# Solving Control Problems using Reinforcement Learning

Nicholas Falzon
nicholas.falzon.21@um.edu.mt

## 1 Introduction

### 1.1 Reinforcement Learning

Reinforcement learning is an area of Machine Learning which is concerned around taking suitable action to maximize reward in a particular situation. It is employed by various software and machines to find the best possible behavior or path it should take in a specific situation. It is an autonomous, self-teaching system that essentially learns by trial and error. It performs actions with the aim of maximizing rewards, or in other words, it is learning by doing in order to achieve the best outcomes [2].

Reinforcement learning uses algorithms that learn from outcomes and decide which action to take next. After each action, the algorithm receives feedback that helps it determine whether the choice it made was correct, neutral or incorrect. It is a good technique to use for automated systems that have to make a lot of small decisions without human guidance [2].

### 1.2 Reinforcement Learning vs Supervised Learning vs Unsupervised Learning

Reinforcement learning differs from supervised learning because in supervised learning, the model is trained using training data which has the target values embedded within it, so the model is trained knowing the correct answer which it should ultimately provide. Conversely, in reinforcement learning, there is no target provided but the agent decides how to perform a given task. Therefore, in the absence of a training dataset, it is bound to learn from its experience [2].

Reinforcement learning also differs from unsupervised learning since unsupervised methods are trained on unlabelled data while reinforcement learning is trained on no predefined data at all. Additionally, unsupervised learning works by understanding patterns in the input data while reinforcement learning follows a trail and error method [9].

### 1.3 Brief Introduction to Value Based, Policy Based and Actor Critic Models

The primary goal in RL is to enable an agent to learn an optimal policy denoted $\pi^*$ that maximizes the cumulative reward over a sequence of actions. The agent interacts with an environment by taking actions, receiving feedback in the form of rewards or penalties, and updating its knowledge based on this feedback. There are three types of models which are generally used to achieve this goal. These RL models are classified by considering which component is optimized by the algorithm — the value function or the policy.

#### 1.3.1 Value Based Methods.

In value based methods, the focus is on learning the optimal value function, denoted $Q^*$ or $V^*$. The value function estimates expected cumulative future rewards for being in a given state and following the current policy thereafter [17]. In this case, the policy is generated implicitly since it is not directly updated. In essence, finding an optimal value function leads to having an optimal policy since the optimal policy can be obtained directly from the optimal value function [16], as can be seen below.

$$\pi^*(s) = \underset{a}{argmax} Q^\pi(s, a)$$

#### 1.3.2 Policy Based Methods.

On the other hand, in policy based methods the policy $\pi$ is optimized directly instead of a value function. This means that instead of learning the expected sum of rewards given a state and an action, the function that maps state to action is learned directly. In other words, in policy based methods, the policy itself is manipulated in order to find the optimal policy [1].

#### 1.3.3 Actor Critic Methods.

Finally, actor critic methods employ a combination of both value based and policy based methods. In actor critic methods, the policy is referred to as the actor which proposes a set of possible actions given a state, and the estimated value function is referred to as the critic, which evaluates actions taken by the actor based on the given policy [20].

These models and the differences between them will be elaborated on further in the following sections of the report.

## 2 Background

In RL, an agent is built to in order to make smart decisions. To make these intelligent decisions, the agent learns from the environment by interacting with it through trial and error and receiving rewards (positive or negative) as unique feedback. Its goal is to maximize its expected cumulative reward [7].

The agent's decision making process is called the policy $\pi$. Given a state, a policy will output an action or a probability distribution over actions. That is, given an observation of the environment, a policy will provide an action (or multiple probabilities for each action) that the agent should take. The

objective is always to find an optimal policy $\pi^*$, a policy that leads to the best expected cumulative reward [7].

In order to find this optimal policy, there are a number of different RL methods which can be employed.

### 2.1 Value Based Methods

In value-based methods, we learn a value function that maps a state to the expected value of being at that state. The value of a state is the expected discounted return the agent can get if it starts at that state and then acts according to our policy [6].

Since the policy is not trained/learned, its behavior is specified beforehand. For instance, if the policy should, given the value function, take actions that always lead to the biggest reward, a Greedy Policy should be used [6].

