Reinforcement Learning & Q-Learning Networks With OpenAI's LunarLander Environment

Ryan Ueda Teo Shao Ming
Singapore Polytechnic
School of Computing
Singapore
ryanueda.21@ichat.sp.edu.sg

***Abstract*— Reinforcement Learning was first introduced by Rich Sutton, a professor of Computer Science at University of Alberta. Otherwise known as the father of Reinforcement Learning due to his significant contributions to the topics, namely temporal difference learning and policy gradient methods. In this paper, various Q-Learning frameworks and their workings, as well as implementations will be discussed to solve the OpenAI LunarLander-v2 environment.**

Keywords—reinforcement learning, policy, gradient, actor, critic, temporal difference, openai, lunarlander

I. INTRODUCTION

Reinforcement learning is a subfield of machine learning that focuses on training an agent to interact with an environment and learn how to make decisions that maximize some notion of cumulative reward. It is a type of learning that is inspired by the way humans learn by trial and error through interaction with the environment.

One popular algorithm in reinforcement learning is Q-learning, which is a model-free, off-policy algorithm that can learn to make optimal decisions based on the environment's rewards and the agent's actions. The Q-learning algorithm maintains a Q-table, which maps states and actions to their respective Q-values, representing the expected long-term reward of taking that action from that state.

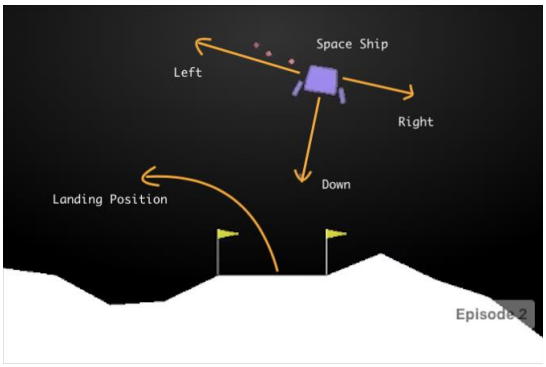
II. OPENAI LUNARLANDER-V2 ENVIRONEMNT

LunarLander-v2 is a reinforcement learning environment developed by OpenAI. The objective of the game is to land a lunar lander spacecraft on a designated landing pad on the surface of the moon. The lander is controlled by applying thrust to its engines in four different directions: left, right, down, and up. The environment is simulated, and the lander is subject to the laws of physics, including gravity and the thrust of the engines.

The game is designed to be a challenging test of an agent's ability to learn and make decisions in a complex, dynamic environment. The lander must navigate through a rocky, uneven terrain and avoid obstacles while also controlling its descent to the landing pad. The lander has a limited amount of fuel, and the game ends when the lander crashes or successfully lands on the pad.

The game is implemented using the OpenAI Gym, a toolkit for developing and comparing reinforcement learning algorithms. The game state is represented by a number of variables including the lander's position, velocity, fuel level, and angle. The agent's actions are represented by the thrust applied to the engines in each direction. The agent receives a reward for each successful landing, and a penalty for crashing or using too much fuel. The agent's goal is to learn a policy, a mapping of states to actions, that maximizes the expected reward over time.

Fig. 1. LunarLander-v2 Environment



The LunarLander-v2 Environment is:

- **Fully Observable**:
  All necessary state information is known observed at every frame.
- **Single Agent**:
  There is no competition nor cooperation
- **Deterministic**:
  There is no stochasticity in the effects of actions or rewards obtained
- **Episodic**:
  Reward is dependent only on current state and action
- **Discrete Action Space**:
  Only 4 Actions: Thrust, Left, Right, Nothing
- **Static**:
  There is no penalty or state change during action deliberation
- **Finite Horizon**:
  Episode terminates after a successful land, crash or set number of steps

- **Mixed Observation Space**:
  - X Distance From Target Site
  - Y Distance From Target Site
  - X Velocity
  - Y Velocity
  - Angle Of Ship
  - Angular Velocity Of Ship
  - Left Leg Is Grounded
  - Right Leg Is Grounded

Finally, after each step a reward is granted.
The total reward of an episode is the sum of the rewards for all steps within that episode.

Reward System:

- Moving from top of screen to landing pad with 0 speed: + 100-140 pts
- Lander moves away from landing pad: - same reward as moving same distance towards the pad
- Crashing/Landing: +/- 100 pts
- Grounding Each Leg: + 10 pts
- Thrusting Main Engine: - 0.3 pts

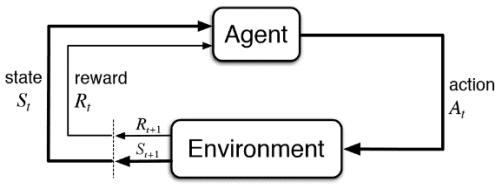
III. Reinforcement Learning, Q-Learning & Their Implementations

Reinforcement learning has been applied to a wide range of problems in various fields such as robotics, game playing, and control problems. One example of its application is in robotics where reinforcement learning has been used to teach robots how to perform tasks such as grasping objects.

There have been many developments in reinforcement learning since its inception. One of the most significant developments is the use of deep neural networks in reinforcement learning. This has led to the development of deep reinforcement learning algorithms such as Deep Q-Networks (DQN) and Asynchronous Advantage Actor-Critic (A3C).

Another development is the use of meta-learning in reinforcement learning. Meta-learning is a technique that allows an agent to learn how to learn. This means that an agent can learn how to solve new tasks more efficiently by leveraging knowledge from previous tasks.

Fig. 2. Overview of Reinforcement Learning



Q-learning is a model-free reinforcement learning algorithm used for learning optimal policies in a Markov Decision Process (MDP). It is based on the idea of estimating the value of an action in a given state by using a Q-value function. The Q-value function represents the expected cumulative reward of taking an action in a given state and following the optimal policy thereafter.

