Solving Control Problems using Reinforcement Learning

Matthias Bartolo
matthias.bartolo.21@um.edu.mt
University of Malta
Msida, Malta

1 Introduction

1.1 Brief Overview of Reinforcement Learning

Reinforcement Learning (RL) is the process of learning what to do—how to translate events into actions—in order to maximise a numerical reward signal [1]. In RL, the learner is not taught which actions to perform but instead must experiment to determine which actions provide the highest reward. Additionally, RL is distinguished by two key characteristics: reliance on trial-and-error search and the capacity to maximise rewards over extended periods rather than seeking immediate satisfaction. The RL framework is often broken down into an interaction loop, as can be seen in Figure 1. In the loop, the RL agent executes some action in an environment based on some policy that transitions its current state to a new one while collecting some reward and an observation [1].

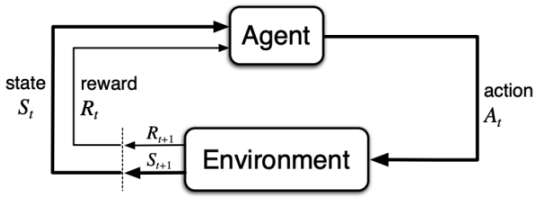

Figure 1. RL Interaction Loop (Source: [1])

The policy, which is frequently represented as π, serves as a function that guides the agent's decision-making process in an environment. It associates each state s_t with an action A_t, determining the agent's movements from the current state s_t to the next state s_{t+1} at a given time step t. R_{t+1} represents the numerical value that the agent obtains as a result of its actions in a state. These rewards, R_t, are critical in refining the agent's policy, assisting it in determining the appropriate action to take in various states in order to achieve its ultimate goal: the optimal policy π^*. Maximizing the predicted cumulative reward, which is the sum of all rewards collected over multiple time steps, is required to achieve this optimum policy [2]. Furthermore, in RL the environment refers to the external system or context with which an agent interacts and learns. It encompasses everything outside the agent's control, presenting states and rewarding the agent's actions, which are critical for the agent to observe, learn, and make decisions in a dynamic environment.

1.2 Markov Decision Process

Formally, RL may be expressed as a *Markov Decision Process (MDP)* [3]. An MDP is a mathematical framework for modelling sequential decision-making problems in which the present state of the system, potential actions, and probability of transitioning to the next state are known [4]. Moreover, an MDP can be modelled as the following tuple: (S, A, P, R, γ), whereby:

- S is a finite set of possible states $s_t \in S$ observed by the agent at time step t.
- A is a finite set of available actions $a_t \in A$ that the agent can execute at time step t.
- P is the transition dynamics $p(s_{t+1}|s_t, a_t)$ that determine the probability of moving to s_{t+1} after taking action a_t in state s_t.
- R is the immediate reward function $r(s_t, a_t, s_{t+1})$ that provides the reward obtained after transitioning from s_t to s_{t+1} through action a_t.
- γ is the discount factor $\gamma \in [0, 1]$ that regulates the balance between immediate and future rewards and shapes the agent's decision-making.

These MDPs can be used to represent both continuous and episodic tasks. The latter refers to tasks that can be completed in a finite number of timesteps within a finite time frame: $1 \leq t \leq T$, whereby the task possesses a terminal state. On the other hand, continuous tasks refer to processes that have no terminal state, thus enabling the RL agent to continue to interact with the environment indefinitely.

1.3 Brief Overview of Deep Reinforcement Learning

Employing techniques such as bootstrapping, dynamic programming, and tabular search, RL methods face the issue of handling and generalizing large-scale data in broad environments. A solution to this problem pertains to the use of *Deep Reinforcement Learning (DRL)*. DRL combines the concept of RL methods with *Deep Learning (DL)*, connecting their strengths to solve shortcomings in classic tabular techniques. As described in [3], DRL effectively navigates away from the confines of manually built state representations by combining complex neural network topologies such as *Artificial Neural Networks (ANNs)* and *Convolutional Neural Networks (CNNs)*. It excels at exploring huge and complicated search spaces, allowing RL agents to generalise their learning across different settings. Deep RL uses neural networks to estimate

value functions or rules, as opposed to traditional tabular techniques, which suffer from scalability in contexts with exponentially vast state spaces. This in turn allows agents to learn and make decisions in high-dimensional and continuous state spaces, enabling them to display flexible behaviour across a wide range of challenging situations [3].

1.4 Differentiating RL from traditional ML methods

Although reinforcement learning is a subset of machine learning, it differs from traditional machine learning methods, such as supervised and unsupervised learning, in a number of ways. As previously mentioned, in RL, the agent learns by interacting with the environment and receiving feedback in the form of rewards or penalties for its actions, as opposed to supervised and unsupervised learning, which utilise learning algorithms that require a large amount of labelled examples and unlabeled data, respectively. Moreover, while supervised and unsupervised learning algorithms are frequently presumed to be distinct and equally distributed in the realm of ML, RL usually entails sequential decision-making, in which the agent's actions in the present might impact the rewards or penalties it receives in the future. Although supervised and unsupervised learning are typically used for classification, dimensionality reduction, or feature extraction, RL is goal-oriented, with the agent's purpose being to adopt a policy that maximises cumulative reward over time. Unlike supervised learning, which is often a batch process, RL is typically an online learning method that allows the agent to gain experience over time.

1.5 Comparing Value-Based, Policy-Based, and Actor-Critic RL Models

Value-Based, *Policy-Based*, and *Actor-Critic* are the three models that collectively form the basis of the RL decision-making paradigms. Value-based models, such as *Q-learning*, make use of a function $V(s)$ or $Q(s,a)$ to evaluate the value associated with states or state actions, respectively, and choose the action with the highest estimated value [1]. On the contrary, policy-based models, such as *REINFORCE*, generate action probabilities directly based on states, selecting the action with the highest likelihood [5]. Actor-critic models integrate both of the aforementioned techniques by employing an *actor* to select actions and a *critic* to assess the quality of the chosen actions. As a result, this would modify the actor's behaviour over time, depending on the critic's input [6]. Fundamentally, the main contrast between these paradigms is determined by the way each model chooses the optimal action: value-based models depend on estimated values, policy-based models rely on probability, and actor-critic models rely on both value estimates and probabilities.

