Solving The Lunar Lander Problem under Uncertainty using Reinforcement Learning

Yunfeng Xin, Soham Gadgil, Chengzhe Xu

Abstract—Reinforcement Learning (RL) is an area of machine learning concerned with enabling an agent to navigate an environment with uncertainty in order to maximize some notion of cumulative long-term reward. In this paper, we implement and analyze two different RL techniques, Sarsa and Deep Q-Learning on OpenAI Gym's *LunarLander-v2* environment. We then introduce additional uncertainty to the original problem to test the robustness of the mentioned techniques. With our best models, we are able to achieve an average reward of 170 with the Sarsa agent and 200 with the Deep Q-Learning agent on the original problem. We also experiment with additional uncertainty using these trained models, and discuss how the agents perform under these added uncertainties.

Index Terms—Deep Reinforcement Learning, Neural Networks, Q-Learning, POMDP, Sarsa

1 Introduction

Over the past several years, reinforcement learning [1] has been proven to have a wide variety of successful applications including robotic control [2], [3]. Different approaches have been proposed and implemented to solve such problems [4], [5]. In this paper, we solve a well-known robotic control problem — the lunar lander problem using different reinforcement learning algorithms, and then test the agents' performances under environment uncertainties and agent uncertainties.

2 Problem Definition

We aim to solve the lunar lander environment in the OpenAI gym kit using reinforcement learning methods.[1] The environment simulates the situation where a lander needs to land at a specific location under low-gravity conditions, and has a well-defined physics engine implemented.

The main goal of the game is to direct the agent to the landing pad with as softly and fuel-efficiently as possible. The state space is continuous as in real physics, but the action space is discrete.

3 Related Work

There has been previous works done in solving the lunar lander environment using different techniques. [6] makes use of modified policy gradient techniques for evolving neural network topologies. [7] uses a control-model-based approach that learns the optimal control parameters instead of the dynamics of the system. [8] explores spiking neural networks as a solution to OpenAI virtual environments.

These approaches show the effectiveness of a particular algorithm for solving the problem. However, they do not consider additional uncertainty. Thus, we aim to first solve the lunar lander problem using traditional Q-learning techniques, and then analyze different techniques for solving the problem as well as verify the robustness of these techniques as additional uncertainty is added.

1. Our code is available at https://github.com/rogerxcn/lunar_lander_project

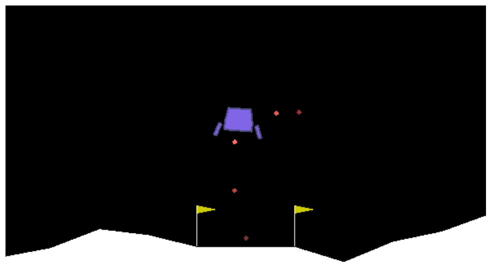

Fig. 1: Visulization of the lunar lander problem.

4 Model

4.1 Framework

The framework used for the lunar lander problem is gym, a toolkit made by OpenAI [9] for developing and comparing reinforcement learning algorithms. It supports environments for various learning environments, ranging from Atari games to robotics. The simulator we use is called *Box2D* and the environment is called *LunarLander-v2*.

4.2 Observations and State Space

The observation space determines various attributes about the lander. Specifically, there are 8 state variables associated with the state space, as shown below:

$$state \rightarrow \begin{cases} x \text{ coordinate of the lander} \\ y \text{ coordinate of the lander} \\ v_x\text{, the horizontal velocity} \\ v_y\text{, the vertical velocity} \\ \theta\text{, the orientation in space} \\ v_\theta\text{, the angular velocity} \\ \text{Left leg touching the ground (Boolean)} \\ \text{Right leg touching the ground (Boolean)} \end{cases}$$

All the coordinate values are given relative to the landing pad instead of the lower left corner of the window. The x coordinate of the lander is 0 when the lander is on the

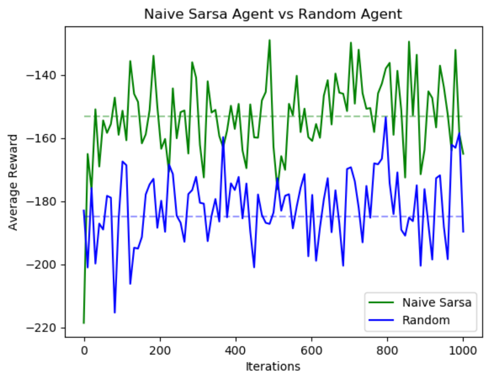

Fig. 2: Performance comparison between Sarsa agent under naive state discretization and random agent.

line connecting the center of the landing pad to the top of the screen. Therefore, it is positive on the right side of the screen and negative on the left. The y coordinate is positve at the level above the landing pad and is negative at the level below.