During the learning process, the Q-value function is updated iteratively by the Q-learning algorithm as the agent interacts with the environment. The agent selects actions based on the current estimates of the Q-value function and updates the estimates of the Q-value function based on the observed rewards.

Q-learning is known to be off-policy, meaning that it learns a value function for a target policy that is different from the behavior policy used to select actions. This allows Q-learning to learn from suboptimal behavior, which can be useful in some situations.

Q-learning is a popular algorithm in reinforcement learning and has been successfully applied to a variety of tasks, including game playing, robotics, and control problems.

Fig. 3. Overview of Q-Learning

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{new value (temporal difference target)}} \overbrace{- \underbrace{Q(s_t, a_t)}_{\text{current value}}}^{\text{temporal difference}}$$

The Q-Learning algorithm uses a Q-table of State-Action Values, otherwise known as the Q-values. In the form of games, this can be interpreted as a sort of *cheat sheet*. This table has a row for each state and a column for each action. Each cell contains the estimated Q-value for the corresponding state-action pair.

All Q-Values are set to zero initially. As the agent interacts and gets feedback, the algorithm then iteratively improves these values until they converge with the Optimal Q-Values. It then updates them using the Bellman Equation.

Fig. 4. Bellman Equation

$$V(s) = \max_{a \in A} \left(R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V(s') \right)$$

Q-Learning finds the optimal policy by learning Q-Values for each state-action pair.

- Initially, agent randomly picks actions
- Upon interacting with the environment, learns which actions are better based on rewards obtained. These experiences are used to incrementally update Q-Values
- Equation used to update these values are based off the Bellman Equation
- Two main actions in Q-Learning:
    - **Current Action:** action from which current state that is executed, and whose Q-Value is updated
    - **Target Action:** has highest Q-Value from next state, used to update current action's Q-Value

How Do Estimates Become More Accurate Over Time?

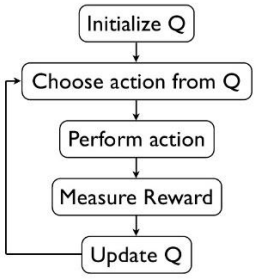
Although we start with arbitrary estimates, at every time-step, the estimates become slightly more accurate as they get updated with real-time observations.

The update formula combines these 3 terms in weighted proportions:

- Reward For Current Action
- Best Estimated Q-Value of Next State-Action
- Estimated Q-Value of Current State-Action

Although 2/3 of these values are estimates, one of them, the reward of the current action, is concrete real data, used to update the Q-Values.

Fig. 5. Q-Learning Process



IV. Deep Q Networks (DQN) & Its Many Variations

Deep Q-Networks (DQN) is a deep reinforcement learning algorithm that combines Q-Learning with deep neural networks (DNNs) to solve problems with high-dimensional state spaces. The agent's brain in Q-learning is the Q-table, but in DQN the agent's brain is a deep neural network.

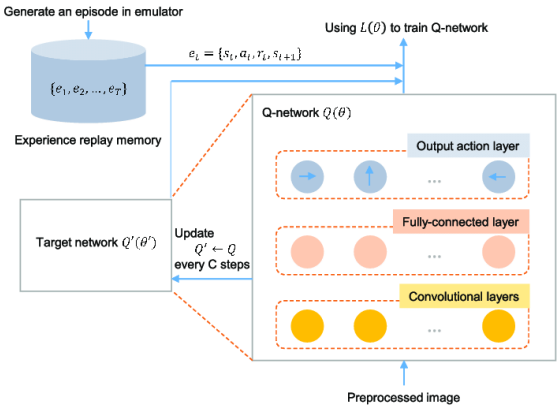
The DQN architecture has two neural nets, the Q network and the Target networks, and a component called Experience Replay. The Q network is the agent that is trained to produce the Optimal State-Action value. Experience Replay interacts with the environment to generate data to train the Q Network.

The update formula for DQN is similar to that of Q-learning but instead of using a table to store all state-action pairs, it uses a deep neural network.

There exists many variations that make use of the base concept and framework of Q-Learning. Some are extensions of the vanilla DQN, such as DDQN or Dueling DQN, while others use Actor-Critic methods, such as Proximal Policy Optimization (PPO).

V. METHODOLOGY

Fig. 6. DQN Architecture

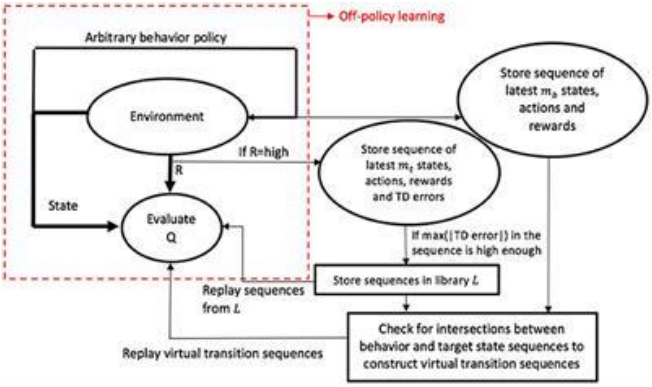


Experience Replay is a technique used in reinforcement learning to improve the efficiency of learning. It involves storing the agent's experiences in a replay buffer and randomly sampling from it to train the agent.

The replay buffer stores the agent's experiences as tuples of (state, action, reward, next state) and is used to break the correlation between consecutive samples and reduce the variance of updates.

By randomly sampling from the replay buffer, the agent can learn from past experiences and avoid forgetting important information.

