Solving Control Problems using Reinforcement Learning

Gabriel Hili
gabriel.hili.20@u.edu.mt

ACM Reference Format:
Gabriel Hili. 2023. Solving Control Problems using Reinforcement Learning. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 What is Reinforcement Learning

Reinforcement Learning (RL) is a type of machine learning that involves an agent learning to interact with an environment in order to maximize a reward. [1] The agent receives feedback in the form of positive or negative rewards for its actions, and it learns to choose actions that maximise the cumulative reward over time. Reinforcement learning algorithms typically use some form of value function or action-value function to represent the expected long-term reward for a given state or state-action pair. These functions are iteratively updated based on the agent's experiences in the environment, using techniques such as temporal difference learning or Q-learning.

1.2 How it differs from other ML approaches

There exist many different Machine Learning (ML) approaches. Supervised learning consists of labelled input-output pairs, and the goal is to learn a function that accurately predicts the output given the input. In contrast, unsupervised learning does not work with labelled data, and it is up to the model to learn any patterns present in the data. RL does not work with predefined data, but samples from an environment. The model can receive an observation of the environment as some encoding, choose which action to take based on this encoding, perform the action on the environment, receive some reward based on the current state-action pair performed, and repeats. The goal of RL is to learn a state-action function such that given any state, the model knows which actions will maximise the cumulative reward.

1.3 Value-Based, Policy-Based, and Actor-Critic Models

Value-based methods involve an agent learning a value function that estimates the long-term reward for each state or state-action pair. This value function helps the agent determine which actions will lead to the highest expected reward in a given state. The value function is usually represented as a table or function and is continually updated based on the agent's experiences in the environment. The agent uses the value function to choose actions that maximise the expected reward.

Unlike value-based methods, policy-based methods do not learn a value function, and focus on learning a policy directly, which maps from states to actions. The model follows this policy in order to take actions given the current state. Policy-based methods are useful for high-dimensional or continuous action spaces as there is no need to enumerate through every state-action pair. Moreover, they can also learn a stochastic policy which means unlike value-based methods, different actions can be taken given the same state. Generally, policies are updated using gradient-based optimisation techniques such as Stochastic Gradient Descent (SDG).

Actor-Critic based algorithms are a combination of value-based and policy-based approaches, where both a value function and a policy are learnt. The critic learns the value function while the policy is learnt by the actor. The value function is used to evaluate the policy while the policy is used to determine the agent's actions. The actor takes an action based on the current policy, which is evaluated by the critic. The action is performed on the environment and a new state and reward are received. The critic updates the parameters of its value function using the reward received by taking the action. Similarly, using the updated value function from the critic, the actor then updates its policy.

In summary, value-based, policy-based, and actor-critic methods are all different approached towards a reinforcement learning problem. Value-based methods aim to assign Q-Values to every state, or state-action pairs in order to take the action that maximises the cumulative reward given the current state. This value-function can take the form of a table, or a function estimator like a neural network. Policy-based methods learn a policy directly and are useful for high-dimensional, continuous action spaces, or if its difficult to learn a value function. Actor-Critic methods combine both value-based and policy-based methods. The actor performs actions based on a policy, while the critic evaluates the actions of the actor based on its value function.

2 BACKGROUND

2.1 Value-Based Methods

Value-based methods in Reinforcement Learning involve the use of a value function, which estimates the expected long-term reward for a given state or state-action pair. This value function is used by the agent to select actions that will maximize the expected reward in a given state. The value function can be represented as a table or a function, and is continually updated based on the agent's interactions with the environment.

The value function is updated based on the difference between the expected reward and the observed reward for a given state or state-action pair, as can be seen by Fig.1. $Q(s,a)$ is the current estimate of the value of taking action a in state s. α is the learning rate,

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

```
Q[s][a] = Q[s][a]+alpha*(r+gamma*max(Q[s])-Q[s][a])
```

Figure 1: Equation and Python Code used to update the value function Q

which determines the weight given to new information. A value of 0 means that new information is ignored, while a value of 1 means that the new information completely replaces the old estimate. r is the observed reward for taking action a in state s. γ is the discount factor, which determines the importance of future rewards. A value of 0 means that only the current reward is considered, while a value of 1 means that all future rewards are given equal weight. s' is the next state that the agent transitions to after taking action a in state s. a' is the action taken in the next state s'. $Q(s',a')$ is the current estimate of the value of taking action a' in state s'.