2 Background

2.1 Value Based Methods

As previously mentioned, value-based methods are a type of RL algorithm that makes use of a function $V(s)$ or $Q(s,a)$ to evaluate the value associated with states or state actions, respectively. The equations for these functions are illustrated in Equations 1 and 2 reflecting the expected return from being in a particular state or executing an action from a particular state, respectively. Furthermore, the aforementioned functions that are derived from the *Bellman equation* can be broken down into a summation of the immediate rewards R_{t+1} and the discounted expected future reward $V(S_{t+1})$. [1]

$$V(s) = \mathbb{E}\left[R_{t+1} + \gamma \cdot V\left(S_{t+1}\right) \mid S_t = s\right] \tag{1}$$

$$Q(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \cdot Q(S_{t+1}, A_{t+1}) | S_t = s, A_t = a\right] \tag{2}$$

Aligning with the goal of RL to select a policy that maximises expected return, these methods use an implicit policy that selects the optimal action while converging to the optimal value function. Moreover, these value-based methods refer to a variety of methodologies, including iterative value methods and Q-learning [1]. As illustrated in Figure 2, the value iteration algorithm uses the Bellman optimality equation to update the state value, resulting in a deterministic policy that simulates the optimal value function ($V^*(s)$).

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
  $\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$

Figure 2. Value Iteration Algorithm. (Source: [1])

Similarly, *Q-learning* attempts to approximate the $Q(s,a)$ function by iteratively updating action values depending on interactions with the environment. The *temporal difference (TD)* error between anticipated and observed Q-values is used to evaluate the best action-value function [7]. Moreover, Q-learning tries to converge towards the optimal Q-values ($Q^*(s,a)$) through this iterative process, allowing the agent to make informed decisions by selecting actions that maximise these learnt action values. The Q-learning algorithm uses an off-policy update method, as illustrated in Figure 3 whereby the Q-learning agent would select an action based on an $\epsilon - greedy\ policy$, while updating its Q-values based on the highest expected return, i.e., the largest Q-values of

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 3. Q-Learning Algorithm. (Source: [1])

the successor state[1]. The equation for the Q-values update function can be seen in Equation 3.

$$Q(s, a) = Q(s, a) + \alpha \cdot \left(R + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a)\right) \quad (3)$$

2.2 Deep Q-Learning Methods

2.2.1 Deep Q Network. The Deep Q Network (DQN) is an extension of the Q-learning technique but replaces the tabular Q-values with a deep neural network. This modification attempts to increase generalisation capabilities, particularly in cases with huge state spaces, which traditional Q-learning and memory techniques fail to control [3]. As illustrated in Equation 4 the DQN utilises a deep neural network in the form of a new parameter θ which symbolises the deep neural network's weights. In addition, the DQN also utilises the loss function, as described in Equations 5 and 6 to achieve the aforementioned optimal state action value function [8].

$$Q(s, a; \theta) \approx Q^*(s, a; \theta) \quad (4)$$

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(.)}\left[(y_i - Q(s, a|\theta_i))^2\right] \quad (5)$$

$$y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q\left(s', a'; \theta_{i-1}\right) \mid s, a\right] \quad (6)$$

Listing 1. Deep Q Network: Network Update

```
# Calculating the Q-values for the current states
qvalues = self.policy_net(states).gather(1, actions)

with torch.no_grad():
    # Calculating the Q-values for the next states using the
        target network
    target_qvalues = self.target_net(next_states)
    # Calculating the maximum Q-values for the next states
    max_target_qvalues = torch.max(target_qvalues, axis=1).values.
        unsqueeze(1)
    # Calculating the next Q-values using the Bellman equation
    next_qvalues = rewards + GAMMA * (1 - dones.type(torch.float32
        )) * max_target_qvalues

# Calculating the loss
loss = self.criterion(qvalues, next_qvalues)

# Backpropagating Loss
self.optimizer.zero_grad()
loss.backward()
```

The deep Q-learning algorithm employed in DQN utilises both a policy network and a target network to facilitate efficient and stable learning. The policy network, or Q-network, continually approximates Q-values for state-action pairings in deep Q-learning to guide action selection. Similarly to normal Q-learning, the policy network updates the Q-values through the temporal difference error and gradient descent, as illustrated in Listing 1. Subsequently, the target network, a separate network with fixed parameters updated from the policy network on a regular basis, stabilises learning by providing consistent Q-value objectives during training [8].

DQNs also employ an experience *Replay Buffer*, which can be used to sample through transition data during training in order to facilitate more stable learning. These transitions consist of a tuple of five elements containing: state, action, reward, done, and after state. Transitions in the environment are added to a replay buffer, and the loss and gradient are calculated using a batch of transitions sampled from the replay buffer rather than utilising the most recent transition while training [1, 8].

Listing 2. Deep Q Network: Select Action Method

```
def select_action(self, state):
    if random.random() <= self.epsilon: # Exploration
        action = self.env.action_space.sample()
    else: # Exploitation
        with torch.no_grad():
            state = torch.from_numpy(state).float().unsqueeze(0).
                to(device)
            qvalues = self.policy_net(state)
            action = qvalues.argmax().item()
    return action
```

Considering that DQN is an extension of Q-learning, it also employs the $\epsilon - greedy$ policy to select an action. Thus, ensuring a balance between agent exploration and exploitation [1, 3]. As illustrated in Listing 2, the select action method utilises the aforementioned policy, whereby the optimal action is chosen with a probability of $1 - \epsilon$ otherwise, a random action is chosen.