#### 2.1.1 State-Value Function.

$$V_\pi(s) = E_\pi[G_t \mid S_t = s]$$

For each state, the state-value function outputs the expected return if the agent starts at that state and then follows the policy for all future timesteps [6].

#### 2.1.2 Action-Value Function.

$$Q_\pi(s, a) = E_\pi[G_t \mid S_t = s, A_t = a]$$

In the action-value function, for each state and action pair, the action-value function outputs the expected return if the agent starts in that state, takes that action, and then follows the policy forever after [6].

#### 2.1.3 Q-Learning.

The Q-learning algorithm uses a Q-table of state-action Values (referred to as Q-values). This Q-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 [4].

Initially all the Q-values are set to zero and as the agent interacts with the environment and gets feedback, the algorithm iteratively improves these Q-values until they converge to the Optimal Q-values [4]. These values are updated using the Bellman equation:

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma max_a Q(S', a) - Q(S, A)]$$

The Bellman equation is used to iteratively approximate the optimal Q-value, which is the maximum future reward.

### 2.2 Deep Q-Networks

DQNs are variations of vanilla Q-Learning which uses a neural network to map input states to (action, Q-value) pairs, unlike the vanilla Q-Learning algorithm which uses a Q-table to map each state-action pair to its corresponding Q-value [21]. The primary advantage of replacing the Q-table with a neural network is that vanilla Q-Learning becomes computationally expensive and requires a lot of memory to store the Q-values when dealing with large state spaces, as the size of the Q-table grows exponentially with the number of states and actions. On the other hand, DQNs use a deep neural network to approximate the Q-values for each (state, action) pair, which makes it feasible to handle large state spaces [13].

One of the critical components of a DQN is experience replay. This technique stores past experiences (state, action, reward, next state) in a replay buffer. During training, random batches of experiences are sampled from this buffer, breaking temporal correlations and making the learning process more stable [8].

### 2.3 Policy Based Methods

In value based methods, the policy $\pi$ only exists because of the action-value estimates since the policy is just a function (for instance, greedy policy) that will select the action with the highest value given a state.

With policy based methods, the policy is optimized directly without having an intermediate step of learning a value function.

The major advantage which policy based methods have over value based is unlike value based methods, policy based methods can learn a stochastic policy. This means the agent can explore the state space without always taking the same trajectory, eliminating the need to manually implement an exploration/exploitation trade-off [5].

Furthermore, policy based methods can handle perceptual aliasing, where two states appear the same but require different actions. Under a value based policy, the policy may get stuck in a loop, resulting in the agent spending a lot of time training before finding the correct path. An optimal stochastic policy, however, will randomly move in these states, preventing the agent from getting stuck [5].

Policy based methods are more effective in high-dimensional action spaces and continuous action spaces. Instead of assigning a score for each possible action at each time step given the current state (like DQN), policy-based methods output a probability distribution over actions, making them more suitable for situations with a large number of potential actions [5].

Nevertheless, policy based methods often converge to a local optimum instead of a global optimum due to the nature of the optimization process. Moreover, they can also be less efficient in terms of training speed, which can result in a longer training time compared to value based methods. They also exhibit high variance in their performance, meaning that the quality of the policy produced can vary significantly across different runs of the algorithm, even with the same parameters [5].

### 2.4 Actor-Critic Methods

Actor-Critic methods are a type of RL algorithm that combines the strengths of both policy based and value based

methods. It consists of two components: the Actor, which decides what action to take, and the Critic, which evaluates the action taken by the Actor and provides feedback on how to adjust the policy [10].

The Actor-Critic method is beneficial in environments with high-dimensional and stochastic continuous action spaces, and it's particularly useful for learning stochastic policies. The Actor's learning is based on the policy gradient approach, while the Critic evaluates the action produced by the Actor by computing the value function [10].

There are two types of policies an actor critic model can learn: deterministic and stochastic. A deterministic policy maps each state to a single action with certainty, meaning the agent will always take the same action given a state. This is suitable for tasks where the same action should be taken for the same state every time [3].