An example of a value-based algorithm is Q-Learning. Q-Learning is usually used with an ϵ-Greedy policy. Initially, the algorithm takes actions at random and updates the Q-value of the state-action pair seen using the equation seen in Fig.1. As more episodes are played out, the explorative approach diminishes and an exploitative approach is favoured. Given the current state, the action with the highest Q-value from that state is taken.

2.2 Deep Q-Networks

Deep Q-Networks (DQN) are similar to value-based methods in that they utilise a value function in order to determine the action to take given the current state. However, when the environment state is large or sometimes even infinite, it becomes impossible to store a table for all possible state-action pairs. Hence a function estimator is used, which given the current state (encoded as some vector), will output the best action to take.

DQN utilise neural networks as the function estimator. Similarly to Q-Learning, DQN are evaluated in the same way. Instead of consulting the Q-Table for the next action, the neural network is used. It is important to note that the agent's experiences are highly correlated, because the actions taken by the agent in a given state depend on the actions taken in the previous states. This can make it difficult for the neural network to learn an optimal control policy, because the training data is not Independent and Identically Distributed (IID).

This problem is remedied by maintaining a replay memory, which helps to stabilize the learning process by storing a sample of the agent's experiences. A replay memory is a sample of every state-action-reward-state tuple that the agent has experienced. Then the neural network is trained using a mini-batch from the replay-buffer. This helps to smooth out the learning process and can lead to better performance.

In this algorithm, the label for the training data is being sampled from the network itself, which contrasts to regular supervised learning, where the labels are fixed. Fixing the labels of a given experience by creating a copy of the current network, Q' which will be used to update the current Q improves performance. As the authors of the original paper put it [2]: *Generating the targets using an older set of parameters adds a delay to the time an update to Q is made and the time the update affects the targets y_j, making divergence or oscillations much more unlikely.*

```
1 target_qvalues = dqn_target(b_next_states)
2 max_target_qvalues = torch.max(target_qvalues, axis=1).values.unsqueeze(1)
3 expected_qvalues = b_rewards + GAMMA*(1-b_dones.type(torch.int64)) *
        max_target_qvalues
```

Figure 2: Python Implementations of line 12 from Algorithm 1. *Main Update Equation: Line Number 3*

The main update equation for DQN is seen on **Line 12 in Algorithm 1** and the code implementation is seen in **Fig. 2**

Algorithm 1 Deep Q-Network (DQN) Algorithm

```
 1: $Q \leftarrow$ Policy Network
 2: $Q' \leftarrow Q$                                    ▷ Target Network
 3: $R \leftarrow$ Replay Buffer
 4: $c \in \mathbb{N}$
 5: for each episode $e$ do
 6:     $s \leftarrow 0$
 7:     for each $step$ in $e$ do
 8:         $a \leftarrow \epsilon$-Greedy$_Q$
 9:         $r, s' \leftarrow env.step(a)$
10:         $R.append((s,a,r,s'))$                     ▷ Append to Replay Memory
11:         $batch \leftarrow sample(R)$
12:         $y \leftarrow r + \gamma \max_{a'} Q'(s',a')$ for each $(s,a,r,s')$ in $batch$
13:         Evaluate $batch$ on $Q$, update weights using current estimate and $y$
14:         $s \leftarrow s'$
15:         if $step$ mod $c = 0$ then
16:             $Q' \leftarrow Q$
```

2.3 Policy-Based Methods

Policy-based methods in reinforcement learning involve learning a policy directly, rather than learning a value function that is used to determine the optimal policy. These methods are useful in cases where the space of possible policies is large or continuous, or when it is difficult to define a value function due to intractability. The main differences between value-based methods are that policy-based methods do not learn a Q-Value Function and simply learn a policy directly, and policy-based methods can learn stochastic policies unlike value-based methods, which means they may take different actions given the same observation.