Listing 3. Double Deep Q Network: Network Update

```
# Calculating the Q-values for the current states
qvalues = self.policy_net(states).gather(1, actions)

# Calculating the Q-values for the next states
with torch.no_grad():
    # Using the policy network to select the action with the
        highest Q-value for the next states
    next_state_actions = self.policy_net(next_states).argmax(dim
        =1, keepdim=True)
    # Using the target network to calculate the Q-value of the
        selected action for the next
    next_qvalues = self.target_net(next_states).gather(1,
        next_state_actions)
    # Calculating the next Q-values using the Bellman equation
    target_qvalues = rewards + GAMMA * (1 - dones.type(torch.
        float32)) * next_qvalues

# Calculating the loss
loss = self.criterion(qvalues, target_qvalues)
```

2.2.2 Double Deep Q Network. The *Double Deep Q-Network (DDQN)* algorithm is an improvement on the traditional DQN approach, aimed at addressing overestimation difficulties in Q-learning methods and reducing maximisation bias [9]. DDQN works by separating action selection from action value estimation using two independent neural networks. In DDQN, the network with the highest Q-value determines

the ideal action, whereas the alternate network evaluates the value of that action. This difference is apparent when comparing both Equations 7 and 8, and Listings 1 and 3. Additionally, DDQN improves the algorithm's stability and learning efficiency by limiting the overestimation bias found in single-network Q-learning techniques [3].

$$Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a; \boldsymbol{\theta}_t^-\right) \tag{7}$$

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q\left(S_{t+1}, \operatorname{argmax} Q\left(S_{t+1}, a; \boldsymbol{\theta}_t\right), \boldsymbol{\theta}_t^-\right) \tag{8}$$

2.2.3 Dueling Deep Q Network.

The *Dueling Deep Q-Network (Dueling-DQN)* architecture presents a significant improvement on the traditional DQN aimed at providing more efficient learning whilst allowing the model to more precisely identify crucial state values and advantageous actions [3]. Moreover, Dueling-DQN presents a revolutionary design that separates the estimation of state values and the assessment of each action independently of the value of remaining in a specific state [10]. The revised Dueling-DQN architecture is apparent in Listing 4.

Listing 4. Dueling Deep Q Network: Architecture

```
class DuelingDQN(nn.Module):
    def __init__(self, ninputs, noutputs):
        super(DuelingDQN, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(ninputs, 64),
            nn.Tanh(),
            nn.Linear(64, 32)
        )
        self.valfunc = nn.Linear(32, 1)
        self.advfunc = nn.Linear(32, noutputs)

    def forward(self, X):
        o = self.net(X)
        value = self.valfunc(o)
        adv = self.advfunc(o)
        return value + adv - adv.mean(dim=-1, keepdim=True)
```

2.3 Policy Based Methods

Policy-based methods are types of RL algorithms that compute action probabilities directly from states, choosing the action with the highest probability [5]. This implies that, rather than estimating the value of each action in a given state (as in value-based methods), these methods concentrate on learning a parameterised policy ($a = \pi(s|\theta)$) that directly offers the likelihood of doing each action in any given state [11]. Moreover, the parameter θ in policy-based methods is updated iteratively through gradient descent and back-propagation. Policy-based methods such as the *REINFORCE* algorithm choose the action with the highest probability, thus enabling stochastic action selection, in which multiple actions are chosen with changing probability based on the learnt policy's outputs for a given state [1].

2.3.1 Difference between Policy-based & Value-based.

Value-based methods seek to estimate the optimal value function ($V^*(s)$) by estimating the anticipated returns for state-action combinations, whereas policy-based approaches seek to directly optimise the best policy by mapping states to actions in order to maximise returns ($a = \pi(s|\theta)$) [11]. In addition, while value-based methods estimate expected returns for each state-action combination, policy-based methods create probability distributions across actions per state, reflecting the likelihood of actions delivering favourable outcomes [1, 3]. Value-based methods rely on external exploration techniques, such as $\epsilon - greedy$ policy, whereas policy-based approaches add variability intrinsically by sampling stochastically from the policy's action probability distribution [1, 11]. An additional difference between value-based and policy-based techniques pertains to value-based methods being best suited for scenarios with small and discrete action-spaces. Policy-based techniques, on the other hand, have the ability to generalise effectively over continuous action spaces due to the parameterised characteristics of the policy [11].

2.4 Actor-Critic Methods

Actor-critic methods merge the above value-based and policy-based methods to form an architecture that incorporates an *actor* that executes actions and a critic that evaluates the chosen action [1, 6]. The actor-critic reinforcement learning architecture is depicted as a high-level block diagram in Figure 4.

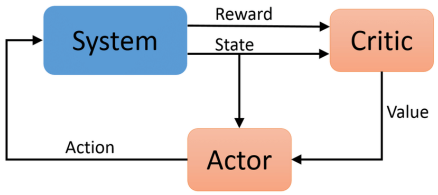

Figure 4. Actor Critic Architecture. (Source: [12])

In the actor-critic framework, the actor uses a policy-based mechanism based on a parameterised policy ($\pi_\theta(s, a)$) to select actions. The critic, on the other hand, often adopts a value-based strategy, estimating the expected rewards associated with the actions specified by the actor's policy using a parameterised value function ($\hat{q}_w(s, a)$) [6].

$$\Delta\theta = \alpha\nabla_\theta(log\pi_\theta(s, a))\hat{q}_w(s, a) \tag{9}$$

Mathematically, the actor network adjusts its policy weights, as shown in Equation 9, where the critic network, which provides the action value estimate ($q_w(s, a)$), is critical in calculating this update at each time step t. Following the policy adjustment in the actor network, the actor then selects the next action, a_{t+1}, based on the new state, s_{t+1}. As a result, the critic network adjusts its parameters using Equation 10. [1, 6, 12]

$$\Delta w = \beta(R(s, a) + \gamma\hat{q}_w(s_{t+1}, a_{t+1}) - \hat{q}_w(s_t, a_t))\nabla_w\hat{q}_w(s_t, a_t) \tag{10}$$