4.3 Action Space

There are four discrete actions available: do nothing, fire left orientation engine, fire right orientation engine, and fire main engine. Firing the left and right engines introduces a torque on the lander, which causes it to rotate, and makes stabilizing difficult.

4.4 Reward

Defining a proper reward directly affects the performance of the agent. The agent needs to maintain both a good posture mid-air and reach the landing pad as quickly as possible. Specifically, in our model, the reward is defined to be:

$$\begin{aligned}\text{Reward}(s_t) = &-100 * (d_t - d_{t-1}) - 100 * (v_t - v_{t-1}) \\ &-100 * (\omega_t - \omega_{t-1}) + \text{ hasLanded}(s_t)\end{aligned}$$

where d_t is the distance to the landing pad, v_t is the velocity of the agent, and ω_t is angular velocity of the agent at time t. $hasLanded()$ is the reward function of landing, which is a linear combination of the boolean state values representing whether the agent has landed softly on the landing pad.

With this reward function, we encourage the agent to lower its distance to the landing pad, decrease the speed to land smoothly, keep the angular speed at minimum to prevent rolling, and not to take off again after landing.

5 Approaches

5.1 Sarsa

Since we do not encode any prior knowledge about the outside world into the agent and the state transition function is hard to model, Sarsa [10] seems to be a reasonable approach to train the agent using an exploration policy. The update rule for Sarsa is:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

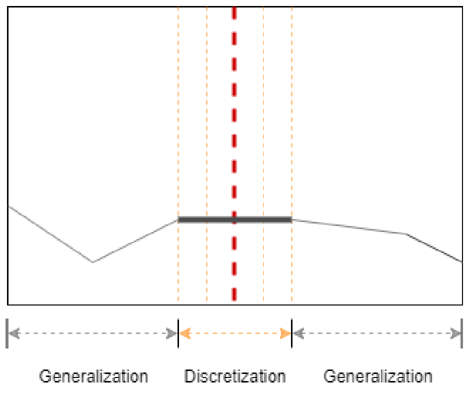

Fig. 3: Diagram showing the state discretization and generalization scheme of the x coordinate state variable.

At any given state, the agent chooses the action with the highest Q value corresponding to:

$$a = argmax_{a \in actions} Q(s, a)$$

From the equation we can see that we need to discretize the states and assign Q values for each state-action pair, and that we also need to assign a policy to balance exploitation and exploration since Sarsa is an on-policy algorithm.

Intuitively, a simple exploration policy can be the ϵ-greedy policy [11], where the agent randomly chooses an action with probability ϵ and chooses the best actions with probability $1 - \epsilon$. A simple way of discretizing the state is to divide each continuous state variable into several discrete values. However, as shown in Fig. 2, the result shows that the agent can only reach marginally better performance than a random agent, and cannot manage to get positive rewards even after 10,000 episodes of training. It then becomes obvious that we cannot simply adopt these algorithms out-of-the-box, and we need to tailor them for our lunar lander problem.

5.1.1 State Discretization

There are five continuous state variables and two boolean state variables, so the complexity of the state space is on the order of $O(n^5 \times 2 \times 2)$, where n is the number of discretized values for each state variable. Thus, even discretizing each state variable into 10 values can produce 400,000 states, which is far too large to explore within a reasonable number of training episodes. It explains the poor performance we were seeing previously. Therefore, we need to devise a new discretization scheme for these states.

Specifically, we examine the initial Q values learned by the agent and observe that the agent wants to move to the right when it is in the left half of the space, and move to the left when it is in the right half of the space. As a result, all the x coordinates far from the center can be generalized into one single state because the agent will always tend to move in one direction (as shown in Fig. 3), which helps reduce the state space.