Policy gradient methods are a type of policy-based method that involve learning a policy by estimating the gradient of the expected reward with respect to the policy parameters and using it to update the policy in the direction that increases the expected reward [3]. The aim of policy-gradient methods is to learn a parameterised policy by optimising the parameters using a gradient-based method. Unlike value-based method, a Q-Value Function is not learned. This makes it useful for scenarios where the state-action space is large or infinite. Similarly to other policy-based methods, policy-gradient

$$\theta_{t+1} = \theta_t + \alpha \cdot \hat{A}(s,a) \cdot \Delta_\theta log\pi_\theta(s|a)$$

Figure 3: Update rule for the parameters θ of policy π in Policy-Gradient Methods

$$\theta = \theta + \alpha \cdot \gamma^t \cdot G \cdot \Delta ln\pi_\theta(A_t|S_t)$$

Figure 4: Update rule for the parameters θ of policy π in REINFORCE [4]. G is the cumulative discounted reward based on some discount factor γ.

methods are able to learn stochastic policies and handle continuous actions, which is unable to be achieved using value-based methods such as Q-Learning. However, policy-gradient methods suffer from high variance estimates of the gradient update, which can make the learning process difficult.

Policy-Gradient methods are typically updated using the equation depicted in Fig.3. In this equation, θ_t is the policy parameters at time step t, θ_{t+1} is the policy parameters at time step $t+1$, α is the learning rate, $\hat{A}(s,a)$ is the advantage function at state s and action a, and $\Delta_\theta log\pi_\theta(s|a)$ is the gradient of the log of the policy with respect to the policy parameters at state s and action a. The advantage function $\hat{A}(a|s)$ is a measure of how much better action a is at state s compared to the average of all actions. It is typically defined as the difference between the expected return and the value function estimate at state s and action a. The value function estimate is a prediction of the expected return from state s and action a. Note that although policy-gradient methods do not use a value function in order to select actions, a value function estimate may still be used to learn the optimal policy.

The main update equation used in Policy-Gradient Methods can be seen in Fig.3. The update equation used in REINFORCE can be seen in Fig. 4 and its code implementation can be seen in Fig. 5.

2.4 Actor-Critic Methods

Actor-Critic algorithms use two neural networks to take the best action given the current state to maximise the cumulative future reward (actor), and to determine the usefulness of that action in giving a reward (critic) which in return improves the actor network. As previously mentioned, value-based methods only update a value-function, and policy-based methods only work update a policy. Actor-Critic methods are a combination of the two and have both a policy-network (the actor) and a value-approximator (the critic). Actor-Critic methods generally achieve better performance over value and policy-based methods however they are computationally more intensive.

One state-of-the-art Actor-Critic algorithm is Deep Deterministic Policy Gradient (DDPG), which is an off-policy algorithm that updates its models based on a fixed policy [5]. The training process for DDPG involves collecting a batch of experiences in the form of (state, action, reward, next state) tuples from the environment. These experiences are used to update the critic model, whilst the actor model is updated using the gradient of the expected reward

```
1  def grad_log_p(obs):
2     y = obs @ theta
3     grad_log_p0 = obs - obs*logistic(y)
4     grad_log_p1 = - obs*logistic(y)
5
6     return grad_log_p0, grad_log_p1
7
8
9  def update(rewards, obs, actions):
10    #Get Gradient Log Probabilities
11    gradlogp = np.array([
12       grad_log_p(ob)[action] for ob,action in zip(obs,actions)])
13
14    discounted_rewards = np.zeros(len(rewards))
15    cumulative_rewards = 0
16
17    #Get Cumulative Discounted Rewards
18    for i in reversed(range(0, len(rewards))):
19       cumulative_rewards = cumulative_rewards * gamma + rewards[i]
20       discounted_rewards[i] = cumulative_rewards
21
22    #Multiply Log Probabilities With Cumulative Discounted Rewards
23    dot = gradlogp.T @ discounted_rewards
24
25    #Apply Learning Rate
26    return alpha*dot
```

Figure 5: Python Implementation of the REINFORCE update rule seen in Fig. 4. *Main Update Equation:* 26

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

Figure 6: Target Value Computation for DDPG Q-Function (Actor Model)

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

Figure 7: Main Update Equation for the Actor Model in DDPG. B is a set of sample experiences from the replay memory

with respect to the policy parameters.

DDPG uses replay memory and fixed target networks to stabilise learning, which makes it more sample efficient and less sensitive to hyperparameters than regular actor-critic algorithms. Moreover DDPG is suited for continuous action-space environments. Ornstein-Uhlenbeck is added to the action produced by the actor in order to encourage exploration. Finally, gradient descent is used to update the actor and critic networks.