On the other hand, a stochastic policy maps each state to a probability distribution over actions. Given a state, the agent will choose an action randomly based on the probability distribution. This allows the agent to capture the uncertainty in the environment and adapt its actions based on the probability of success [3].

Deterministic policies provide a straightforward mapping from states to actions, making them easier to understand and implement. However, they can lead to poor exploration of the action space, especially in large and complex environments. They can get stuck in suboptimal solutions because they would keep choosing the same action for a given state. Stochastic policies overcome this potential issue since they introduce randomness in the action selection process by allowing the agent to explore different actions and potentially discover better policies [5].

### 2.5 Actor-Critic Algorithms

In Experiment 2, I decided to use SAC (stochastic policy) and DDPG (deterministic policy), hence why these are the two algorithms I shall be explaining.

#### 2.5.1 Soft Actor Critic (SAC).

Soft Actor Critic (SAC) is an algorithm that optimizes a stochastic policy in an off-policy way, forming a bridge between stochastic policy optimization and DDPG-style approaches [15].

A central feature of SAC is entropy regularization. The policy is trained to maximize a trade-off between expected return and entropy, which is a measure of randomness in the policy. This has a close connection to the exploration-exploitation trade-off: increasing entropy results in more exploration, which can accelerate learning later on. It can also prevent the policy from prematurely converging to a bad local optimum [15].

SAC trains a stochastic policy with entropy regularization, and explores in an on-policy way. The entropy regularization coefficient $\alpha$ explicitly controls the explore-exploit tradeoff, with higher $\alpha$ corresponding to more exploration, and lower $\alpha$ corresponding to more exploitation. The right coefficient (the one which leads to the stablest / highest-reward learning) may vary from environment to environment, and could require careful tuning [15].

These are the key model training update equations for SAC:

$$y(r, s', d) = r + \gamma(1-d)\left(\min_{i=1,2} Q_{\phi_{targ,i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \tilde{a}' \sim \pi_\theta(\cdot|s')$$

This equation is responsible for updating the targets for the Q functions. It is implemented in the code at line 16 of Figure 1.

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_{\phi_i}(s,a) - y(r,s',d))^2$$

for i = 1,2

This equation updates the Q-functions by one step of gradient descent. It is implemented in the code at line 20 of Figure 1.

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s)|s)\right)$$

where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable with respect to $\theta$ using the reparameterization trick.

This equation updates the policy by one step of gradient descent. The way the policy is optimized makes use of the reparameterization trick, in which a sample from $\pi_\theta(\cdot|s)$ is drawn by computing a deterministic function of state, policy parameters, and independent noise [15]. This trick is used to make sure that sampling from the policy is a differentiable process so that there are no problems in backpropagating the errors [12]. It is implemented in the code at line 32 of Figure 1.

$$\phi_{targ,i} \leftarrow \rho \phi_{targ,i} + (1-\rho)\phi_i$$

for i = 1,2

This equation updates the target networks. It is implemented in the code at lines 38-45 of Figure 1.

#### 2.5.2 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 [14].

This approach is closely connected to Q-learning, and is motivated the same way: if you know the optimal action-value function $Q^*(s,a)$, then in any given state, the optimal

```
1 def update(step):
2         states, actions, rewards, dones, next_states = replay_memory.sample_batch()
3 
4         #Critic Loss
5         action_values1 = q_net1(states, actions)
6         action_values2 = q_net2(states, actions)
7 
8         with torch.no_grad():
9             target_actions, target_log_probs = actor(next_states)
10 
11            next_action_values = torch.min(
12                q_net1_target(next_states, target_actions),
13                q_net2_target(next_states, target_actions)
14            )
15            next_action_values[dones] = 0.0
16            target = rewards + GAMMA * (next_action_values - ALPHA * target_log_probs)
17 
18        Qloss1 = F.smooth_l1_loss(target, action_values1)
19        Qloss2 = F.smooth_l1_loss(target, action_values2)
20        Qloss_total = (Qloss1 + Qloss2)
21 
22        q_net_optimizer.zero_grad()
23        Qloss_total.backward()
24        q_net_optimizer.step()
25 
26        #Policy Loss
27        actions, log_probs = actor(states)
28        action_values = torch.min(
29            q_net1(states, actions),
30            q_net2(states, action
31        )
32        policy_loss = (ALPHA * log_probs - action_values).mean()
33 
34        policy_optimizer.zero_grad()
35        policy_loss.backward()
36        policy_optimizer.step()
37 
38        for target_param, param in zip(q_net1_target.parameters(), q_net1.parameters()):
39            target_param.data.copy_(param.data * TAU + target_param.data * (1.0 - TAU))
40 
41        for target_param, param in zip(q_net2_target.parameters(), q_net2.parameters()):
42            target_param.data.copy_(param.data * TAU + target_param.data * (1.0 - TAU))
43 
44        for target_param, param in zip(actor_target.parameters(), actor.parameters()):
45            target_param.data.copy_(param.data * TAU + target_param.data * (1.0 - TAU))
```