Therefore, we define the discretization of the x coordinate at any state with the discretization step size set at 0.05 as:

$$d(x) = \min(\lfloor \frac{n}{2} \rfloor, \max(-\lfloor \frac{n}{2} \rfloor, \frac{x}{0.05}))$$

The same intuition of generalization is applied to other state variables as well. In total, we experiment with 4 different ways of discretizing and generalizing the states with different number of discretization levels.

5.1.2 *Exploration Policy*

As the size of the state space exceeds 10,000 even after the process of discretization and generalization, the probability ϵ in the ϵ-greedy policy needs to be set to a fairly large number (around 20%) for the agent to explore most of the state within a reasonable number of episodes. However, this means that the agent will pick a sub-optimal move once every five steps.

In order to minimize the performance impact of the ϵ-greedy policy while still getting reasonable exploration, we decay ϵ in different stages of training. The intuition is that the agent in the initial episodes knows little about the environment, and thus more exploration is needed. After an extensive period of exploration, the agent has learnt enough information about the outside world, and needs to switch to exploitation so that the Q value for each state-action pair can eventually converge.

Specifically, the epsilon is set based on the following rules:

$$\epsilon = \begin{cases} 0.5 & \text{\#Iteration} \in [0, 100) \\ 0.2 & \text{\#Iteration} \in [100, 500) \\ 0.1 & \text{\#Iteration} \in [500, 2500) \\ 0.01 & \text{\#Iteration} \in [2500, 7500) \\ 0 & \text{\#Iteration} \in [7500, 10000) \end{cases}$$

5.2 Deep Q-Learning

The deep Q-Learning method [12], [13] makes use of a multi-layer perceptron, called a Deep Q-Network (DQN), to estimate the Q values. The input to the network is the current state (8-dimensional in our case) and the outputs are the Q values for all state-action pairs for that state. The Q-Learning update rule is as follows:

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

The optimal Q-value $Q^*(s,a)$ is estimated using the neural network with parameters θ. This becomes a regression task, so the loss function at iteration i is obtained by the temporal difference error:

$$\mathcal{L}_i(\theta_i) = \mathbb{E}_{(s,a)\sim\rho(.)}[(y_i - Q(s,a;\theta_i))^2]$$

where

$$y_i = \mathbb{E}_{s'\sim\mathcal{E}}[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})]$$

Here, θ_{i-1} are the network parameters from the previous iteration, $\rho(s,a)$ is a probability distribution over states s and actions a, and $\mathcal{E}$ is the environment the agent interacts with. Gradient descent is then used to optimise the loss function and update the neural network weights. The neural network parameters from the previous iteration, θ_{i-1}, are kept fixed while optimizing $\mathcal{L}_i(\theta_i)$.

One of the challenges here is that the successive samples are highly correlated since the next state depends on the current state and action. This is not the case in traditional supervised learning problems where the successive samples are iid. To tackle this problem, the transitions encountered by the agent are stored in a replay memory $\mathcal{D}$. Random minibatches of transitions $\{s,a,r,s'\}$ are sampled from $\mathcal{D}$ during training to train the network. This technique is called *experience replay* [14].

We use a 3 layer neural network, as shown in Fig. 3, with 128 neurons in the hidden layers. We use $ReLU$ activation for the hidden layers and $LINEAR$ activation for the output layer. The number of hidden neurons are chosen based on analyzing different values, as shown in section 6.2.1. The learning rate used is 0.001 and the minibatch size is 64.

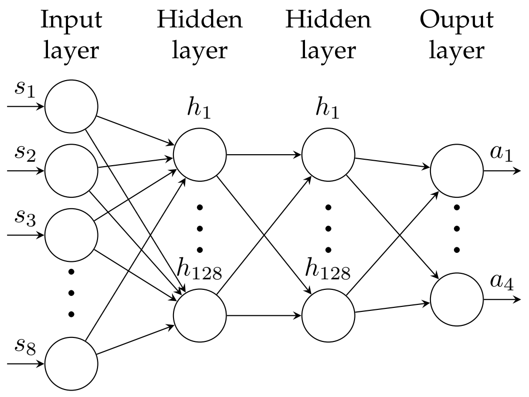

Fig. 4: Neural Network used for Deep Q-Learning

5.2.1 *Exploration Policy*

Similar to Sarsa, an improved ϵ-greedy policy is used to select the action, with ϵ starting at 1 to favor exploration and decaying by 0.996 for every iteration, until it reaches 0.01, after which it stays the same.