Fig. 7 shows the main update equation used to update the actor model given a sample batch from the memory replay B. Similarly, Fig. 8 shows the main update equation used to update the critic model. The target networks of each model are updated by taking a weighted average over the current target parameters of the actual *local* networks based on some parameter τ. The code implementation of Fig. 7 and Fig. 8 can be seen in Fig. 13.

Soft Actor-Critic (SAC) is another state-of-the-art actor-critic method [6]. It is an online, model-free, off-policy actor-critic reinforcement learning method and is a combination of DDPG and

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

Figure 8: Main Update Equation for the Critic Model in DDPG. B is a set of sample experiences from the replay memory

```
1  # Critic loss
2  Qvals = critic(states, actions)
3  with torch.no_grad():
4      actions_ = actor_target(next_states)
5      Qvals_ = critic_target(next_states, actions_)
6      Qvals_[dones] = 0.0
7      target = rewards + GAMMA * Qvals_
8  critic_loss = F.smooth_l1_loss(target, Qvals)
9
10 # Actor loss
11 actor_loss = -critic(states, actor(states)).mean()
```

Figure 9: Python Implementation of DDPG update rules. *Target Value Computation:* 7. *Actor Update Equation:* 11. *Critic Update Equation: Line Number* 8.

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

Figure 10: Target value computation for SAC Q-Function (Actor Model), where $\tilde{a}' \sim \pi_{\theta}(\cdot|s')$

stochastic-policy optimisations. SAC computes an optimal policy that maximizes both the long-term expected reward and the entropy of the policy, which is a measure of policy uncertainty given the state. Maximizing both the expected cumulative long-term reward and the entropy balances exploration and exploitation in the environment. The SAC algorithm uses entropy regularised value and Q functions and updates the policy by trying to maximise the value function. Maximising both the expected cumulative long-term reward and the entropy balances exploration and exploitation of its environment. This entropy regularised term encourages the agent to explore the environment and take actions that are uncertain, rather than always taking the action that is expected to result in the highest reward.

In order to update our critic network, we must take the derivative of the expected reward. This poses a problem because we do not know the true data distribution. This is remedied by reparameterising the function such that it allows for gradient-based optimisation. We do this by introducing a new set of parameters such that they can be chosen to be normally distributed. By doing this, we can take the gradient of the expected reward plus the entropy regularisation term with respect to the parameters of the actor network, which we want to optimise. This allows us to use standard gradient-based optimisation techniques, such as stochastic gradient descent (SGD) to optimise the actor network.

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

Figure 11: Main Update Equation for the Actor Model in SAC. B is a set of sample experiences from the replay memory

$$\nabla_{\theta}\frac{1}{|B|}\sum_{s\in B}min_{\forall i\in\{1,2\}}(Q_{\Phi_i}(s,\tilde{a}_{\Theta}(s))-\alpha\cdot log\ \pi_{\Theta}(\tilde{a}_{\Theta}(s)|s))$$

Figure 12: Main Update Equation for the Critic Model in SAC. B is a set of sample experiences from the replay memory. $\tilde{a}_{\theta}(s)$ is a sample from $\pi_{\theta}(\cdot|s)$ which is differentiable w.r.t to θ

```
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, actions)
31     )
32     policy_loss = (ALPHA * log_probs - action_values).mean()
33
34     policy_optimizer.zero_grad()
35     policy_loss.backward()
36     policy_optimizer.step()
```

Figure 13: Python Implementation of SAC update rules. *Target Value Computation:* 16. *Actor Update Equation:* 32. *Critic Update Equation:* 20.