2.4.1 Stochastic and Deterministic Policies. Actor-critic methods can be categorised into two groups: those who employ a *deterministic policy* (Equation 11) and those who employ a *stochastic policy* (Equation 12). Stochastic policies, which are similar to policy-based techniques, function by having the actor construct a probability distribution that includes all possible actions for a given state. In addition, this technique utilises both exploration and exploitation while ensuring that less desirable actions have a likelihood of being selected. On the other hand, deterministic policies work similarly to the mechanics of value-based techniques, by deterministically selecting the same action for a given state. As a result, such an approach inherently facilitates no exploration when compared to the stochastic policy approach. Furthermore, to introduce exploration, various strategic methods can be utilised, such as applying bootstrapping or introducing exploration noise in action selection. These methods are essential in deterministic policies, as they allow variability in action selection while ensuring random exploration. [1, 13]

$$\pi : S \rightarrow A \tag{11}$$

$$\pi : S \times A \rightarrow [0, 1] \tag{12}$$

2.4.2 Proximal Policy Optimisation with Generalized Advantage Estimation. *Proximal Policy Optimisation (PPO)* is an on-policy RL algorithm that can be used in both discrete and continuous environments. As an on-policy algorithm, PPO modifies its policy by gathering experiences directly from the present policy, guaranteeing that the data used for learning corresponds with the policy being updated. The technique uses a *trust region optimisation* strategy to manage the amount of policy updates, preventing excessive deviations that might jeopardise stability. Despite the fact that on-policy algorithms often require more samples than off-policy algorithms, PPO's on-policy characteristics add to its dependability and adaptability for scenarios with changing dynamics. Additionally, PPO alternates between sampling data from the environment and optimising a *surrogate* objective function, which enables multiple epochs of minibatch updates using stochastic gradient ascent. The PPO surrogate function, which is illustrated in Equation 13 can be further simplified as depicted in both Equations 14 and 15. Equation 14 clips the loss value to a maximum of $g(\epsilon, A^{\pi_{\theta_k}}(s, a))$, whilst assigning the minimum value to the policy update ratio ($\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$) given that it would be smaller. On the other hand, Equation 15 modifies the advantage estimate (A) depending on a parameter (epsilon), altering positive and negative advantages separately to regulate the update amount within the PPO algorithm. [14]

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right] \tag{13}$$

$$L\left(s, a, \theta_k, \theta\right) = \min \left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)} A^{\pi_{\theta_k}}(s, a),\ g\left(\epsilon, A^{\pi_{\theta_k}}(s, a)\right) \right) \tag{14}$$

where

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0 \end{cases} \tag{15}$$

Generalized Advantage Estimation (GAE) is an RL method for estimating the advantages of taking actions in a particular state. It computes an estimate of the advantage function, which indicates the benefit of performing a certain action in a specific condition over a random action. In addition, GAE also introduces the γ parameter to control the trade-off between variance and bias in the advantage estimation. Introducing GAE to the PPO objective function allows for improved learning by recording detailed advantage estimations, resulting in more refined decision-making and flexibility. The calculation of GAE is depicted through Equation 16. [15]

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{(l=0)}^{\infty} (\gamma\lambda)^l \delta^V{}_{t+1} \tag{16}$$

The network update for the PPO algorithm with GAE, showcasing the policy update ratio and clipping function, can be illustrated in Listing 5. Additionally, Listing 6, showcases the advantage estimation calculation.

Listing 5. PPO with GAE: Network Update

```
obs, actions, logprobs, rewards, dones, obs_, values, values_, gae = batch
# Normalizing the generalized advantage estimation
gae = (gae - torch.mean(gae).to(device)) / (torch.std(gae) + 1e-6).to(device)
# Calculating the target value
target = gae + values
# Getting the state values from the critic network
state_values = self.critic_net(obs)
# Calculating the critic loss
critic_loss = self.criterion(state_values, target).mean()
# Getting the mean and standard deviation from the actor network
new_loc, new_scale = self.actor_net(obs)
# Creating a normal distribution from the mean and standard deviation
dist = Normal(loc=new_loc, scale=new_scale+1e-6)
# Calculating the log probability of the action
new_logprobs = torch.sum(dist.log_prob(actions), dim=-1, keepdim=True)
# Calculating the importance sampling ratio (Policy Update Ratio)
rho = torch.exp(new_logprobs - logprobs).to(device)
# Calculating the surrogate 1 (Policy Update Ratio * Advantage)
surrgt1 = rho * gae
# Calculating the surrogate 2 (Clipping)
surrgt2 = rho.clamp(1-self.epsilon, 1+self.epsilon) * gae
# Calculating the policy loss (Minimum)
policy_loss = -torch.minimum(surrgt1, surrgt2).mean().to(device)
# Calculating the loss
loss = policy_loss + 0.5*critic_loss
# Zeroing the gradients of the actor and critic networks
self.actor_optimizer.zero_grad()
self.critic_optimizer.zero_grad()
# Backpropagating the loss
loss.backward()
# Updating the weights of the actor and critic networks
self.actor_optimizer.step()
self.critic_optimizer.step()
```

Listing 6. Generalized Advantage Estimation Calculation

```
# Iterating through the running memory in reverse
for i in range(len(b_obs)-1,-1,-1):
    # Calculating the delta value
    delta = b_rewards[i] + GAMMA * b_values_[i] * (1-b_dones[i]) -
        b_values[i]
    # Calculating the GAE
    gae = delta + GAMMA * LAMBDA * (1-b_dones[i]) * gae
    # Appending the GAE to the list of GAEs
    gaes.insert(0, gae)
```