6 Experiments

6.1 The Original Problem

The goal of the problem is to direct the lander to the landing pad between two flag poles as softly and fuel efficiently as possible. Both legs of the lander should be in contact with the pad while landing. The lander should reach the landing pad as quickly as possible, while maintaining a stable posture and minimum angular speed. Also, once landed, the lander should not take off again. In the original problem, uncertainty is added by applying a randomized force to the center of the lander at the beginning of each iteration. This causes the lander to be pushed in a random direction. The lander must recover from this force and head to the landing pad.

We experiment with tackling the original problem using Sarsa and deep Q-learning as described in our approach section, and our observations are demonstrated in section 7.

6.2 Introducing Additional Uncertainty

After solving the original lunar lander problem, we analyze how introducing additional uncertainty can affect the performance of the agents and evaluate their robustness to different uncertainties.

6.2.1 *Noisy Observations*

Retrieving the exact state of an object in a physical world can be hard, and we need to rely on noisy observations such as a radar to infer the real state of the object. Thus, instead of directly using the exact observations provided by the environment, we add a zero-mean Gaussian noise of scale 0.05 into our observation of the location of the lander. The standard deviation is deliberately set to 0.05, which corresponds to our discretization step size. Specifically, for each observation of x, we sample a random zero-mean Gaussian noise

$$s \sim \mathcal{N}(\mu = 0, \sigma = 0.05)$$

and add the noise to the observation, so that the resulting random variable becomes

$$Observation(x) \sim \mathcal{N}(x, 0.05)$$

We then evaluate the resulting performance of two agents: one using the original Q values from the original problem, and the other using the Q values trained under such noisy observations.

We notice that we can frame this problem as a POMDP, and compare its performance with the two Sarsa agents mentioned above. We calculate the alpha vector of each action using one-step lookahead policy using the Q values from the original problem, and calculate the belief vector using the Gaussian probability density function

$$\mathrm{PDF}(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(o(x)-x)^2}{2\sigma^2}}$$

This way, we can get the expected utility of each action under uncertainty by taking the inner product of the corresponding alpha vector and the belief vector. The resulting policy simply picks the action with the highest expected utility.

Notice that we could have used transition probabilities of locations to assist determining the exact location of the agent. However, after experimenting with different transition probability functions, we concluded that the transition probability in a continuous space is very hard to model, and naive transition probability functions will cause the agent to perform even worse than the random agent.

6.2.2 *Random Engine Failure*

Another source of uncertainty in the physical world can be random engine failures due to the various unpredictable conditions in the agent's environment. The model needs to be robust enough to overcome such failures without impacting performance too much. To simulate this, we introduce action failure in the lunar lander. The agent takes the action provided 80% of the time, but 20% of the time the engines fail and it takes no action even though the provided action is firing an engine.

6.2.3 *Random Force*

Uncertainty can also come from unstable particle winds in the space such as solar winds, which result in random forces being applied to the center of the lander while landing. The model is expected to manage the random force and have stable Q maps with enough fault tolerance.

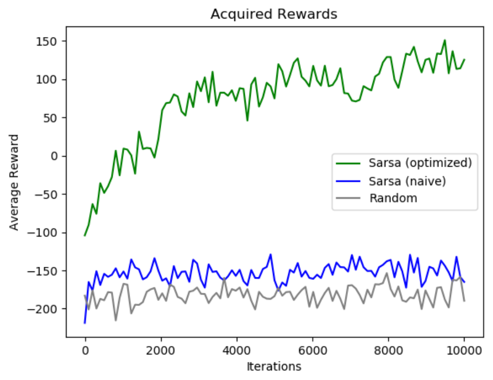

Fig. 5: Performance comparison of our state discretization and policy scheme (green), naive state discretization and policy scheme (blue), and random agent (grey).