3 METHODOLOGY

3.1 Experiment 1

3.1.1 Problem Definition. The aim of Experiment 1 is to successfully achieve an average reward of 195 over the last 50 episodes in the 'LunarLander-v2' environment, using DQN and a policy-based algorithm (REINFORCE). In the LunarLander-v2 environment, the agent must learn to control the thrust of the lunar lander's engines in order to land the spacecraft safely on the landing pad. The environment provides the agent with observations of the spacecraft's

Hyperparameter	Value
GAMMA	0.99
BATCH_SIZE	64
BUFFER_SIZE	10,000
LEARNING_RATE	0.01
MIN_REPLAY_SIZE	5,000
EPS_START	1.0
EPS_DECAY	0.995
EPS_END	0.05
TARGET_UPDATE_FREQ	0.05

Table 1: DQN Hyperparameters

Hyperparameter	Value
GAMMA	0.99
MAX_EPISODES	10,000
LEARNING_RATE	0.01

Table 2: REINFORCE Hyperparameters

position, velocity, and other relevant variables, and the agent must choose actions in the form of thrust commands to control the spacecraft's descent. The agent receives a reward for successfully landing the spacecraft on the landing pad, and incurs a penalty for crashing or running out of fuel.

The goal of the agent in the LunarLander-v2 environment is to learn a policy that maximizes the cumulative reward it receives over time. This requires the agent to learn to make good decisions about how to control the spacecraft's descent in order to land it safely on the landing pad. The observation vector of the LunarLander environment is of length 8 and the action space is of length 4.

3.1.2 Configuration & Hyperparameters. Table 1 shows the parameters for the DQN algorithm. `GAMMA` represents the discount factor. `BATCH_SIZE` represents the amount of transitions sampled from the Replay Memory and passed to the DNN. `BUFFER_SIZE` represents the maximum amount of transitions stored by the Replay Memory at any given time. `LEARNING_RATE` is the learning rate used for the ADAM optimiser. `MIN_REPLAY_SIZE` represents the minimum amount of transitions stored by the Replay Memory at any given time. `EPS_START` represents the starting value of ϵ in the ϵ-Greedy Policy. `EPS_DECAY` is multiplied with ϵ every episode. `EPS_END` is the minimum value ϵ can be. `TARGET_UPDATE_FREQ` is the minimum value ϵ can be. The DNN is composed of 8 inputs, 64 hidden neurons, and 4 outputs. The Activation Function is *tanh*. The optimiser is *ADAM*. The Loss Function is $SmoothL1Loss$.

Similarly, the hyperparameters for REINFORCE are shown in Table 2. `GAMMA` represents the discount factor. `MAX_EPISODES` represents the maximum amount of episodes that will be experienced till an average score of 195 over the last 50 episodes is reached. The DNN is composed of 8 inputs, 16 hidden neurons, and 4 outputs. The Activation Function is $ReLU$. The optimiser is *ADAM*.

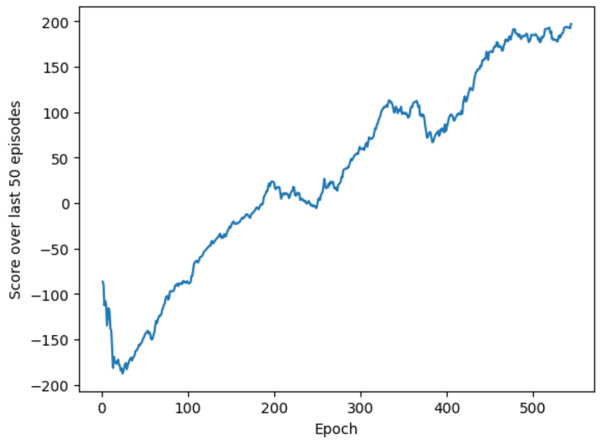

Figure 14: DQN rewards over episodes experienced in the LunarLander-v2 environment.

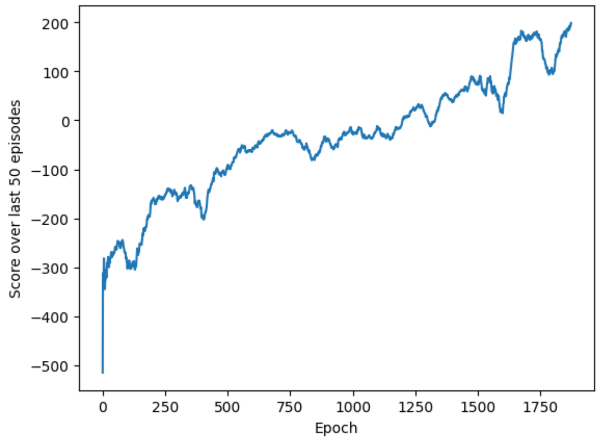

Figure 15: REINFORCE rewards over episodes experienced in the LunarLander-v2 environment.