**Figure 1.** SAC Training Update Code

action $a^*(s)$ can be found by solving $a^*(s) = \underset{a}{argmax} Q^*(s,a)$ [14], as was mentioned earlier on in this report.

DDPG trains a deterministic policy in an off-policy way. Because the policy is deterministic, if the agent were to explore on-policy, in the beginning it would probably not try a wide enough variety of actions to find useful learning signals. To make DDPG policies explore better, noise is added to their actions at training time. The authors of the original DDPG paper recommended time-correlated OU noise, but more recent results suggest that uncorrelated, mean-zero Gaussian noise works perfectly well. Since the latter is simpler, it is preferred [14].

These are the key model training update equations for DDPG:

$$y(r, s', d) = r + \gamma(1-d) Q_{\phi_{targ}}(s', \mu_{\theta_{targ}}(s'))$$

This equation computes the targets. It is implemented in the code at line 11 of Figure 2.

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s,a) - y(r,s',d))^2$$

This equation updates the Q-function by one step of gradient descent. It is implemented in the code at line 12 of Figure 2.

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

This equation updates the policy by one step of gradient descent. It is implemented in the code at line 15 of Figure 2.

$$\phi_{targ} \leftarrow \rho \phi_{targ} + (1-\rho)\phi$$
$$\theta_{targ} \leftarrow \rho \theta_{targ} + (1-\rho)\theta$$

These equations are used to update the target networks. It is implemented in the code at lines 27-31 of Figure 2.

```
1 def update():
2         states, actions, rewards, dones, next_states = replay_memory.sample_batch()
3 
4         ou_noise.reset()
5         # Critic Loss
6         Qvals = critic(states, actions)
7         with torch.no_grad():
8             actions_ = actor_target(next_states)
9             Qvals_ = critic_target(next_states, actions_)
10            Qvals_[dones] = 0.0
11            target = rewards + GAMMA * Qvals_
12        critic_loss = F.smooth_l1_loss(target, Qvals)
13 
14        # Actor Loss
15        actor_loss = -critic(states, actor(states)).mean()
16 
17        # Update Networks
18        actor_optimizer.zero_grad()
19        actor_loss.backward()
20        actor_optimizer.step()
21 
22        critic_optimizer.zero_grad()
23        critic_loss.backward()
24        critic_optimizer.step()
25 
26        # Update Target Networks
27        for target_param, param in zip(actor_target.parameters(), actor.parameters()):
28            target_param.data.copy_(param.data * TAU + target_param.data * (1.0 - TAU))
29 
30        for target_param, param in zip(critic_target.parameters(), critic.parameters()):
31            target_param.data.copy_(param.data * TAU + target_param.data * (1.0 - TAU))
```