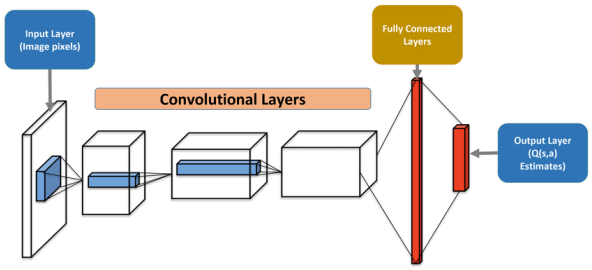
Fig. 7. Experience Replay



There are a few actions that occur in the Experience Replay layer:

- Executes ε-greedy action and receives next state and reward
- stores results data to be used as training data
- select a training batch randomly from replay data (to train the DQN)
- predict Q-Values for all actions using current state from the sample
- Obtain input to target network using next state from the sample (obtain Target Q Value)
- Compute Loss and Back Propagate (for Q Network, Target Network untrained thus no loss
- Repeat for next time-step
- After certain number of time-steps, Q Network weights = Target Network Weights (improved prediction capability)

Fig. 8. Double DQN



The Double DQN is similar to the vanilla DQN network in terms of architecture. The difference is, Double DQN uses two identical network models. One learns during the Experience Replay (like DQN) and the other is a copy of the last episode of the first model.

The Q-Value is computed using the second model. In DQN, Q-Value is calculated by adding reward to the next state

maximum Q-Value. As a result, if the Q-Value computed for a specific state is high the value obtained from the output of the neural network for that specific state becomes higher every time.

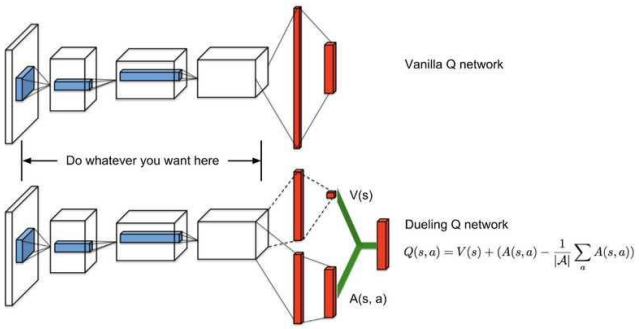
To reduce the difference in output values, the use of a secondary model is implemented. Since the secondary model is a copy of the main model from the last episode, difference between values are lower than the main model and hence is used to obtain the Q-Value

The main difference between Double DQN (DDQN) and vanilla DQN is that in DDQN we use the main value network for action selection and the target network for outputting the Q values.

The main advantage of DDQN over DQN is that it helps reduce the overestimation of Q values and as a consequence helps train faster and have more stable learning.

However, DDQN can also be more computationally expensive than DQN because it requires two networks instead of one.

Fig. 9. Duelling DQN



The difference in a Dueling DQN lies in its architecture and output. It is formed in a way to produce the formula below:

Fig. 10. Duelling DQN Q-Value Equation

$$Q(s.a) = V(s) + A(s.a)$$

In the above formula, V(s) refers to the Value of State S and A refers to the Advantage of performing an action while in state s.

It is important to note that the value of a state is independent of action. This refers to how good an action is in a particular state.

Fig. 11. Q-Value & Advantage Equation

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{n}\sum_{a'} A(s,a')$$

What Is Advantage

The agent could be in a state where all actions produce the same Q-Value, meaning there is no good action in this state.

If the mean of all Advantages is subtracted from each advantage, we get a value very close or equal to 0, where the Q-Value would be the Value that the state possesses.

As a result, Q-Value will not overshoot, and the states that are independent of action would not have a high Q-Value to train on.

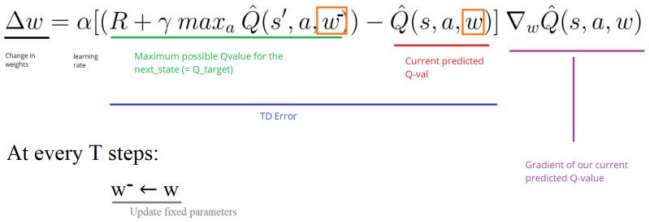
Fig. 12. Duelling DQN with Prioritized Experience Replay

Prioritized Experience Replay (PER) is a type of experience replay in reinforcement learning where we more frequently replay transitions with high expected learning progress, as measured by the magnitude of their temporal-difference (TD) error.

In prior work, experience transitions were uniformly sampled from a replay memory. However, this approach simply replays transitions at the same frequency that they were originally experienced, regardless of their significance.

PER allows agents to learn from transitions sampled with non-uniform probability proportional to their temporal-difference (TD) error. The sumtree data structure is used to store priorities and efficiently sample transitions with high priority.

Fig. 13. Temporal Difference Error (Expanded)



In some instances, the batch chosen from the memory is not an eligible batch, meaning that the experience memories chosen are not significantly important to learn. In that case, the important experience memories have to be chosen for putting in the batch.

How are we able to identify importance in this case? If the Q-Value from the next state is vastly different as compared to the Q-Value from the current state, it means that the importance is high for whether the Q-Value in the next state increases or decreases. This is known as *Temporal Difference Error*.

The formula for Temporal Difference Error is listed as:

Fig. 14. Temporal Difference Error

$$TD = |Q(s,a) - Q(s+1,a)|$$

and the probability of an Experience Memory being chosen would be:

Fig. 15. Experience Memory Importance Function

$$p_i = \frac{(\mathrm{TD}_i + \epsilon)^a}{\sum_k^{memory\ size} (\mathrm{TD}_k + \epsilon)^a}$$

In the formula above, epsilon is a small value to prevent division by 0 and alpha is a random value between 0 and 1. 1 refers to the most important memories and 0 means its random. Hence, p would be the probability that an experience is important, and the batch will then get filled considering experience probabilities.