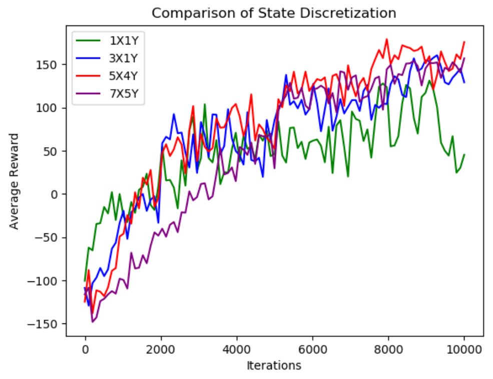

Fig. 6: Performance comparison of different state discretization schemes in Sarsa.

We apply a random force each time the agent interacts with the environment and modify the state accordingly. The force is sampled from a Gaussian distribution for better simulation of real-world random forces. The mean and variance of the Gaussian distribution are set in proportion to the engine power to avoid making the random force either too small to have any effect on the agent or too large to maintain control of the agent.

7 Results and Analysis

7.1 Sarsa

7.1.1 *The Original Problem*

Fig. 5 shows the average reward acquired by the the random agent and the Sarsa agent with naive state discretization and our customized discretization scheme. With a naive discretization which quantizes each state variable into 10 equal steps, the agent cannot learn about the outside world very effectively even after 10,000 episodes of training. Due to the huge state space, the acquired reward is only marginally better than the random agent. In contrast, with our customized discretization scheme which combines small-step discretization and large-step generalization, the agent is able to learn the Q values rapidly and gather positive rewards after 500 episodes of training. The results show that for continuous state spaces, proper discretization and generalization of the states help the agent learn the values faster and better.

Fig. 6 also shows how different discretization schemes affects the learning speed and the final performance the agent is able to achieve. The notation $aXbY$ is used to denote that the the x coordinate is discretized into a levels while the y coordinate is discretized into b levels. The results indicate that as the number of discretization level increases, the agent in general learns the Q values more slowly, but is able to achieve higher performance after convergence.

7.1.2 *Handling Noisy Observations*

Fig. 7 shows the results after feeding the noisy observations under three agents. The first agent uses the Q values learnt from the original problem under discretization scheme 5X4Y and takes the noisy observations as if they were exact. The second agent is re-trained under the noisy environment using the noisy observations. The third agent uses the Q values learnt from the orginal problem, but uses one-step lookahead alpha vectors to calculate the expected reward for each action.

Each data point in Fig. 7 represents the average acquired reward in 10 episodes and can help eliminate outliers. The results show that the POMDP agent (POMDP in Fig. 7) receives the highest average rewards and outperforms the other agents. Of the other two agents, the agent trained under noisy observations (Trained Q in Fig. 7) fails to generalize information from these episodes.

In general, there is a significant performance impact in terms of average acquired rewards with the added uncertainty of noisy observation, and such a result is expected: when the agent is close to the center of the space, a noisy x observation can significantly change the action which the agent picks. For example, when $x = 0.05$, a noisy observation has a 15.87% probability of flipping the sign so that $x < 0$ according the Gaussian cumulative distribution function

$$\text{CDF}(x) = \frac{1}{\sqrt{2\pi} \times 0.05} \int_{-\infty}^{-0.05} e^{-\frac{(x-0.05)^2}{2\times 0.05^2}} dx$$

This means that the noise has a decent chance of tricking the agent into believing that it is in the left half of the screen while it is in fact in the right half of the screen. Therefore, the agent will pick an action that helps the agent move right, instead of original optimal action of moving left.

The POMDP agent has the correct learned Q value and takes the aforementioned sign-flipping observation scenario into account using the belief vector, which explains why it is performing the best by getting the highest average rewards. The agent trained under noisy observation, however, is learning the wrong Q value in the wrong state due to the noisy observation and does not take the noisy observation into account. Thus, it is performing the worst of all three agents.

7.1.3 *Handling Random Force*

Fig. 8 shows the result after applying different random forces to the Sarsa agent under state discretization of 5X4Y.