2.4.3 Deep Deterministic Policy Gradient.

Deep Deterministic Policy Gradient (DDPG) is an actor-critic, model-free RL algorithm that applies the concepts of Deep Q-Learning's success to the domain of continuous action. The DDPG architecture as depicted in Figure 5, employs a deterministic off-policy policy gradient method that allows for operation in continuous action spaces. A DDPG agent maintains four function approximators to estimate the policy and value functions. The *actor* network in DDPG employs a policy network $\pi(s|\theta)$ that, given a state, outputs a deterministic continuous action rather than the typical probability distribution. The *critic* network is a Q-value network $Q(s, a|\phi)$ that accepts state action pairs as input and outputs a Q-value. The two additional networks that DDPG incorporates are the *target actor* $\pi_{targ}(s|\theta_t)$ and the *target critic* $Q_{targ}(s, a|\phi_t)$. These target networks are updated regularly every few iterations, contrary to the main actor and critic networks, to facilitate more stable training. This is due to the inclusion of these networks inside the loss function. In addition, DDPG employs an experience replay buffer to introduce exploration during training by drawing on prior experiences. This technique is based on the use of a DQN within the DDPG framework. Moreover, DDPG further enhances exploration by integrating *Ornstein-Uhlenbeck* action noise generation into its framework. [16]

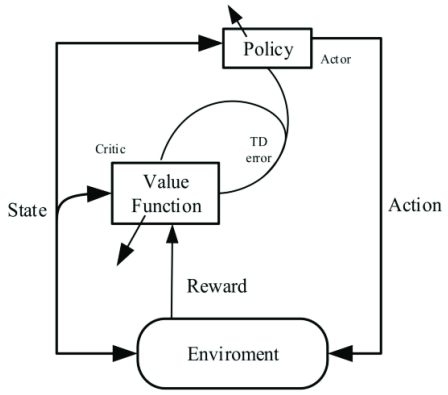

Figure 5. Deep Deterministic Policy Gradient Architecture. (Source: [17])

The DDPG update function starts by first randomly sampling a batch of transitions from the replay buffer. The target values are then computed from the respective target networks using Equation 17.

$$y\left(r, s', d\right) = r + \gamma(1-d)Q_{\text{targ}}\left(s', \mu_{\text{targ}}\ (s')\right) \tag{17}$$

Following this, the Q-function is updated using gradient descent as described in Equation 18, and the policy is updated using a one-step gradient ascent as described in Equation 19.

$$\nabla_\phi \frac{1}{|\mathcal{B}|} \sum_{(s,a,r,s',d)\in\mathcal{B}} \left(Q_\phi(s,a) - y\left(r, s', d\right)\right)^2 \tag{18}$$

$$\nabla_\phi \frac{1}{|\mathcal{B}|} \sum_{s\in\mathcal{B}} Q_\phi\left(s, \mu_\phi(s)\right) \tag{19}$$

Finally, the target networks' parameters are updated through Equations 20 and 21. The DDPG network update algorithm, as illustrated by the above equations, corresponds to the network update function shown in Listing 7.

$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta \tag{20}$$

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi \tag{21}$$

Listing 7. DDPG: Network Update

```
def update(self):
    # Retrieving a batch of transitions from the replay buffer
    states, actions, rewards, dones, next_states = self.
        replay_buffer.sample_batch()
    # Resetting the action noise
    self.action_noise.reset()
    # Calculating the Critic loss
    Qvals = self.critic(states, actions)
    with torch.no_grad():
        # Getting the actions from the policy target network
        actions_ = self.actor_target(next_states)
        # Getting the Q values from the critic target network
        Qvals_ = self.critic_target(next_states, actions_)
        # Setting the Q values to 0 if the episode is done
        Qvals_[dones] = 0.0
        # Calculating the target
        target = rewards + GAMMA_DDPG * Qvals_
    # Calculating the critic loss
    critic_loss = self.criterion(target, Qvals)
    # Calculating the actor loss
    actor_loss = -self.critic(states, self.actor(states)).mean()
    # Zeroing the gradients of the policy network, and the
        optimizer
    self.policy_optimizer.zero_grad()
    actor_loss.backward()
    self.policy_optimizer.step()
    # Zeroing the gradients of the critic network, and the
        optimizer
    self.dqn_optimizer.zero_grad()
    critic_loss.backward()
    self.dqn_optimizer.step()
    # Updating the target networks
    for target_param, param in zip(self.actor_target.parameters(),
        self.actor.parameters()):
        target_param.data.copy_(param.data * TAU + target_param.
            data * (1.0 - TAU))
    for target_param, param in zip(self.critic_target.parameters()
        , self.critic.parameters()):
        target_param.data.copy_(param.data * TAU + target_param.
            data * (1.0 - TAU))
```

3 Methodology

This paper examines the effectiveness of the aforementioned algorithms as a sequence of experiments for approximating the *Lunar Lander problem* [18]. This classic rocket trajectory optimisation problem was provided in a pre-built gymnasium environment and follows Pontryagin's maximum principle (it is optimal to fire engines at full throttle). As illustrated in

Figure 6, the goal for this problem pertains to manoeuvring the Lunar Lander between two flag poles while ensuring that both side rockets make contact with the ground. To guarantee a thorough and equal evaluation, the algorithms were repeatedly tested and reviewed until they met the predetermined success criteria of reaching an average reward of 195.

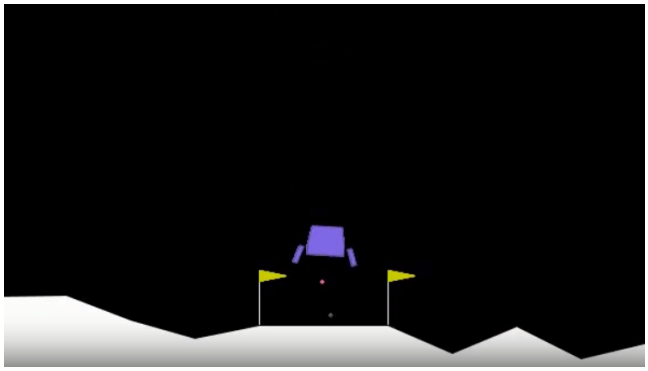

Figure 6. Lunar Lander Problem

The observation space for this problem can be formalised as an 8-dimensional vector containing the following values:

1. X position
2. Y position
3. X velocity
4. Y velocity
5. Angle
6. Angular velocity
7. Left leg touching the ground
8. Right leg touching the ground

Additionally, at every step of a Lunar Lander episode, RL agents are provided with rewards varying based on proximity to the landing pad, lander velocities, and tilting away from horizontal alignment. The episode culminates in a significant reward adjustment, with crashes resulting in a -100 point penalty and safe landings awarding an additional +100 points. Furthermore, the episode terminates with a lander crashing into the moon's surface, moving outside of the screen, or remaining inactive, signifying no collisions or movement.