3.1.3 Measures to validate experiment. The average reward over the last 50 episodes was tabulated every episode and stored in an array. This is so we can gauge the performance of the model as it reaches towards to an average score of 195.

3.2 Experiment 2

3.2.1 Problem Definition. Similarly to Experiment 1, the aim of Experiment 2 is to successfully achieve an average reward of 195 over the last 50 episodes in the 'LunarLanderContinuous-v2' environment, two actor-critic algorithms (SAC and DDPG). In contrast to the discrete LunarLander-v2 environment, LunarLanderContinuous only has an action space of 2, the left and right thrusters. However since the actions are continuous, they also encode the ability to not use any one thruster (i.e a value of 0). The observation vector is similair to the discrete environment, that is the spacecraft's position, velocity, et., Similarly, the agent also receives a reward for

Hyperparameter	Value
GAMMA	0.99
BATCH_SIZE	64
BUFFER_SIZE	10,000
LEARNING_RATE	0.001
MIN_REPLAY_SIZE	5,000
EPSILON	0.05
ALPHA	0.02
TAU	0.05

Table 3: SAC Hyperparameters

Hyperparameter	Value
GAMMA	0.99
BATCH_SIZE	64
BUFFER_SIZE	10,000
LEARNING_RATE	0.0003
MIN_REPLAY_SIZE	5,000
TAU	0.01
STD_DEV	0.2

Table 4: DDPG Hyperparameters

successfully landing the spacecraft on the landing pad, and incurs a penalty for crashing or running out of fuel.

3.2.2 Configuration & Hyperparameters. Table 3 show the parameters for the SAC algorithm. GAMMA represents the discount factor. BATCH_SIZE represents the amount of transitions sampled from the Replay Memory and passed to the DNN. BUFFER_SIZE represents the maximum amount of transitions stored by the Replay Memory at any given time. LEARNING_RATE is the learning rate used for the ADAMW optimiser, for both actor and critic networks. MIN_REPLAY_SIZE represents the minimum amount of transitions stored by the Replay Memory at any given time. EPSILON represents the fixed value for the ϵ-Greedy Policy. ALPHA represents the learning rate in the equation used to compute the target Q-Function (See Fig. 10). TAU is used for taking a weighted average of the target network and local network policy parameters in order to update the target network parameters. Both the DNN for the actor and critic networks have a hidden layer of 256 neurons.

Similarly, the hyperparameters for DDPG are shown in Table 4.GAMMA represents the discount factor. BATCH_SIZE represents the amount of transitions sampled from the Replay Memory and passed to the DNN. BUFFER_SIZE represents the maximum amount of transitions stored by the Replay Memory at any given time. LEARNING_RATE is the learning rate used for the ADAMW optimiser, for both actor and critic networks. MIN_REPLAY_SIZE represents the minimum amount of transitions stored by the Replay Memory at any given time. TAU is used for taking a weighted average of the target network and local network policy parameters in order to update the target network parameters. STD_DEV is a hyperparameter for the Ornstein-Uhlenbeck noise function.

3.2.3 Measures to validate experiment. Similarly to Experiment 1, the average reward over the last 50 episodes was tabulated every

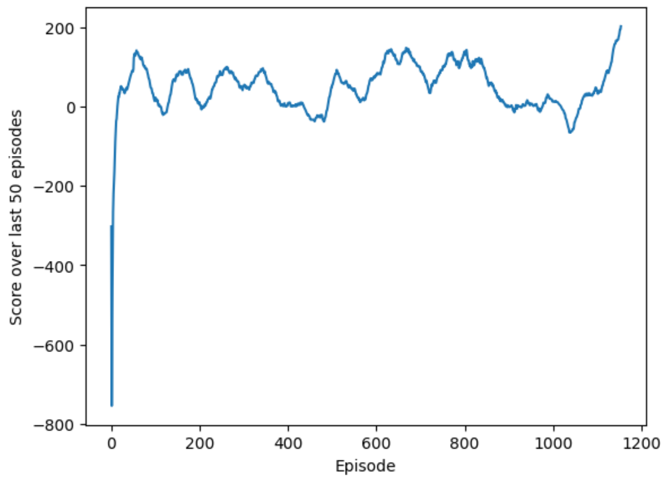

Figure 16: DDPG rewards over episodes experienced in LunarLanderContinuous-v2 environment