In the experiments, we introduce three kinds of random forces: regular, medium and large. In the regular cases, we ensure that random forces do not go too much beyond the agent's engine power. In the medium case, we relax that constraint, and in the large case, we ensure that the agent cannot control itself because the random force becomes much larger than the engine power. The details are described as follows:

1) regular_00: mean equals 0 and variance equals $engine\_power/3$

2) regular_01: mean equals 0 on the x-axis and $engine\_power/6$ on the y-axis, variance equals $engine\_power/3$

3) regular_10: mean equals $engine\_power/6$ on the x-axis and 0 on the y-axis, variance equals $engine\_power/3$

4) regular_11: mean equals $engine\_power/6$ on both x-axis and y-axis, variance equals $engine\_power/3$

5) medium: mean equals $engine\_power$ on both x-axis and y-axis, variance equals $engine\_power * 3$

6) large: mean equals $engine\_power * 2$ on both x-axis and y-axis, variance equals $engine\_power * 5$

The result suggests that agents would perform well and offset the effect of the random force in regular cases, while in the medium and large cases where random forces are more likely to exceed the maximum range engines could compensate, there would be an obvious reduction in reward indicating that the random forces make landing harder. The results reveal the fact that Sarsa agents have learned a robust and smooth Q map where similar state-action pairs would have similar Q value distributions. Slight state variations

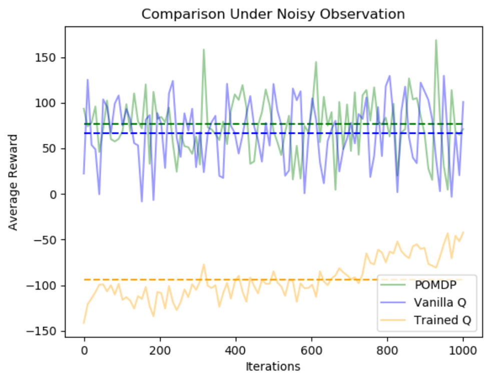

Fig. 7: Comparison of different agent performance under noisy observations.

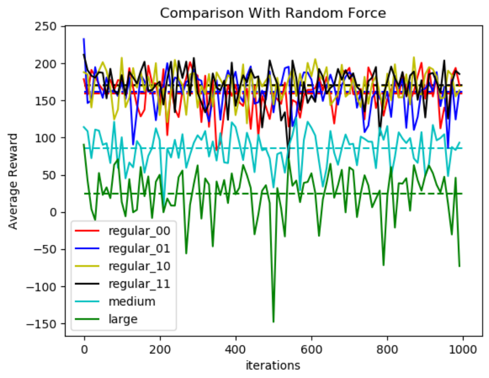

Fig. 8: Comparison of agent performance under different random forces.

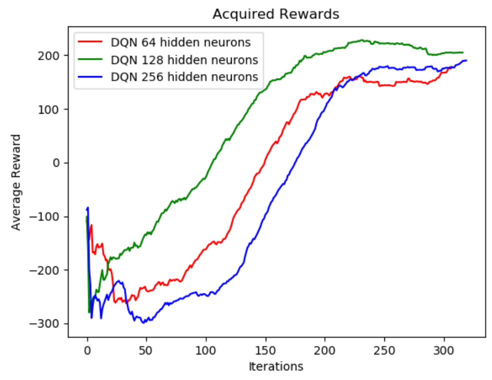

Fig. 9: Average reward obtained by DQN agent for different number of hidden neurons

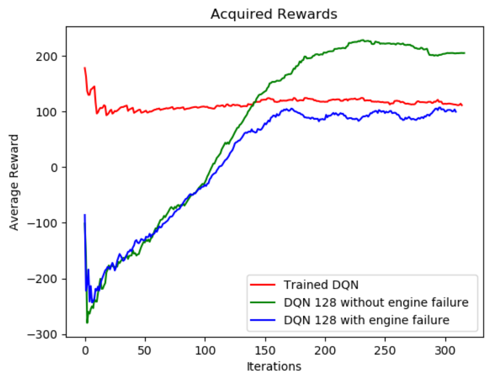

Fig. 10: Comparison of DQN agents under engine failure

caused by random forces would have small influences on Q value and action selection, which increases the fault tolerance of the agent.

When state variations become too large, Q maps would be noisy and total rewards would decrease, in which case agents tend to lose control because of the random forces.