3.1 Experiment 1

In Experiment 1, the agents underwent testing within a discrete action space characterised by the following actions:

1. Do nothing
2. Fire left orientation engine
3. Fire main engine
4. Fire right orientation engine

3.1.1 Standard / Normal DQN. The Standard DQN architecture was composed of two primary layers: a 64-neuron linear layer, followed by a Tanh activation function, and finally an output layer with neurons matching the required number of output actions. Listings 1 and 2 illustrate the use of such an algorithm.

3.1.2 Double DQN. The Double DQN architecture enhanced the standard DQN architecture by separating the action selection component from the action value estimation component using two independent neural networks identical to the standard DQN architecture. Partition of these components is displayed in Listing 3.

3.1.3 Dueling DQN. The Dueling DQN architecture refined the Standard DQN architecture by altering the network architecture that separates the value estimation and advantage functions, allowing for more efficient learning and robust environment representations. The architecture of the revised architecture is apparent in Listing 4.

3.1.4 Double Dueling DQN. The Double Dueling DQN architecture combined the refinements from the above architectures, employing two independent networks as detailed in Listing 3, whilst adopting the Dueling DQN architecture as depicted in Listing 4.

The above RL agents employed consistent hyperparameters, as described in Table 1, following extensive testing and several fine-tuning iterations.

Alpha	Gamma	Loss	Batch	Optimiser
0.001	0.99	SmoothL1	128	Adam

Table 1. Algorithm parameters for Experiment 1

3.2 Experiment 2

In Experiment 2, the agents underwent testing within a continuous action space characterised by the following actions:

1. Thrust: (-1.0, 1.0) for the main engine
2. Rotation: (-1.0, 1.0) for the Lunar Lander

3.2.1 PPO with GAE. The PPO with GAE architecture, as described in Chapter 2, employed two network policies consisting of an actor network and a critic network. Both the critic and actor networks comprised of two linear layers separated by two Tanh activation functions. Their outputs, however, differed significantly as the actor network provided the mean and standard deviation, whereas the critic network produced the value corresponding to the state-action pair. Furthermore, GAE was also utilised for estimating the advantages of taking actions in a particular state for further refined decision making as illustrated in Listings 5 and 6.

3.2.2 DDPG. The DDPG architecture was constructed to have two network definitions, an actor and a critic, whilst also employing *Ornstein-Uhlenbeck* action noise generation. Additionally, it utilised two target network structures that mirror the main ones. The critic networks employed a DQN architecture, while the policy network consisted of three linear layers interspersed with two ReLU activation functions

and concluded with a Tanh activation function. The network update for these architectures is depicted in Listing 7.

The algorithm parameters for this experiment are described in Table 2.

Model	Alpha	Gamma	Loss	Batch	Optimiser
PPO+GAE	0.0003	0.999	SmoothL1	64	AdamW
DDPG	0.0003	0.99	SmoothL1	64	AdamW

Table 2. Algorithm parameters for Experiment 2

4 Results & Discussion

4.1 Experiment 1

The results of Experiment 1 are depicted in Figures 7, 8, 9 and 10. The analysis of results highlights Dueling DQN as the top-performing algorithm, displaying the fastest convergence among the experimented variants. Following closely is Double Dueling DQN, followed by Double DQN, and lastly, Standard DQN, as evidenced in Figure 7. The average rewards per episode also demonstrated that while Standard DQN exhibited a relatively large variation within its updates, the remaining algorithms displayed more gradual improvements and comparatively better performance, characterised by more stabilised updates.

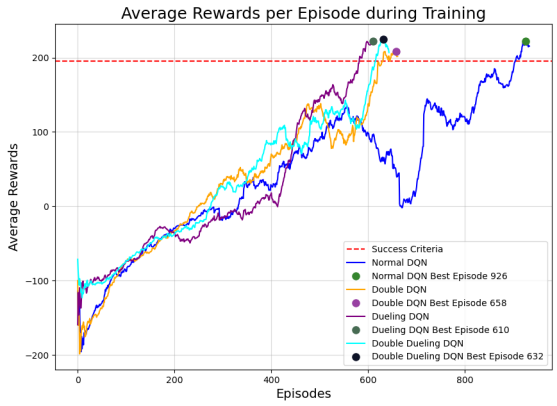

Figure 7. Experiment 1: Average Rewards per Episode

Moreover, detailed examination, as showcased in Figure 9, indicates that Double Dueling DQN achieved the highest maximum episode average rewards, reaching a value of 224.22, closely followed by Dueling DQN at 222.39, and Standard DQN at 222.29, with Double DQN trailing at 208.54.

Additionally, Figure 8 reveals a gradual increase in cumulative episode lengths per episode. These trends collectively underscore the efficacy of Dueling DQN and its variations in achieving faster convergence and higher rewards compared to the other tested models.