Since the training of the network occurs stochastically, experiences with high probability would get chosen over and over, causing the network to overfit. In order to curb this issue, we multiply this value to the training loss:

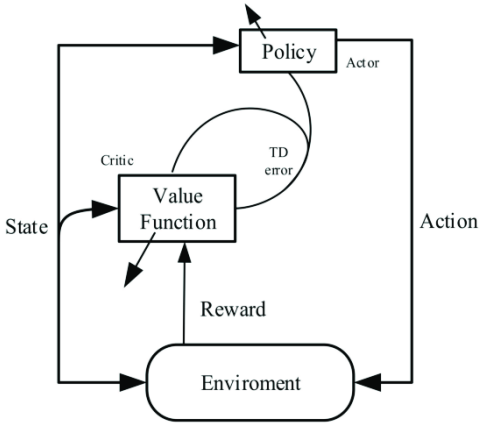
Fig. 16. Importance Penalty

$$Importance = \left(\frac{1}{p_i} \cdot \frac{1}{\text{memory size}}\right)^b$$

VI. Policy Gradients & Actor-Critic Networks

Other than DQNs and its variations, they can be further extended to more complex forms, each catered to specific use cases.

Fig. 17. Deep Deterministic Policy Gradient (DDPG)



Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy.

DDPG is an actor-critic, model-free algorithm based on the deterministic policy gradient that can operate over continuous action spaces2. It combines the actor-critic approach with insights from Deep Q-Networks (DQNs) in particular, the insights that 1) the network is trained off-policy with samples from a replay buffer to minimize correlations between samples, and 2) the network is trained with a target network to give consistent targets during temporal difference backups.

The DDPG algorithm has been shown to work well on a wide range of continuous control problems, including robotic manipulation and locomotion tasks.

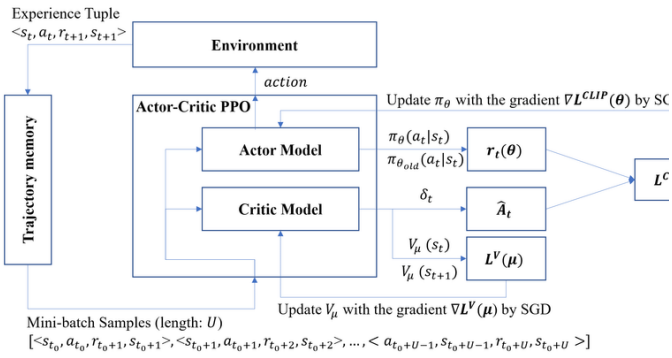
DDPG uses four neural networks:

- Q Network
- Deterministic Policy Network
- Target Q Network
- Target Policy Network

It makes uses of *Actor-Critic Methods*, where the:

- Critic estimates the value function (Q Value or state-value, V)
- while the Actor updates the policy distribution in the direction suggested by the Critic

Fig. 18. Proximal Policy Optimization



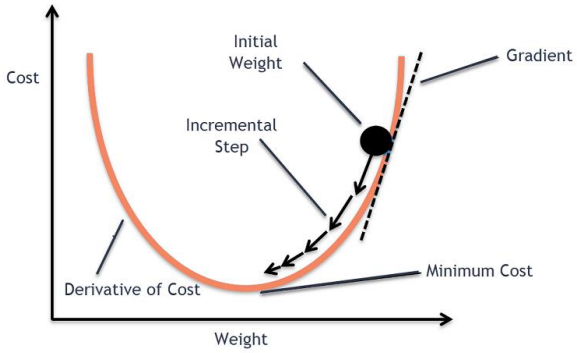
Proximal Policy Optimization (PPO) is a family of model-free reinforcement learning algorithms developed at OpenAI in 20171. PPO algorithms are policy gradient methods, which means that they search the space of policies rather than assigning values to state-action pairs.

PPO proposes a new family of policy gradient methods for reinforcement learning, which alternate between sampling data through interaction with the environment and optimizing a "surrogate" objective function using stochastic gradient ascent.

PPO has become the default reinforcement learning algorithm at OpenAI because of its ease of use and good performance.

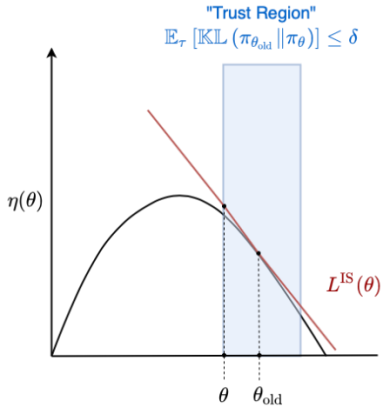
The goal of PPO is to optimize a policy to maximize the rewards. To do this, there are two major optimization techniques: Line Search/Stochastic Gradient Ascent and Trust Region.

Fig. 19. Stochastic Gradient Ascent



Line search picks the steepest direction and move a step. However, if the step size is too small, it takes too long, but if the step size is too large, it is overshot, resulting in the resumption from a locally bad policy, harming performance significantly.

Fig. 20. PPO Trust Region



In the trust region, we determine the maximum step size, before locating the optimal point in the region and search from there.

In PPO, how far the policy can be changed from one iteration t another can be limited through the use of *KL-divergence.* KL-divergence measures the difference between two data distributions, *p* and *q*.

Fig. 21. PPO KL Divergence

$$D_{KL}(P||Q) = \mathbb{E}_x \log \frac{P(x)}{Q(x)}$$

In this scenario, the equation is repurposed to measure the difference between two policies. New policies should not be too different from the current one.

How can we limit policy change? We can make use of a lower bound function ***M***:

Fig. 22. Lower Bound Function *M*

$$M = L(\theta) - C \cdot \overline{KL}$$

Where L(θ) is

$$\hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t\right]$$

and the second terms represent KL-divergence

L, in this case, represents the expected advantage function. By recalibrating through the use of the probability ratio between the new and old policy, we are able to come up with this estimated value. To reduce variance of this estimation, we make use of the advantage function instead of the expected reward.

