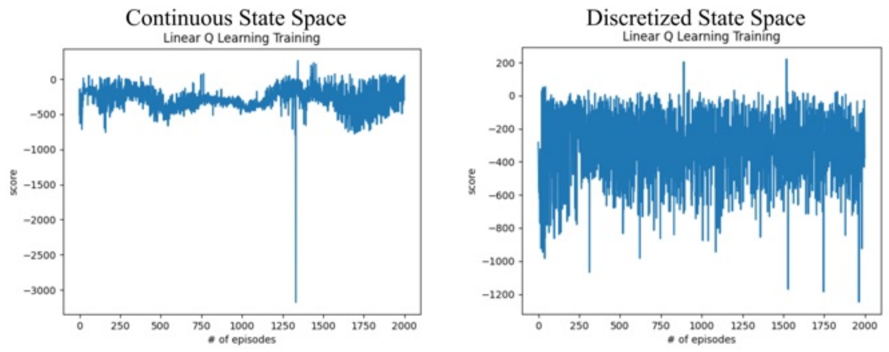

Fig. 1. Figures show the rewards of the agent concerning learning iterations. Figure on the left is the Linear Q agent performance with continuous state space and on the left is the same model's performance in discretized state space.

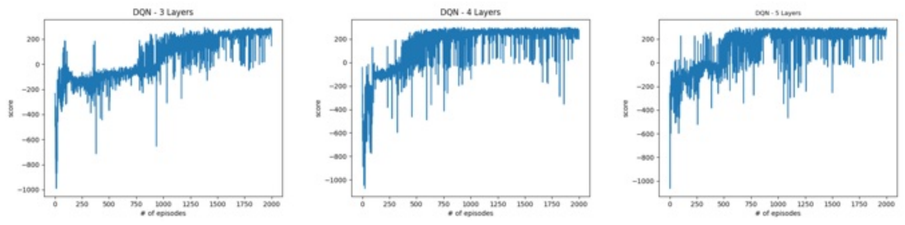

Fig. 2. No Wind, No Turbulence case for 3-4-5 layered DQN model training over number of episodes.

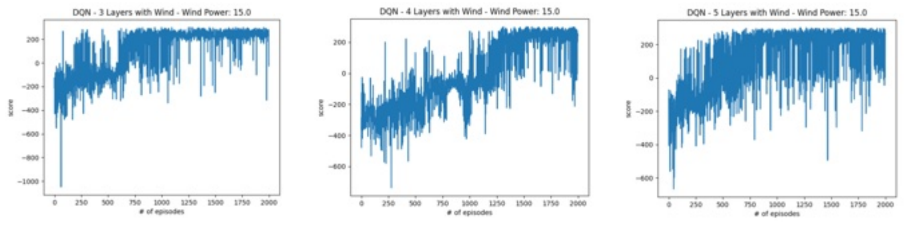

Fig. 3. Windy, No Turbulence case for 3-4-5 layered DQN model training over number of episodes.

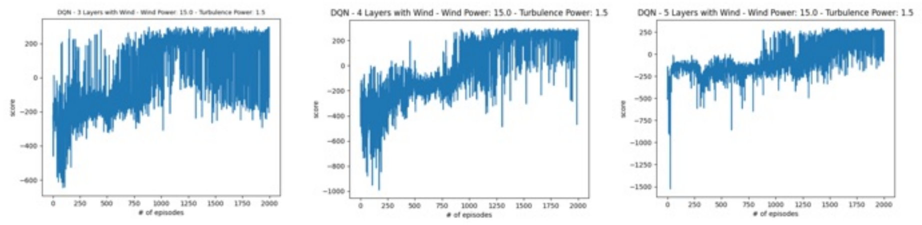

Fig. 4. Windy with Turbulence case for 3-4-5 layered DQN model training over number of episodes.

the trials in the testing phase. Therefore we continued our experiments with an improved approach. In the second experiment, we discretized the observations expecting that it can make it easier for the linear function approximator to learn the relationships. However, based on the learning curve once again we conclude that the agent did not learn. Similar to the previous version all the landings in the test phase failed. We conclude that linear Q-learning is not a suitable approach for this problem without feature engineering or other methods that allow the linear function approximator to learn the q values.