Based on Figure 10, it was noted that Double DQN had the quickest completion time, followed by Dueling, Double Dueling, and Standard DQN in ascending order. Overall, the

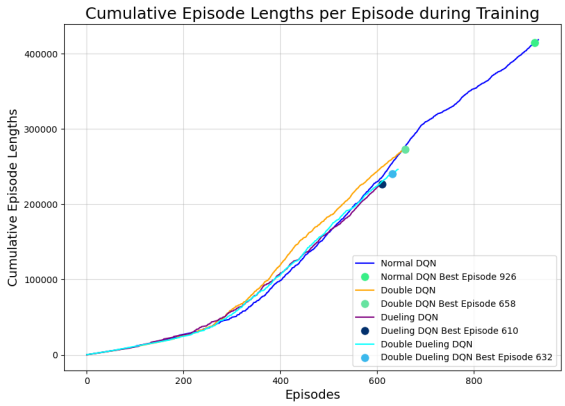

Figure 8. Experiment 1: Cumulative Episode Lengths per Episode

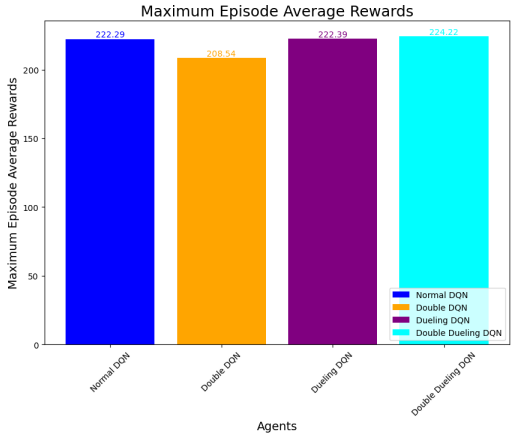

Figure 9. Experiment 1: Maximum Episode Average Rewards

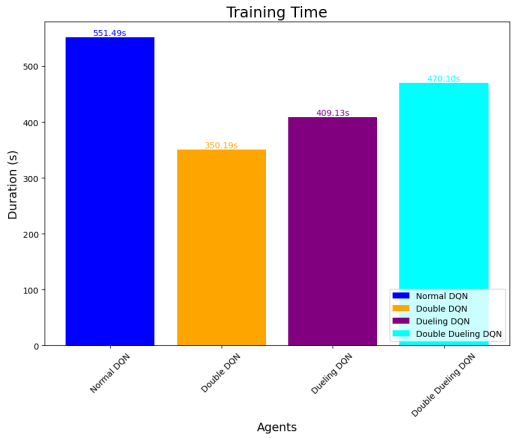

Figure 10. Experiment 1: Training Time

results show that Dueling DQN was the top-performing RL algorithm in the Lunar lander environment, with Double

Dueling DQN falling somewhere between Dueling and Double DQN. Notably, Double DQN performed far better than Standard DQN. However, all algorithms performed relatively well, with just some minor discrepancies between them. It's important to note that the tests were repeated several times to guarantee accurate readings and thorough assessments. This minimal variation in performance can be attributed to the environment's simplicity, which affected the slight differences detected among the algorithms.

4.2 Experiment 2

Figures 11, 12, 13, 14, and 15 depict the outcomes derived from Experiment 2. Figure 11 depicts the implementation of PPO with GAE, which involves the use of eight concurrent environments, utilising parallelization to accelerate the learning process. Analysing the average rewards per episode across all eight environments shows a degree of high performance between them. Taking the best average rewards per episode, it was noted that *Environment 1* was the best-performing environment.

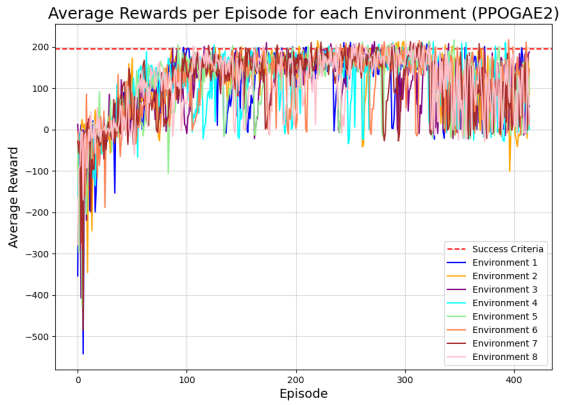

Figure 11. Experiment 2: Average Rewards per Episode for each Environment (PPOGAE2)

Figure 12 compares the top-performing environment for the PPO with GAE algorithm to the average rewards produced by DDPG which uses a single environment. Particularly, the results showed that the stochastic policy utilised by PPO with GAE converged faster than the deterministic policy used in DDPG. However, while DDPG required a longer time to converge, it showed a more stable update in average rewards, in contrast to the significant fluctuation shown in PPO with GAE results.

Figure 13 revealed a clear pattern in which PPO with GAE demonstrated a linear rise in the cumulative episode durations for each episode. DDPG, on the other hand, showed a considerably smaller change in this parameter. This distinction might be attributable to a variety of variables, including the computational techniques used by each method, the unique qualities of their policies, or the fundamental nature of the Lunar Lander environment.

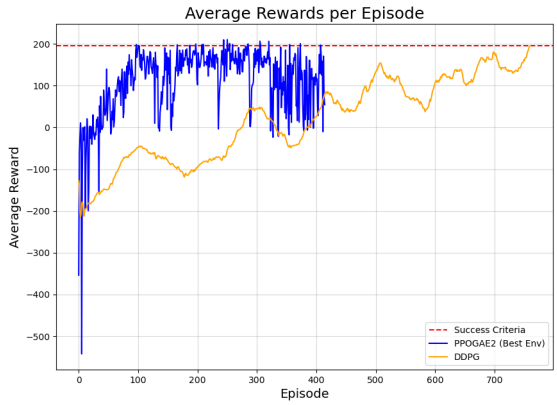

Figure 12. Experiment 2: Average Rewards per Episode

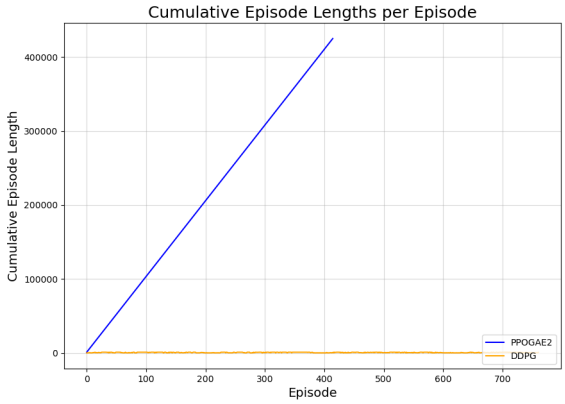

Figure 13. Experiment 2: Cumulative Episode Lengths per Episode

Figures 14 and 15 demonstrate that PPO with GAE achieved a higher maximum episode average reward, which is most likely due to its parallelization-based training process. DDPG, on the other hand, closely followed in obtaining relatively good performance. Additionally, despite the fact that PPO with GAE was trained over many parallel environments, it took significantly longer to complete than DDPG, which was trained simply in a single environment.