With reference to the second term in *M*, in Figure 15, we will explore it in further detail. The proof is rather extensive, and we are able to obtain this function as a form of final derivative.

Fig. 23. KL Divergence in Lower Bound Function *M*

$$\eta(\tilde{\pi}) \geq L_\pi(\tilde{\pi}) - CD_{\text{KL}}^{\max}(\pi, \tilde{\pi}), \quad \text{where } C = \frac{4\epsilon\gamma}{(1-\gamma)^2}.$$

$$D_{\text{KL}}^{\max}(\pi, \tilde{\pi}) = \max_s D_{\text{KL}}(\pi(\cdot|s) \parallel \tilde{\pi}(\cdot$$

$$\epsilon = \max_{s,a} |A_\pi(s,a)|$$

As underlined in red, the second term of the function *M* represents the maximum KL-divergence. However, due to the level of complexity to find this term, we will use the mean of KL-divergence instead.

We can understand these functions mathematically, but what do they truly represent? To understand this, we have to take a look at *L*, as well as what both terms purpose is in the lower bound function *M*.

Fig. 24. Summarized Objective Of Function *M*

$$\begin{aligned} &\underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t\right] \\ &\text{subject to} \quad \hat{\mathbb{E}}_t[\text{KL}[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]] \leq \delta. \end{aligned}$$

or

$$\underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t\right] - \beta\hat{\mathbb{E}}_t[\text{KL}[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]]$$

L is the approximation of the advantage function at the current policy. However, as it shifts away from the old policy, it gets less and less accurate. This inaccuracy possesses an upper bound, and is also the second term in *M*.

After considering the upper bound of this error, we can guarantee that the calculated optimal policy within the trust region is always better than the old policy. If the policy is outside the trust region, even the calculated value may be better but the accuracy can be too off and cannot be trusted. These results can be seen as summarized in Figure 17.

Although we are able to derive the same optimal policy from both equations, delta represents a miniscule value of a threshold. Hence, we set them as hyperparameters that are open to tuning.

Fig. 25. Approximated Quadratic Equation

$$\begin{aligned} &\underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t\right] \quad \mathcal{L} \\ &\text{subject to} \quad \hat{\mathbb{E}}_t[\text{KL}[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]] \leq \delta. \end{aligned}$$

By using the Taylor series, we expand the terms up to their second order.

By approximating the objective and constraint, our objective becomes as follows:

Fig. 26. Objective Function

$$\begin{aligned} \theta_{k+1} = \arg\max_\theta \; & g^T(\theta - \theta_k) \\ \text{s.t.} \; & \frac{1}{2}(\theta - \theta_k)^T F\,(\theta - \theta_k) \leq \delta \end{aligned}$$

Fig. 27. Solution To Objective Function

$$\theta_{k+1} = \theta_k + \underbrace{\sqrt{\frac{2\delta}{g^T F^{-1} g}} F^{-1} g}_{\text{natural policy gradient}}$$

So we have found a method to optimize this equation. However, as with all things in the AI & Machine Learning domain, or anything to do with data for that matter, there is the curse of computational cost.

Fig. 28. Computationally Expensive Operations

$$F = \nabla^2 f = \begin{pmatrix} \frac{\partial^2 f_1}{\partial x_1^2} & \frac{\partial^2 f_1}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f_1}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f_2}{\partial x_1^2} & \frac{\partial^2 f_2}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f_2}{\partial x_1 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f_m}{\partial x_1^2} & \frac{\partial^2 f_m}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f_m}{\partial x_1 \partial x_n} \end{pmatrix}$$

The aim here is to be able to optimize the trust region policy in a high dimensional space. The solution proposed above requires the second order derivative along with its inverse, which is quite an expensive operation.

So how do we solve this? There are two approaches we can take. Firstly, we can approximate calculations where the second order derivative and its inverse are involved, thus lowering the complexity of the computational processes. Secondly, we can make the first order derivative solution closer to the second order derivative solution by adding soft constraints.

Both the Trust Region Policy Optimization as well as the Actor Critic using Kronecker-Factored Trust Region (ACKTR) adopts the first approach, while PPO leans towards the latter.

By adding a soft constraint to the objective function, the optimization will in turn have increased insurance that we are optimizing within the trust region.

Why use the former approach? Although such a method may result in bad decisions once in a while, there exists a good balance between performance and speed of optimization. Obtaining reasonable performance with the most simplicity and low computational cost, is one of the values at the center of machine learning.

This results in a decreased chance of a bad decision. There are a few ways of implementing this, such as incorporating an adaptive KL-divergence penalty, or clipping the objective function.

Fig. 29. Adaptive KL Penalty Objective Function

$$\underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \hat{A}_t \right] - \beta \hat{\mathbb{E}}_t [\text{KL}[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]]$$

Fig. 30. Clipped Objective Function

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \underset{\tau \sim \pi_k}{\mathrm{E}} \left[\sum_{t=0}^{T} \left[\min(r_t(\theta) \hat{A}_t^{\pi_k}, \text{clip}\,(r_t(\theta), 1-\epsilon, 1+\epsilon)\, \hat{A}_t^{\pi_k}) \right] \right]$$

VII. Experimentation & Results

We have explored a large plethora of methods and implementations of Q-Learning frameworks and models, but how do they perform in practice, even if they look good on paper. Theoretically, majority of models are backed by strong evidence, with certain models even catered to specific use cases. However, these results do not always reflect what they should in practice. So how do we go about evaluating the performance of these models?