For the most basic condition in the environment without wind and turbulence, we see that the agent does not learn with 2 layers using the DQN method. In Figure. 2 we see the agent starts learning with 3 layers and performs best with 4 layers. Looking at the learning curves we observe that the agent starts getting more than 200 points more quickly as the networks get deeper. However, our tests show that 4-layered networks perform better than the others. Out of 1000 trials, we get 90.3% successful landings, with an average of 254.53 points. The random agent fails the landing 100% of the time. When we applied an independent two-sample t-test on the rewards they received in the testing phase we obtained a $p=0.0<0.05$ showing our result is statistically significant.

When the wind is added to the environment the results do not change drastically. In Figure. 3 we see that different than the previous experiment the learning curve for the 3-layer network starts getting successful landings faster than deeper networks. Also, the learning curve for the 3-layer network looks more stable. However, the model with 4 layers outperformed the others with an average score of 191.27 and a success rate of 73.5%. The random agent failed the landing 100% of the time once again. independent two-sample t-test on the rewards they received in the testing phase we obtained a $p=0.0<0.05$ showing our result is statistically significant.

With the addition of turbulence, we get the most difficult environment. The model with the 4 layers performs best once again with an average score of 153.88 points and a success rate of 69.2%. Unsurprisingly, the random agent fails all the landings once again. two-sample t-test on the rewards they received in the testing phase we obtained a $p=0.0<0.05$ showing our result is statistically significant.

4 Discussion

Experiments showed that the linear Q-learning approach is not successful even with discretized observations in the most basic condition. The Linear Q-Learning algorithm's ineffectiveness in addressing the Lunar Lander problem can be attributed to its inherent simplicity. Linear Q-Learning, which uses a linear function to approximate the Q-value function, struggles with high-dimensional state spaces and environments requiring the capture of complex state-action relationships. The Lunar Lander problem presents challenges beyond the capacity of linear approximation methods. This limitation is a strong reminder of the linear model's inability to handle the non-linear dynamics often present in more complex RL environments.

On the contrary, the DQN approach, which utilizes deep neural networks to approximate the Q-value function, demonstrates significant success in tackling the Lunar Lander challenge. The strength of DQN lies in its ability to learn non-linear approximations of the Q-value function, enabling it to effectively process and act upon high-dimensional sensory inputs. By leveraging the representational power of deep neural networks, DQN can capture the intricate relationships between actions and states in complex environments, a capability that linear models lack.

The transition from simpler models to more sophisticated ones like DQN highlights an essential consideration in RL. The scalability of solution approaches with environmental complexity. As the complexity of the task increases, the need for more complex models becomes obvious. In our study, the improvement of the DQN architecture through the addition of more layers further underlines this point. Increasing the depth of the neural network allowed for a more nuanced understanding and representation of the environment, leading to improved performance. This adaptation underscores the adaptability of DQN architectures to the demands of more complex environments, where the intricacies of state-action relationships exceed the representational capabilities of simpler models.

Contrary to what we expected the added wind and turbulence do not increase the complexity of the environment as the way that we expected. We expected that a more complex model would perform better as the complexity of the environment increases. After thinking about the problem more deeply we realized that wind and turbulence affect the velocity and angular velocity of the lander but this information is captured in the observation space. The only thing that changes is the variety of possible combinations introduced to the network. Our testing in a no wind no turbulence environment with the agent that is trained in the wind and turbulence condition performs better than the agent trained specifically for that environment supports this interpretation of the results. The model generalizes better because it encounters a larger variety of states and this helps the agent to perform better.