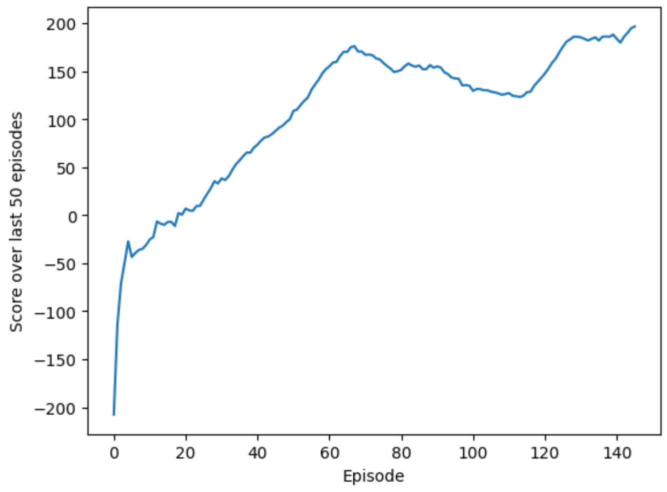

Figure 17: SAC rewards over episodes experienced in LunarLanderContinuous-v2 environment

episode and stored in an array, which allows us to view the training process of the models as a graph.

3.3 Libraries Used

The core Python libraries used to perform both experiments were: Gym, PyTorch, Pandas, Numpy and MatPlotlib.

4 RESULTS AND DISCUSSIONS

4.1 Experiment 1

As can be seen by Fig. 14, an average score of 195 was achieved after experiencing 500 episodes when using DQN with the previously mentioned hyperparameters (See Table 1). Similarly, as can be seen by Fig. 15, an average score of 195 was achieved after experiencing 1700 episodes when using the REINFORCE algorithm with the previously mentioned hyperparameters (See Table 2). Taking account multiple training instances, DQN always converges faster and more consistently, whereas REINFORCE often times gets stuck in negative reward loops and requires more episodes to converge to an average score of 195. In conclusion, DQN is far better suited for the LunarLander-v2 environment.

4.2 Experiment 2

In contrast to the discrete environment, SAC and DDPG managed to converge to an average score of 195 in 150 and 1200 episodes respectively. As can be seen by Fig 17, SAC was quick to converge to the intended value and did not get stuck in any local maximums. On the other hand, DDPG quickly converged to an average score of 0 but floated around that value for almost a 1000 episodes. Then by the end, an average score of 195 was reached. Based on the training history of DDPG, it was unlikely that it would exceed 195 and most probably the average reward would fall back down to 0. This is not the case for SAC which quickly took an upward trend and could potentially reach higher average scores.

4.3 Conclusion

In conclusion, the project discusses two experiments using reinforcement learning algorithms on the LunarLander-v2 and LunarLander Continuous-v2 environments. The goal of both experiments is to achieve an average reward of 195 over the last 50 episodes. Experiment 1 uses a DQN and a policy-based algorithm (REINFORCE) while Experiment 2 uses two actor-critic algorithms (SAC and DDPG). The experiment details the specific configurations and hyperparameters used, as well as the measures used to validate the experiment (keeping track of the average score every episode). For the discrete environment, it was found that DQN with the hyperparameters detailed in Table. 1 outperformed REINFORCE with the hyperparameters detailed in Table 2, in terms of convergence speed and stability. Similarly for the continuous environment, it was found that SAC with the hyperparameters detailed in Table 3 outperformed DDPG with the hyperparameters detailed in Table 4, in terms of convergence speed and stability. REINFORCE and DDPG took multiple attempts of training in order to reach an average score of 195, because they would get stuck in local maximum, or not be able to exceed an average score of 0. Hence, these algorithms are more unstable and difficult to apply to similar real-world scenarios due to their unreliability (at least with the current hyperparameters used). On the other hand, DQN and SAC immediately converged to the intended value and would probably achieve a higher average score if left to train for longer. In summary, DQN was the best algorithm for the discrete environment and SAC was the best algorithm for the continuous environment.

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.
[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.
[3] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Advances in neural information processing systems*, vol. 12, 1999.
[4] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3, pp. 229–256, 1992.
[5] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.
[6] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning.* PMLR, 2018, pp. 1861–1870.