**Figure 2.** DDPG Training Update Code

# 3 Methodology

## 3.1 Experiment 1

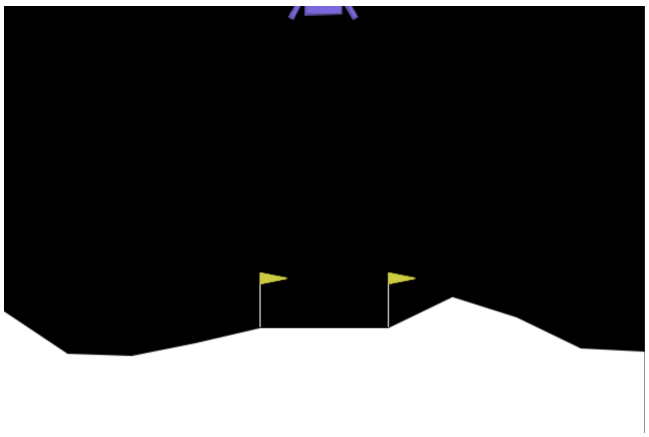

**Figure 3.** Lunar-Lander Start State

### 3.1.1 Environment.

The environment is a rocket trajectory optimization problem where the lander starts at the top center of the viewport, (as can be seen in Figure 3) with a random initial force applied to its center of mass. The goal is for the lander to land

between the two yellow flags in as smooth of a manner as possible, utilizing the engines only when needed.

The LunarLanader environment is [19]:

- Fully Observable: All necessary state information is known observed at every frame.
- Single Agent: There is no competition or cooperation.
- Deterministic: There is no stochasticity in the effects of actions or the rewards obtained.
- Episodic: The reward is dependent only on the current state and action.
- Static: There is no penalty or state change during action deliberation.
- Finite Horizon: The episode terminates after the lander crashes (the lander's body gets in contact with the surface), the lander gets outside of the viewport (x coordinate is greater than 1) or the lander is not awake (does not move and does not collide with any other body) [11]

#### 3.1.2 Observation Space.

Each observation is an 8-dimensional vector containing:

1. Lander X coordinate (Continuous)
2. Lander Y coordinate (Continuous)
3. X velocity (Continuous)
4. Y velocity (Continuous)
5. Angle of ship (Continuous)
6. Angular velocity of ship (Continuous)
7. If left leg is grounded (Boolean)
8. If right leg is grounded (Boolean)

#### 3.1.3 Action Space.

In this version of the environment, there are four discrete actions which can be taken:

- Do nothing.
- Fire left orientation engine.
- Fire right orientation engine.
- Fire main engine.

The main engine is located at the bottom of the lander and when fired will reduce the speed at which the rocket descends. Firing the left orientation engine will rotate the rocket to the right in a clockwise motion, while firing the right orientation engine will rotate the rocket to the left in an anti-clockwise motion.

#### 3.1.4 Rewards.

The reward received after each step is determined by the following:

- The reward is increased/decreased the closer/further the lander is to the landing zone (located between the two yellow flags).
- The reward is increased/decreased the slower/faster the lander is moving.
- The reward is decreased the more the lander is tilted (angle not horizontal).
- The rewards is increased by 10 points for each leg that is in contact with the ground.
- The reward is decreased by 0.03 points for each frame in which one of the side engines are firing.
- The reward is decreased by 0.3 points for each frame that the main engine is firing.

Additionally, each episode receives an additional reward of -100 for crashing the lander or +100 points for landing safely.

I considered the problem to be solved if the last 50 episodes obtain an average score of 195 or larger.

#### 3.1.5 Libraries Used.

For both experiments, I only made use of the code from the various Jupyter Notebooks which were provided on the VLE. I altered the code to work with the LunarLander-v2 environment instead of whichever environment was declared originally. I also changed a few hyperparameters as I saw fit. Therefore, Pytorch was the only library which was used in order to implement neural networks for the algorithms.

### 3.2 Experiment 2

The environment for this experiment is nearly identical to that of experiment 1, with the observation space, rewards, start state and episode termination cases all being the same. The difference between the two experiments lies in the action space. The action space of experiment 1 is discrete while the action space of experiment 2 is continuous and is presented in the form of: Box(-1, +1, (2,), dtype=np.float32) [11]. This means that the action space will be represented as a vector containing 2 values between -1 and +1. The first value determines the throttle of the main engine, while the second specifies the throttle of the side boosters.

Given an action ([main, side]), the main engine will be turned off completely if main < 0 and the throttle scales affinely from 50% to 100% for 0 <= main <= 1 (the main engine does not work with less than 50% power). Similarly, if -0.5 < side < 0.5, the side boosters will not fire at all. If side < -0.5, the left booster will fire, and if side > 0.5, the right booster will fire. Again, the throttle scales affinely from 50% to 100% between -1 and -0.5 and 0.5 and 1, respectively [11].