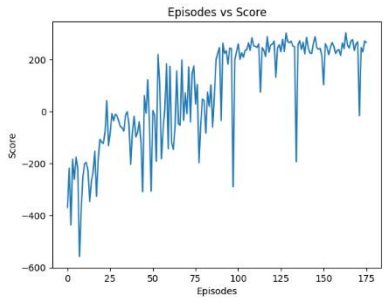
It is clear that the objective goal here is to successfully land the lunar lander in its environment. OpenAI has provided some benchmarks to allow for testing. The main identifier in this case, is where a score of at least 200 or more is considered to be a viable solution. This however, is not enough to provide a comprehensive evaluation of the models performance.

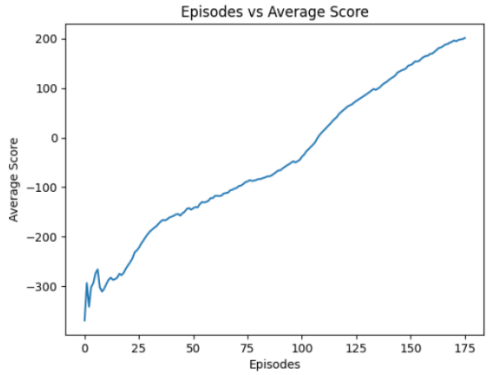
There are multiple additional ways to assess and evaluate the performance of our RL models, namely:

- score
- average score
- success/crash rate
- fuel consumption
- number of episodes required

The LunarLander-v2 environment is considered to be a rather preliminary task to solve, and hence the score, average score and number of episodes required to solve the environment is arguably the most important metrics to look out for.

Fig. 31. DQN Episodes Vs Score/Average Score

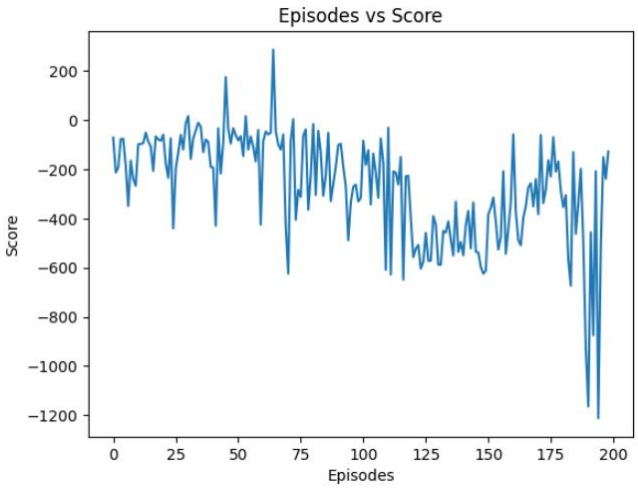


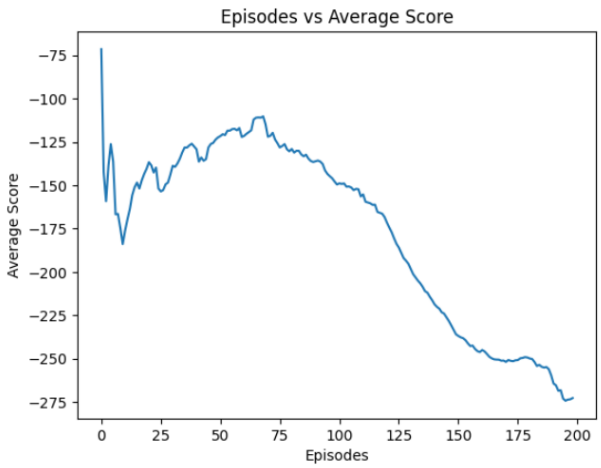


By plotting line graphs of episodes by score and average score, we are able to visualize the general performance of the model. This is important as sometimes the model flukes an episode and skyrockets past the score threshold, making it seem as though the model is performing well, when in reality it was a fluke.

Here, we can see that the vanilla DQN actually performed quite well, solving the environment in only 175 episodes, reaching a score well above 200. Despite the score plot being slightly volatile, this is fine as there is still a general increasing trend, showing that the model is learning instead of fluking the results.

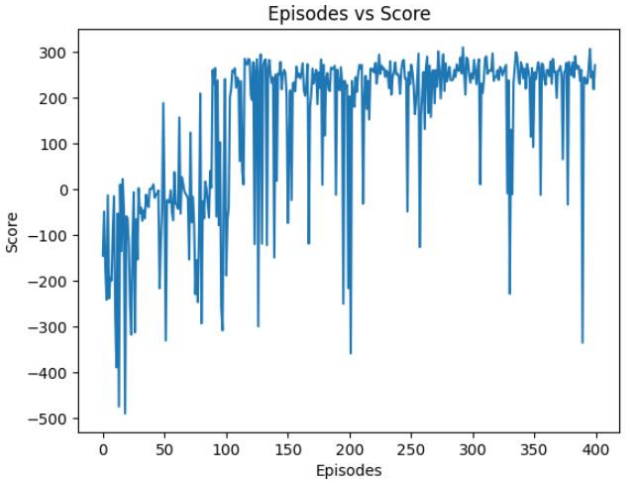
Fig. 32. Double DQN Episodes Vs Score/Average Score

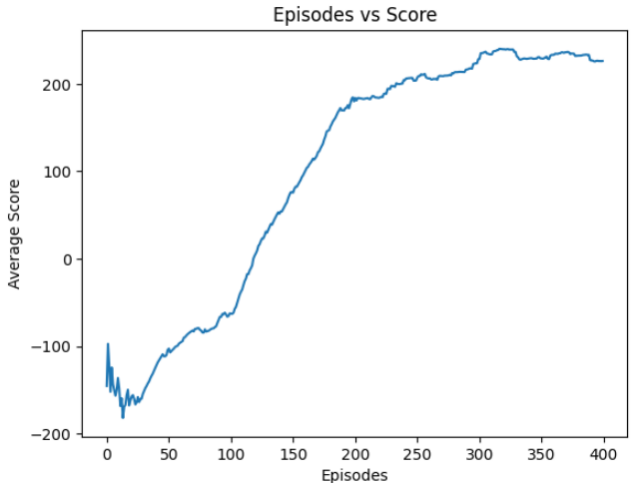




On the contrary, by taking a look at the results of tour Double DQN, we can see that it performed terribly. We can see that it spiked in one episode and hit a score of 200, however, the average score plot shows that it was a fluke, as it shows a very clear and steady decreasing trend in score, with the scale being completely in the negatives. By the end of the training, it hits an impressively poor score of -275.