References

1. Gadgil, S., Xin, Y., Xu, C.: Solving The Lunar Lander Problem under Uncertainty using Reinforcement Learning. arXiv:2011.11850 [cs.LG] (2020), https://doi.org/10.48550/arxiv.2011.11850
2. Gymnasium: Gymnasium documentation. Accessed: [insert the date you accessed this resource], https://gymnasium.farama.org/environments/box2d/lunar_lander/#notes
3. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing Atari with Deep Reinforcement Learning. arXiv:1312.5602 [cs.LG] (2013), https://doi.org/10.48550/arxiv.1312.5602

5 Appendix

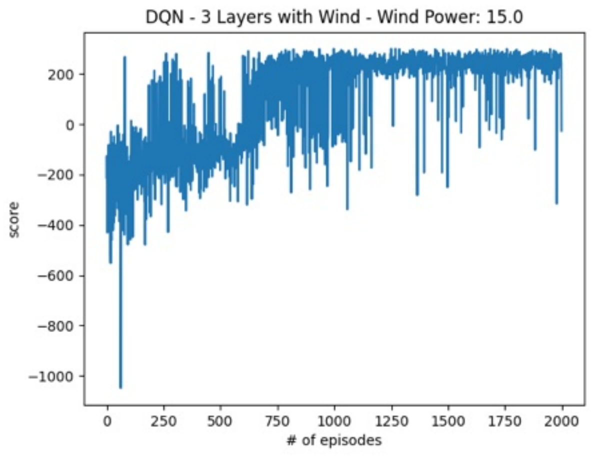

Fig. 5. Figure 6: model_DQN_3_layers_wind_15.0_no_turbulence

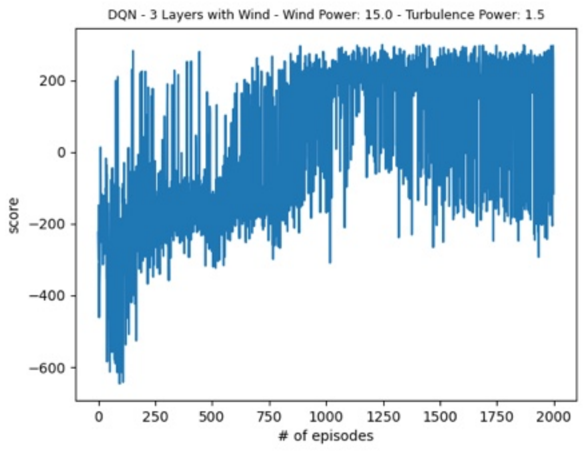

Fig. 6. Figure 7: model_DQN_3_layers_wind_15.0_turbulence_1.5

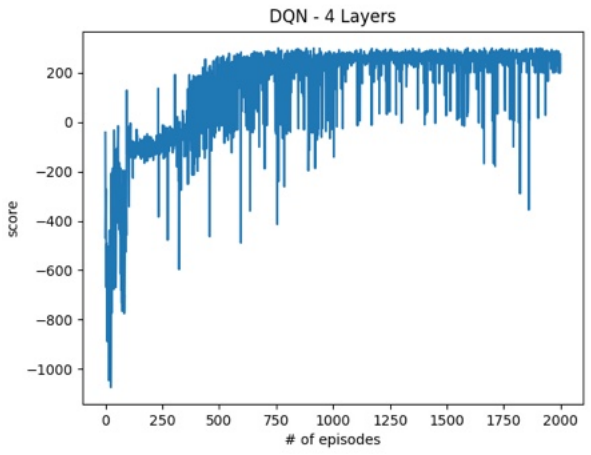

Fig. 7. Figure 8: model_DQN_4_layers_no_wind_no_turbulence

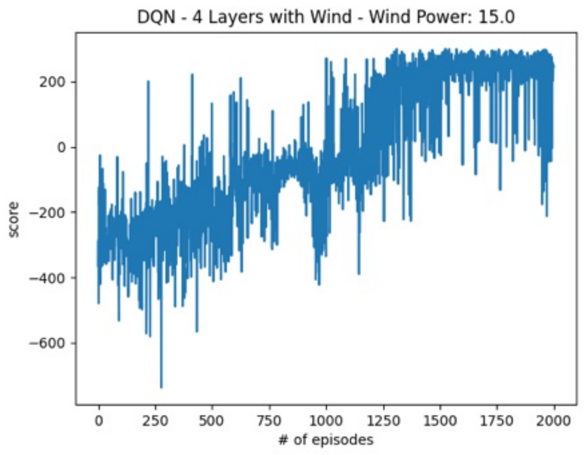

Fig. 8. Figure 9: model_DQN_4_layers_wind_15.0_no_turbulence

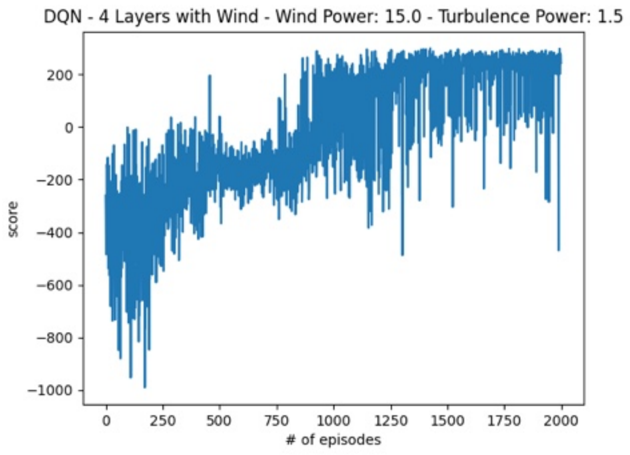

Fig. 9. Figure 10: model_DQN_4_layers_wind_15.0_turbulence_1.5