Because of this continuous action space, value based methods such as a DQN struggle to solve this problem because of the infinite number of potential actions. Rather, policy based or actor critic methods should be used to solve it.

## 4 Results and Discussion

### 4.1 Experiment 1

From the four variations of DQNs tested, the standard DQN with priority experience replay (Figure 6) was the model which managed to solve the problem in the least number of episodes, doing so in around 650.

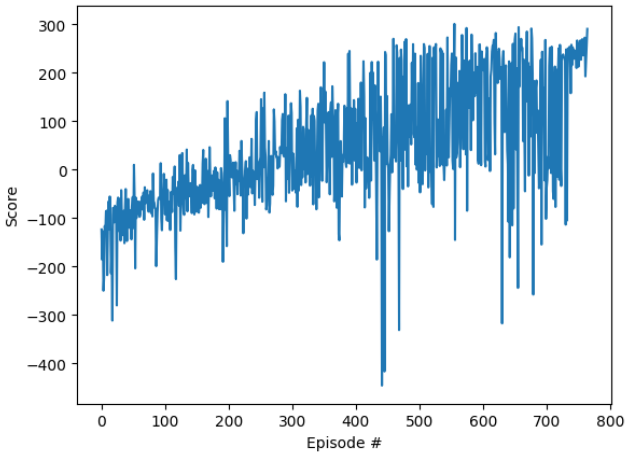


**Figure 4.** Standard DQN Score per Episode

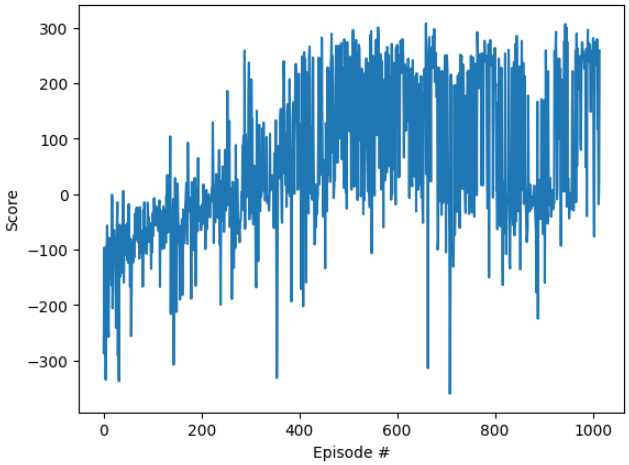


**Figure 5.** Double DQN Score per Episode

Prioritized Experience Replay is a technique used to improve the efficiency of learning. In traditional experience replay, transitions are sampled uniformly from the replay memory. However this technique introduces prioritization, meaning that transitions with higher importance are sampled more frequently. This importance is typically determined by the magnitude of the TD error associated with the transition [18]. Because of this improvement, the algorithm manages to sample more successful transitions more frequently, thus solving the problem in a more efficient manner.

On the other hand, the double DQN (Figure 5) performed the worst, solving the problem in slightly over 1000 episodes.

### 4.2 Experiment 2

As can be seen from comparing Figures 8 and 9, it is clear that SAC performed better in this environment. It solved the problem in around 120 episodes while DDPG solved the problem in around 250 episodes. This may be due to the fact