5 Conclusion

The experiments conducted in this paper provided insights on the performance subtleties among reinforcement learning algorithms with respect to the Lunar Lander environment. In the case of a discrete action space in Experiment 1, Dueling DQN was the best-performing algorithm with a relatively

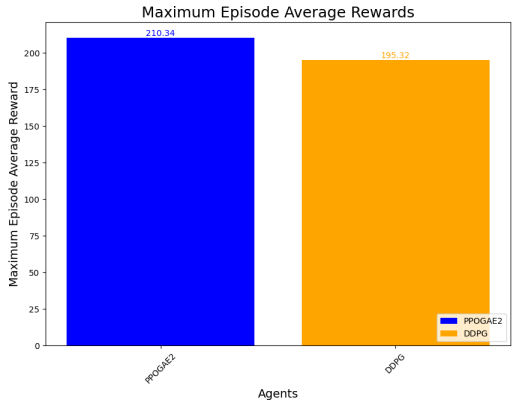

Figure 14. Experiment 2: Maximum Episode Average Rewards

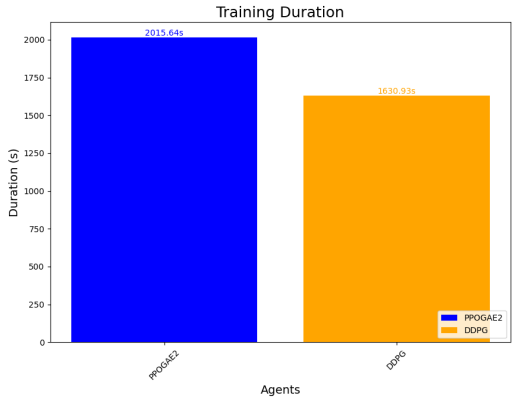

Figure 15. Experiment 2: Training Duration

quick convergence, followed by Double Dueling DQN, Double DQN, and Standard DQN. From the evaluation of this research, it was observed that all the proposed modified architectures consistently outperformed the original Standard DQN architecture. Additionally, it can also be noted that all DQNs were able to finish their environments far quicker than both actor-critic methods in Experiment 2. This outcome was anticipated, as allowing for continuous action spaces creates a new state-action dimension with a theoretically limitless number of potential inputs, considerably complicating the state-action space and exposing the agent to further noise during training. Furthermore, Experiment 2 demonstrated the opposing dynamics of PPO with GAE and DDPG, with PPO demonstrating faster convergence and greater maximum rewards due to parallelization but requiring longer to train than DDPG's single-environment training.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book.html

[2] A. W. M. Leslie Pack Kaelbling, Michael L. Littman, "Reinforcement learning: A survey," *CoRR*, vol. cs.AI/9605103, 1996. [Online]. Available: https://arxiv.org/abs/cs/9605103

[3] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," *CoRR*, vol. abs/1708.05866, 2017. [Online]. Available: http://arxiv.org/abs/1708.05866

[4] M. Otterlo and M. Wiering, "Reinforcement learning and markov decision processes," *Reinforcement Learning: State of the Art*, pp. 3–42, 01 2012.

[5] F. AlMahamid and K. Grolinger, "Reinforcement learning algorithms: An overview and classification," in *2021 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, Sep. 2021. [Online]. Available: http://dx.doi.org/10.1109/CCECE53047.2021.9569056

[6] I. Grondman, L. Busoniu, G. Lopes, and R. Babuska, "A survey of actor-critic reinforcement learning: standard and natural policy gradients," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 42, no. 6, pp. 1291–1307, 2012. [Online]. Available: https://hal.science/hal-00756747

[7] B. Jang, M. Kim, G. Harerimana, and J. W. Kim, "Q-learning algorithms: A comprehensive classification and applications," *IEEE Access*, vol. 7, pp. 133 653–133 667, 2019.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[9] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015. [Online]. Available: http://arxiv.org/abs/1509.06461

[10] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," *CoRR*, vol. abs/1511.06581, 2015. [Online]. Available: http://arxiv.org/abs/1511.06581

[11] M. Sewak, *Policy-Based Reinforcement Learning Approaches*. Singapore: Springer Singapore, 2019, pp. 127–140. [Online]. Available: https://doi.org/10.1007/978-981-13-8285-7_10

[12] Y. Wang, K. Velswamy, and B. Huang, "A long-short term memory recurrent neural network based reinforcement learning controller for office heating ventilation and air conditioning systems," *Processes*, vol. 5, 09 2017.

[13] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ser. ICML'14. JMLR.org, 2014, p. I–387–I–395.

[14] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[15] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1506.02438

[16] 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," in *ICLR*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://dblp.uni-trier.de/db/conf/iclr/iclr2016.html#LillicrapHPHETS15

[17] L. Fan, J. Zhang, Y. He, Y. Liu, T. Hu, and H. Zhang, "Optimal scheduling of microgrid based on deep deterministic policy gradient and transfer learning," *Energies*, vol. 14, p. 584, 01 2021.

[18] O. Klimov, "Gymnasium's lunar lander environment documentation," Gymnasium Documentation, 2022. [Online]. Available: https://gymnasium.farama.org/environments/box2d/lunar_lander/