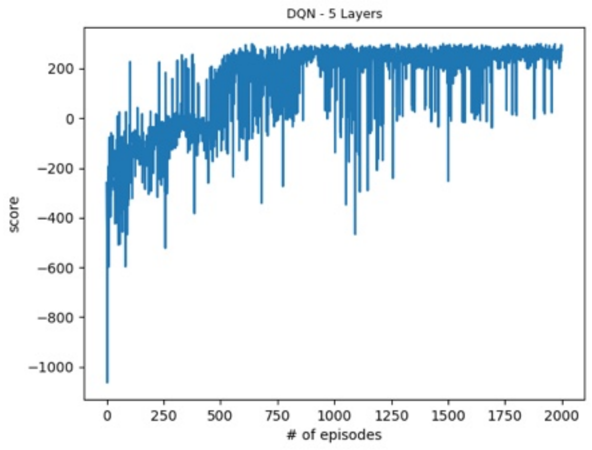

Fig. 10. Figure 11: model_DQN_5_layers_no_wind_no_turbulence

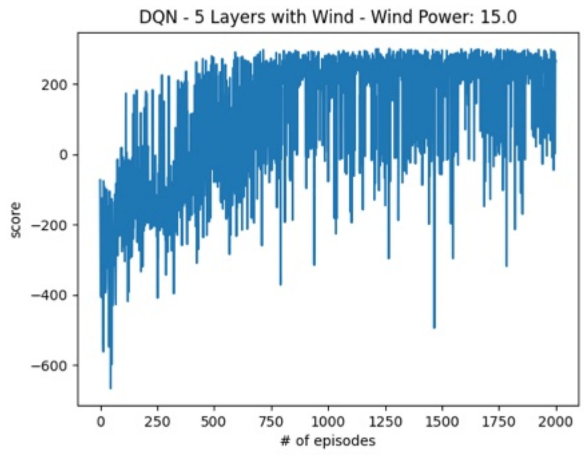

Fig. 11. Figure 12: model_DQN_5_layers_wind_15.0_no_turbulence

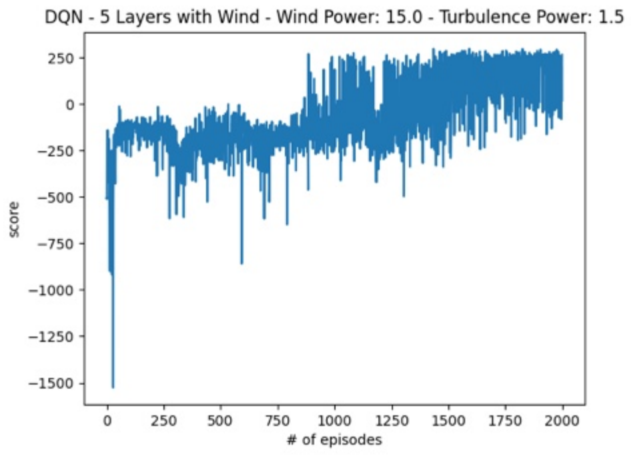

Fig. 12. Figure 13: model_DQN_5_layers_wind_15.0_turbulence_1.5

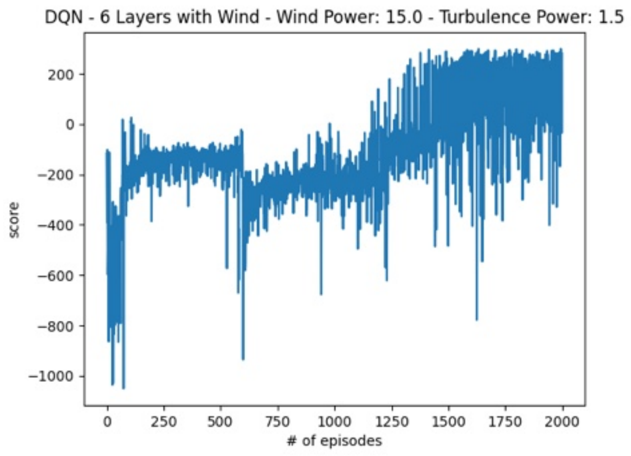

Fig. 13. Figure 14: model_DQN_6_layers_wind_15.0_turbulence_1.5

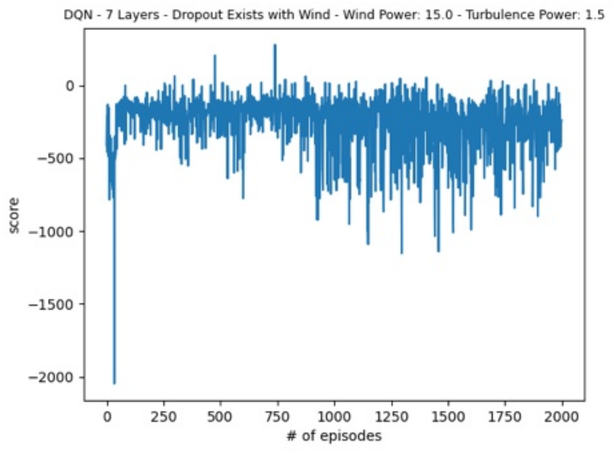

Fig. 14. Figure 15: model_DQN_7_layers_dropout_wind_15.0_turbulence_1.5