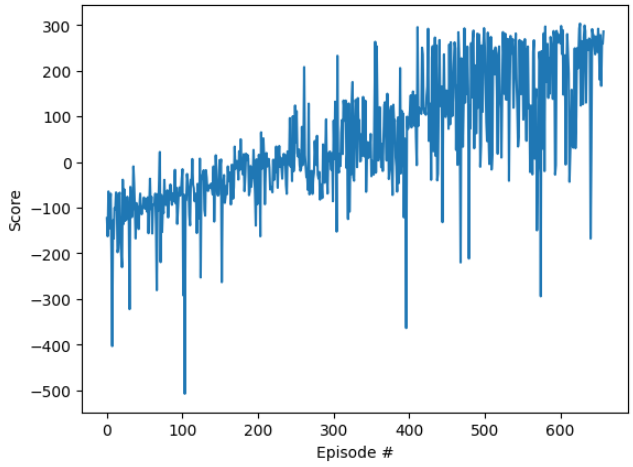


**Figure 6.** Standard DQN with Priority Experience Replay Score per Episode

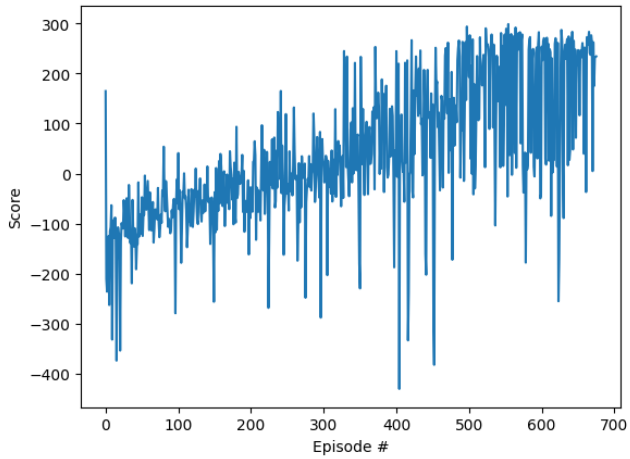


**Figure 7.** Double DQN with Priority Experience Replay Score per Episode

that SAC uses a stochastic policy to learn while DDPG uses a deterministic policy.

As previously explained, a stochastic policy maps each state to a probability distribution over actions. This means that for a given state, the agent will choose an action randomly based on the probability distribution. This approach allows the agent to explore different actions within the same state, increasing the likelihood of finding the optimal solution. Furthermore, stochastic policies can capture the uncertainty in the environment, making them more suitable for tasks involving uncertainty or exploration. This may explain why SAC performed better than DDPG.

It can also be noted that both algorithms performed significantaly better than the standard DQN and the DQN variations tested throughout experiment 1. Although the environment is slightly different, it shows how beneficial it can be

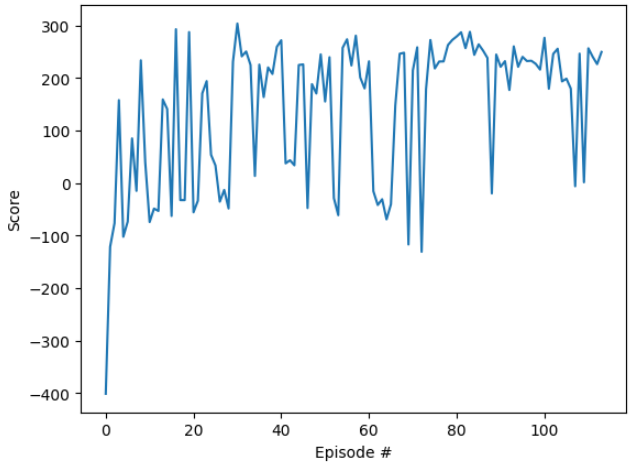


**Figure 8.** SAC Score per Episode

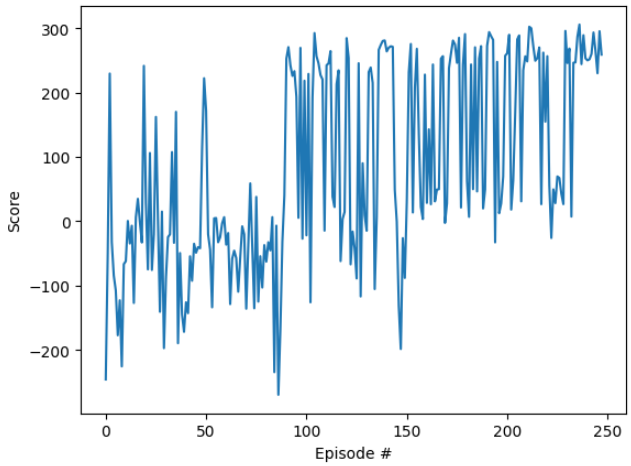


**Figure 9.** DDPG Score per Episode

to use a combination of both value based and policy based methods in order to solve RL problems.

## References


[1] S. Agarwal. 2023. Introduction to the Actor-Critic Model. https://www.codingninjas.com/studio/library/introduction-to-the-actor-critic-model Accessed: 29/01/2024.