7.2 Deep Q-Learning

7.2.1 The Original Problem

Fig. 9 shows the average reward obtained (over the previous 100 episodes) by the DQN agent for different number of neurons in the hidden layers of the neural network used for predicting the Q-values. At the start, the average reward is highly negative since the agent is essentially exploring all the time and collecting more transitions for storing in memory $\mathcal{D}$. As the agent learns, the Q-values start to converge and the agent is able to get positive average reward. For both the plots, the DQN agent is able to converge pretty quickly, doing so in ~ 320 iterations. The DQN implementation performs well and results in good scores. However, the training time for the DQN agent is quite long (~ 8 hours). It is observed that the neural network with 128 neurons in the hidden layers performs the best and it is chosen as the hyperparameter value.

7.2.2 Handling Random Engine Failure

Fig. 10 shows a comparison between the agent using Q-values learned from the original problem when there are engine failures, the re-trained DQN agent when there are engine failures, and the original DQN agent when there are no engine failures. For all the plots, the number of neurons in the hidden layers of the neural network is 128.

The trained DQN agent performs well on the new environment with random engine failures and is able to obtain positive average rewards of $100+$. This shows that even without retraining, the original agent is able to adjust to the uncertainty.

For the re-trained agents, at the start, the curves are almost the same since this is the exploration phase, and both agents are taking random actions and engine failure does not affect the reward much. However, in the later iterations, as the agent learns the environment, the lander without engine failure achieves higher average reward. This is expected since the random engine failures require the agent to adjust to the unexpected behavior while trying to maximise reward. However, even with engine failure, the agent shows the same increasing trend in average reward as the original problem and is able to achieve positive rewards around 100. This shows that the DQN agent is able to estimate the optimal Q-values well even with the added uncertainty.

8 Conclusion

In conclusion, we observe that both the Sarsa and the DQN agents perform well on the orignal lunar lander problem. When we introduce additional uncertainty, the Sarsa agent is able to handle random forces and the DQN agent is able to handle engine failures quite well. However, the re-trained Sarsa agent fails to handle noisy observations. This is understandable since the noisy observations affect the underlying state space and the agent isn't able to generalize information from its environment during training. The POMDP agent performs well with noisy observations and is able to get positive average rewards since it makes use of belief vectors to model a distribution over the possible states.

For future work, we would like to combine the different uncertainties together and analyze how the different agents perform. Also, we would like to consider all uncertainties while testing the different agents rather than testing the agents on different additional uncertainties. This will provide a more holistic overview of the robustness of the different agents.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[2] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[3] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange, "Reinforcement learning for robot soccer," *Autonomous Robots*, vol. 27, no. 1, pp. 55–73, 2009.

[4] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," 1996.

[5] C. Szepesvári, "Algorithms for reinforcement learning," *Synthesis lectures on artificial intelligence and machine learning*, vol. 4, no. 1, pp. 1–103, 2010.

[6] A. Banerjee, D. Ghosh, and S. Das, "Evolving network topology in policy gradient reinforcement learning algorithms," in *2019 Second International Conference on Advanced Computational and Communication Paradigms (ICACCP)*, Feb 2019, pp. 1–5.

[7] Y. Lu, M. S. Squillante, and C. W. Wu, "A control-model-based approach for reinforcement learning," 2019.

[8] C. Peters, T. Stewart, R. West, and B. Esfandiari, "Dynamic action selection in openai using spiking neural networks," 2019. [Online]. Available: https://www.aaai.org/ocs/index.php/FLAIRS/FLAIRS19/paper\/view/18312/17429

[9] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[10] M. J. Kochenderfer, *Decision making under uncertainty: theory and application*. MIT press, 2015.

[11] E. Rodrigues Gomes and R. Kowalczyk, "Dynamic analysis of multiagent q-learning with -greedy exploration," in *Proceedings of the 26th Annual International Conference on Machine Learning*, ser. ICML '09. New York, NY, USA: ACM, 2009, pp. 369–376. [Online]. Available: http://doi.acm.org/10.1145/1553374.1553422

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

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

[14] L.-J. Lin, "Reinforcement learning for robots using neural networks," Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, Tech. Rep., 1993.