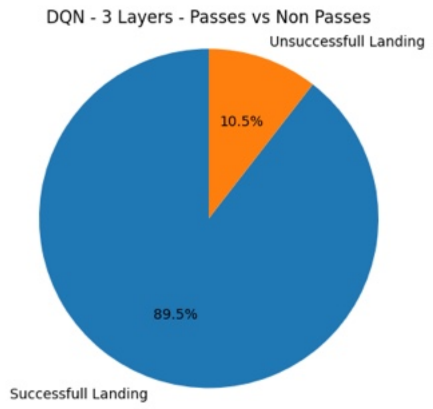

Fig. 15. Figure 16: piemodel_DQN_3_layers_no_wind_no_turbulence

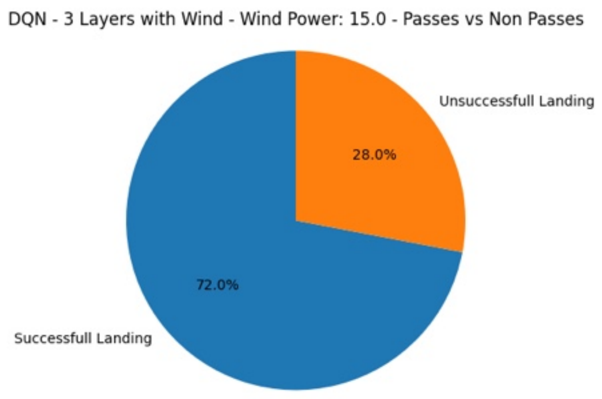

Fig. 16. Figure 17: piemodel_DQN_3_layers_wind_15.0_no_turbulence

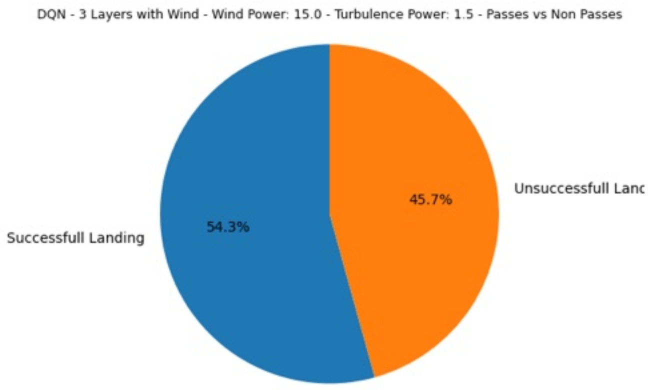

Fig. 17. Figure 18: piemodel_DQN_3_layers_wind_15.0_turbulence_1.5

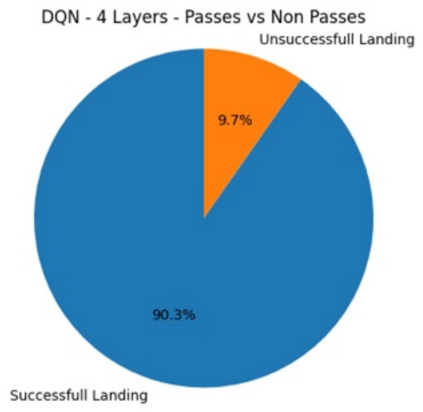

Fig. 18. Figure 19: piemodel_DQN_4_layers_no_wind_no_turbulence

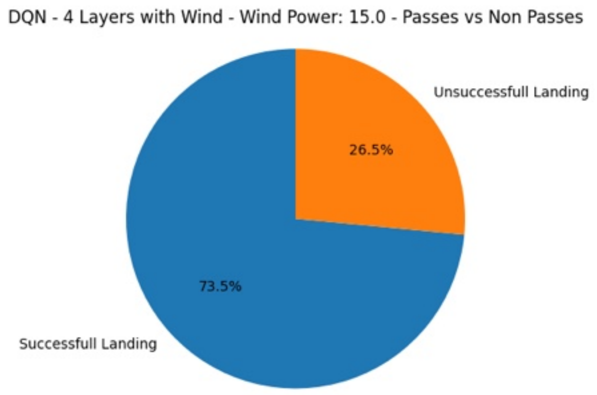