[2] P. Bajaj. 2023. Reinforcement learning. https://www.geeksforgeeks.org/what-is-reinforcement-learning/ Accessed: 25/01/2024.

[3] T. Carr. 2023. Deterministic vs. Stochastic Policies in Reinforcement Learning. https://www.baeldung.com/cs/rl-deterministic-vs-stochastic-policies Accessed: 04/02/2024.

[4] K. Doshi. 2020. Reinforcement Learning Explained Visually (Part 4): Q Learning, step-by-step. https://towardsdatascience.com/reinforcement-learning-explained-visually-part-4-q-learning-step-by-step-b65efb731d3e Accessed: 30/01/2024.

[5] Hugging Face. 2024. The advantages and disadvantages of policy-gradient methods. https://huggingface.co/learn/deep-rl-course/unit4/advantages-disadvantages Accessed: 04/02/2024.

[6] Hugging Face. 2024. Two types of value-based methods. https://huggingface.co/learn/deep-rl-course/unit2/two-types-value-based-methods Accessed: 30/01/2024.

[7] Hugging Face. 2024. What is RL? A short recap. https://huggingface.co/learn/deep-rl-course/unit2/what-is-rl Accessed: 29/01/2024.

[8] E. Gomede. 2023. Deep Q-Networks (DQN): Bridging the Gap between Deep Learning and Reinforcement Learning. https://medium.com/@evertongomede/deep-q-networks-dqn-bridging-the-gap-between-deep-learning-and-reinforcement-learning-5cd73d644c7 Accessed: 31/01/2024.

[9] Intellipaat. 2023. Supervised Learning vs Unsupervised Learning vs Reinforcement Learning. https://intellipaat.com/blog/supervised-learning-vs-unsupervised-learning-vs-reinforcement-learning/ Accessed: 29/01/2024.

[10] D. Karunakaran. 2020. The Actor-Critic Reinforcement Learning algorithm. https://medium.com/intro-to-artificial-intelligence/the-actor-critic-reinforcement-learning-algorithm-c8095a655c14 Accessed: 04/02/2024.

[11] O. Klimov. 2023. Lunar Lander - Gymnasium Documentation. https://gymnasium.farama.org/environments/box2d/lunar_lander/ Accessed: 26/01/2024.

[12] V. V. Kumar. 2019. Soft Actor-Critic Demystified. https://towardsdatascience.com/soft-actor-critic-demystified-b8427df61665 Accessed: 03/02/2024.

[13] Q. T. Luu. 2023. Q-Learning vs. Deep Q-Learning vs. Deep Q-Network. https://www.baeldung.com/cs/q-learning-vs-deep-q-learning-vs-deep-q-network Accessed: 31/01/2024.

[14] OpenAI. 2018. Deep Deterministic Policy Gradient. https://spinningup.openai.com/en/latest/algorithms/ddpg.html Accessed: 04/02/2024.

[15] OpenAI. 2018. Soft Actor-Critic. https://spinningup.openai.com/en/latest/algorithms/sac.html Accessed: 03/02/2024.

[16] B. Or. 2021. Value-based Methods in Deep Reinforcement Learning. https://towardsdatascience.com/value-based-methods-in-deep-reinforcement-learning-d40ca1086e1 Accessed: 29/01/2024.

[17] L. Owen. 2024. Bird's-Eye View of Reinforcement Learning Algorithms Taxonomy. https://dataheadhunters.com/academy/reinforcement-learning-exploring-policy-vs-value-based-methods/ Accessed: 29/01/2024.

[18] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. 2015. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).

[19] T. Sweeney. 2023. Lunar Lander - Open AI. https://wandb.ai/timssweeney/lunar-lander/reports/Lunar-Lander-Open-AI--VmlldzoyNzg5NjE Accessed: 02/02/2024.

[20] Tensorflow. 2023. Playing CartPole with the Actor-Critic method. https://www.tensorflow.org/tutorials/reinforcement_learning/actor_critic Accessed: 29/01/2024.

[21] M. Wang. 2020. Deep Q-Learning Tutorial: minDQN. https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc Accessed: 31/01/2024.