Fig. 33. Dueling DQN Episodes Vs Score/Average Score



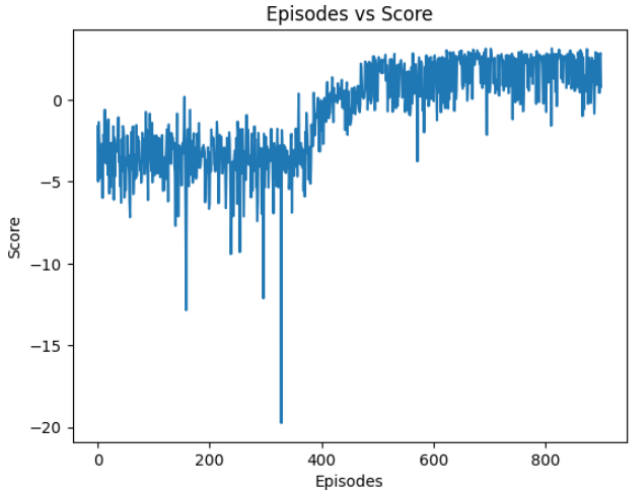


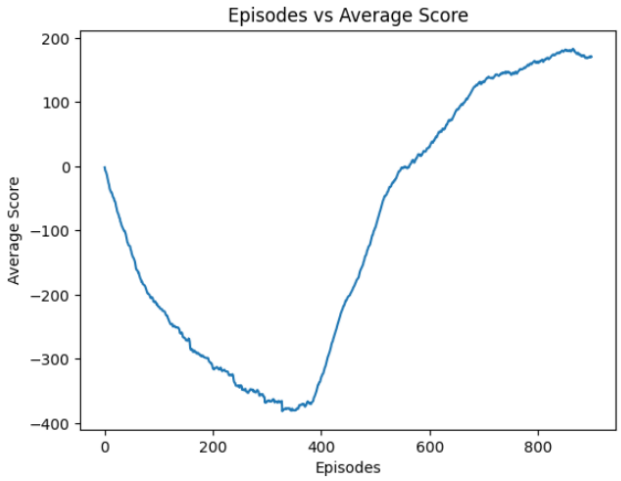
The dueling DQN performed significantly better than the double DQN. We can observe from the average score plot that it learns very quickly, and easily hits a score of 200.

The raw score plot however, is extremely volatile. While it is able to hit scores at high at 300 quite often, it also hits scores at low as -500. Additionally, it is clear to see that the amplitude of the graph is very large, with a large disparity between each episode.

Despite this, the performance can still be considered good, as it takese a very short time to train, as compared to the vanilla DQN and double DQN.

Fig. 34. Dueling DQN Episodes Vs Score/Average Score

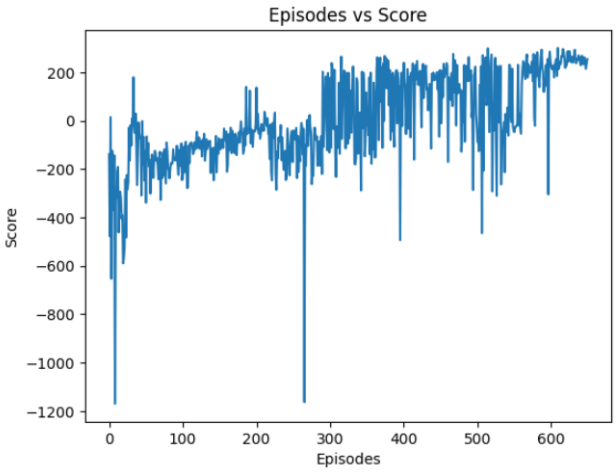


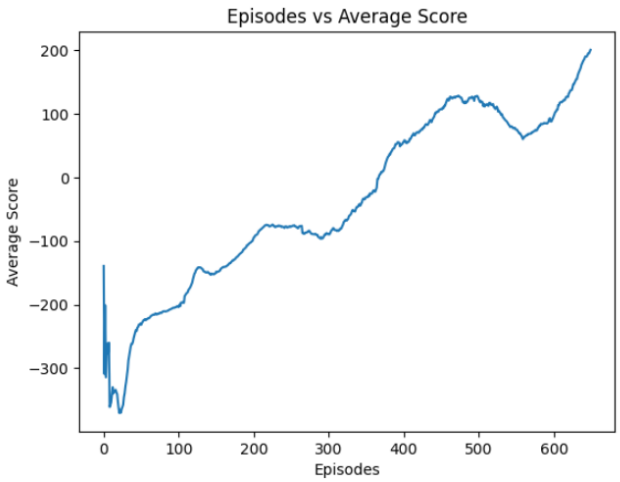


It is clear that the Dueling DQN with Prioritized Experience Replay is much less volatile in training as compared to its vanilla counterpart. For close to 400 episodes, the model learns negatively, to a point as low as almost -400. However, it quickly picks up the pace and skyrockets to a score of 200. It is worth noting that it took a very long time, of 800 episodes to reach an average score of 200.