Fig. 19. Figure 20: piemodel_DQN_4_layers_wind_15.0_no_turbulence

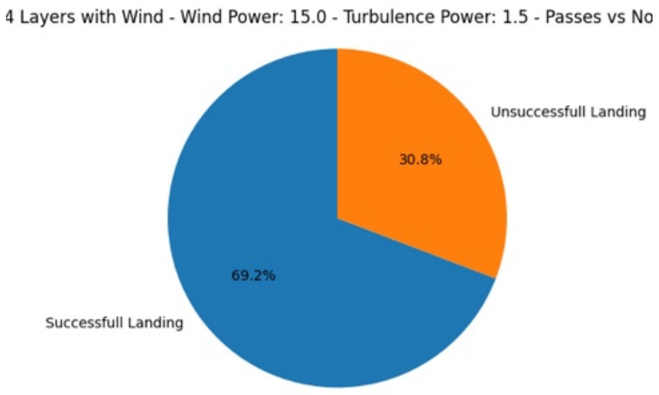

Fig. 20. Figure 21: piemodel_DQN_4_layers_wind_15.0_turbulence_1.5

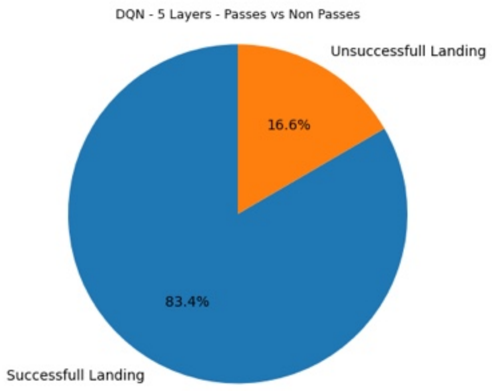

Fig. 21. Figure 22: piemodel_DQN_5_layers_no_wind_no_turbulence

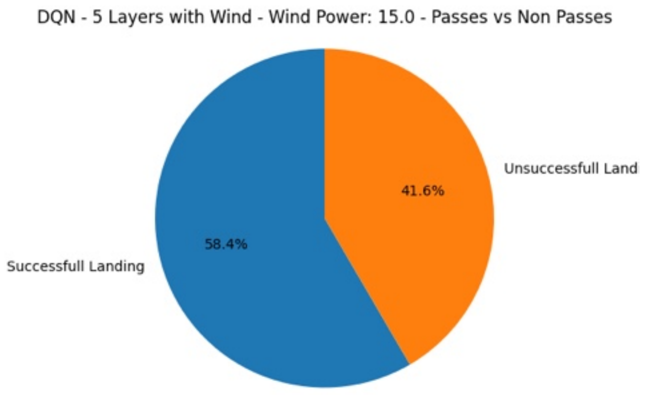

Fig. 22. Figure 23: piemodel_DQN_5_layers_wind_15.0_no_turbulence

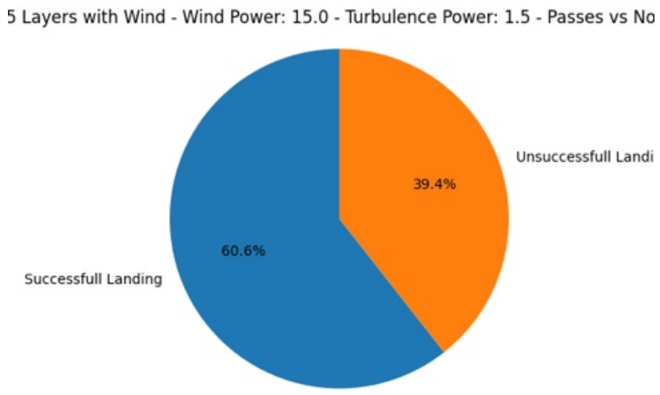

Fig. 23. Figure 24: piemodel_DQN_5_layers_wind_15.0_turbulence_1.5

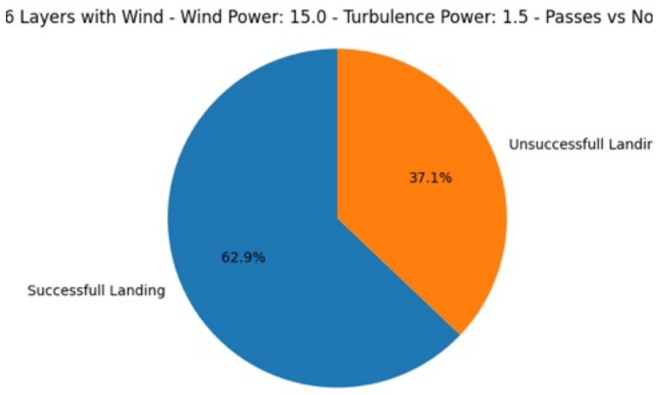