Fig. 35. Deep Deterministic Policy Gradient

Episodes Vs Score/Average Score





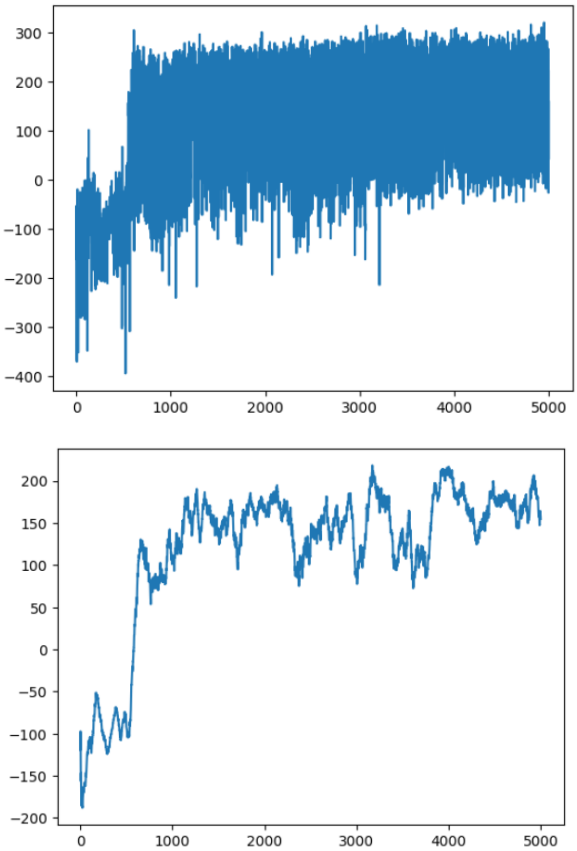
At first glance, the Deep Deterministic Policy Gradient looks very promising, with an average score plot creating a very smooth and relatively consistent upward trend.

On the other hand, when taking a look at the raw score plot, we can see that its not very consistent, and in certain instances hit a low score of -1200. Furthermore, the model starts off training negatively, and reaches a very low score before gradually climbing back up.

This shows that the model is quite volatile and might have gotten lucky in reaching an average score of 200 in 600 episodes.

Fig. 36. Proximal Policy Optimization

Episodes Vs Score/Average Score



Plotting the raw score plot gives us a rather jarring visualization. While it appears as if the model is very volatile during training, we can see that the peaks and troughs of the model are actually generally in the positive region, and frequently hits close to a score of 300.

When observing the average score plot instead, we can see that the model is quite stable, apart from ther sudden huge jump in score at around the 1000th episode mark.

The model was trained for 5000 episodes; however, this was actually the fastest model to train. It is well known that PPO is able to train very fast. Thus, although it may look like the model takes very long to train and takes many episodes to produce results, the model is far from incompetent.

VIII. CONCLUSION

In this project, we have explored multiple different models and hyperparameters to attempt to solve OpenAI's LunarLander-v2 environment.

There are many different models with many different use cases. For example, some use Q-learning frameworks while others use actor-critic methodologies, and can be used accordingly for complex, high variance, or efficiency-related tasks.

In our instance, although a model such as Deep Deterministic Policy Gradient (DDPG) is able to achieve an extremely high score, this is not necessary for such a simple and preliminary environment such as the LunarLander-v2, where there are only 4 discrete actions with a simple objective.

For more detailed explanations behind the workings and concepts of the various Reinforcement Learning models, a full report is written within the jupyter notebook.

IX. REFERENCES

https://neptune.ai/blog/best-reinforcement-learning-tutorials-examples-projects-and-courses

An Introduction to ReinforcementLearning.jl: Design, Implementations and Thoughts (juliareinforcementlearning.org)

GitHub - rll/rllab: rllab is a framework for developing and evaluating reinforcement learning algorithms, fully compatible with OpenAI Gym.

https://markelsanz14.medium.com/introduction-to-reinforcement-learning-part-3-q-learning-with-neural-networks-algorithm-dqn-1e22ee928ecd

Deep Reinforcement learning: DQN, Double DQN, Dueling DQN, Noisy DQN and DQN with Prioritized Experience Replay | by Parsa Heidary Moghadam | Medium

Reinforcement Learning Explained Visually (Part 5): Deep Q Networks, step-by-step | by Ketan Doshi | Towards Data Science

comparison - What is the difference between Q-learning, Deep Q-learning and Deep Q-network? - Artificial Intelligence Stack Exchange

[1509.06461] Deep Reinforcement Learning with Double Q-learning (arxiv.org)

comparison - What exactly is the advantage of double DQN over DQN? - Artificial Intelligence Stack Exchange

Improvements in Deep Q Learning: Dueling Double DQN, Prioritized Experience Replay, and fixed… (freecodecamp.org)

Deep Reinforcement learning: DQN, Double DQN, Dueling DQN, Noisy DQN and DQN with Prioritized Experience Replay | by Parsa Heidary Moghadam | Medium

Deep Deterministic Policy Gradient — Spinning Up documentation (openai.com)

DDPG Explained | Papers With Code

[1509.02971] Continuous control with deep reinforcement learning (arxiv.org)

[1707.06347] Proximal Policy Optimization Algorithms (arxiv.org)

Proximal Policy Optimization (openai.com)

Prioritized Experience Replay Explained | Papers With Code

[1511.05952] Prioritized Experience Replay (arxiv.org)

[2209.00532] Actor Prioritized Experience Replay (arxiv.org)

SumTree for Prioritized Experience Replay (PER) explained - Amir Masoud Sefidian - Sefidian Academy

PyLessons

Summary: Prioritized Experience Replay | by Zac Wellmer | Arxiv Bytes | Medium

Prioritized Experience Replay - MyScienceWork

What is "experience replay" and what are its benefits? (stackexchange.com)