Fig. 24. Figure 25: piemodel_DQN_6_layers_wind_15.0_turbulence_1.5

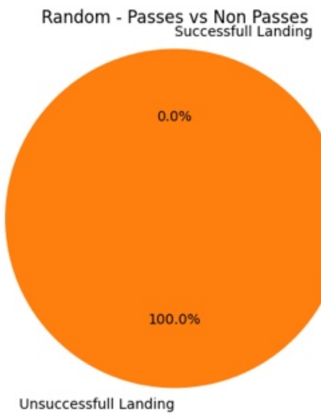

Fig. 25. Figure 26: piemodel_Random_no_wind_no_turbulence

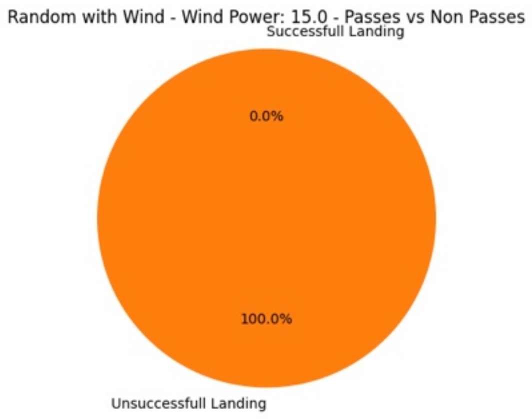

Fig. 26. Figure 27: piemodel_Random_wind_15.0_no_turbulence

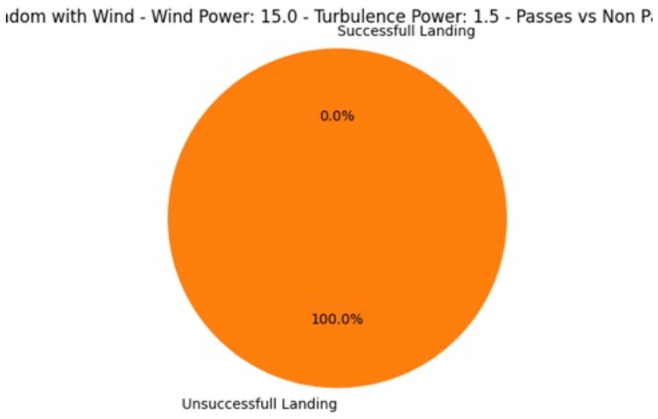

Fig. 27. Figure 28: piemodel_Random_wind_15.0_turbulence_1.5

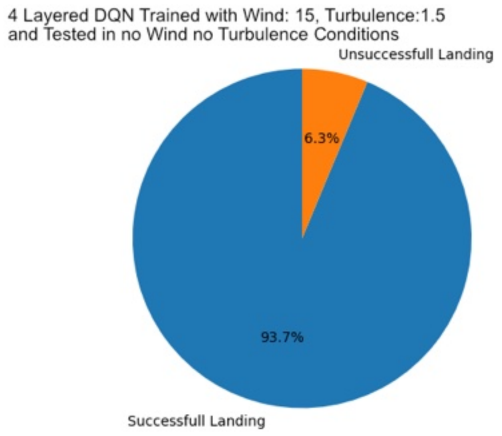

Fig. 28. Figure 29: 4 Layered DQN trained with wind and turbulence but tested on no wind no turbulence conditions.