Reinforcement Learning Algorithms: An Overview and Classification

Fadi AlMahamid (iD), *Senior Member, IEEE*, and Katarina Grolinger (iD), *Member, IEEE*
Department of Electrical and Computer Engineering
Western University
London, Ontario, Canada

Abstract—The desire to make applications and machines more intelligent and the aspiration to enable their operation without human interaction have been driving innovations in neural networks, deep learning, and other machine learning techniques. Although reinforcement learning has been primarily used in video games, recent advancements and the development of diverse and powerful reinforcement algorithms have enabled the reinforcement learning community to move from playing video games to solving complex real-life problems in autonomous systems such as self-driving cars, delivery drones, and automated robotics. Understanding the environment of an application and the algorithms' limitations plays a vital role in selecting the appropriate reinforcement learning algorithm that successfully solves the problem on hand in an efficient manner. Consequently, in this study, we identify three main environment types and classify reinforcement learning algorithms according to those environment types. Moreover, within each category, we identify relationships between algorithms. The overview of each algorithm provides insight into the algorithms' foundations and reviews similarities and differences among algorithms. This study provides a perspective on the field and helps practitioners and researchers to select the appropriate algorithm for their use case.

I. INTRODUCTION

Reinforcement Learning (RL) is one of the three machine learning paradigms besides supervised learning and unsupervised learning. It uses agents acting as human experts in a domain to take actions. RL does not require data with labels; instead, it learns from experiences by interacting with the environment, observing, and responding to results.

RL can be expressed with Markov Decision Process (MDP) as shown in Figure 1. Each environment is represented with a state that reflects what is happening in the environment. The RL agent takes actions in the environment, that causes a change in the environment's current state generating a new state and receives a reward based on the results. The agent receives a positive reward for good actions and a negative reward for bad actions, which helps the agent evaluate the performed action in a given state and learn from experiences.

Video games have been one of the most popular RL applications, and RL algorithms have been mainly tested and evaluated on video games. However, RL has other applications and can be used in different domains such as self-driving cars, natural language processing (NLP), autonomous robotics, delivery drones, and many others. Furthermore, there are many
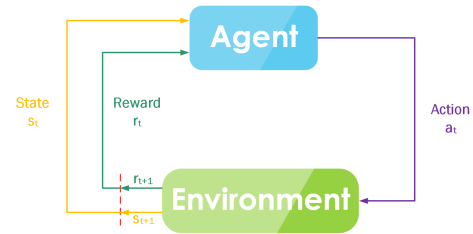
Fig. 1. Markov Decision Process

diverse RL algorithms with different variations. Therefore, it is imperative to understand the differences between RL algorithms, select the appropriate algorithm suitable for the environment type and the task on hand.

The most widely used algorithm is Deep Q-Network (DQN) with its variations because of its simplicity and efficiency. Nevertheless, DQN is suitable only for environments with discrete actions. For example, in autonomous UAV navigation (self-flying drones), many papers tend to simplify the environment to enable the use of DQN [1], [2]. However, in complex real-life environments, DQN would not be suitable if the environment is dynamic or the required actions are continuous. Therefore, to assist in matching the RL algorithm with the task, the classification of RL algorithms based on the environment type is needed.

Consequently, this study provides an overview of different RL algorithms, classifies them based on the environment type, and explains their primary principles and characteristics. Additionally, relationships among different RL algorithms are also identified and described. The paper provides a perspective on the domain and helps researchers and practitioners to select appropriate algorithms for their use cases. Moreover, it provides options for selecting a suitable algorithm for the environment, rather than attempting to simplify the environment for the algorithm [1], [2].

The remainder of the paper is organized as follows: Section II introduces RL and discusses RL's main principles. Section III classifies RL algorithm and provides their overview. Finally, Section IV concludes the paper.

II. BACKGROUND

This section first introduces reinforcement learning. Next, concepts of policy and value functions are described, and

finally, experience replay is explained as a commonly used technique in different RL algorithms.

A. Reinforcement Learning

The RL agent learns from taking actions in the environment, which causes a change in the environment's current state and generates a reward based on the action taken as expressed in the Markov Decision Process (MDP). We define the probability of the transition to state s' with reward r from taking action a in state s at time t, for all $s' \in S$, $s \in S$, $r \in R$, $a \in A(s)$, as:

$$P(s', r|s, a) = Pr\{S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a\} \quad (1)$$

The agent receives rewards for performing actions and uses them to measure the action's success or failure. The Reward R can be expressed in different forms, as a function of the action $R(a)$, or as a function of action-state pairs $R(a, s)$.

The agent's objective is to maximize the expected summation of the discounted rewards, which drives the agent to take the selected actions. The reward is granted by adding all the rewards generated from executing an episode. The *episode* (trajectory) represents a finite number of actions and ends when the agent achieves a final state, for example, when a collision occurs in a simulated navigation environment. However, in some cases, the actions can be continuous and cannot be broken into episodes. The discounted reward, as shown in equation 2 uses a multiplier γ to the power k, where $\gamma \in [0, 1]$. The value of k increases by one at each time step to emphasize the current reward and to reduce the impact of the future rewards, hence the term discounted reward.

$$G_t = E\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right] \quad (2)$$

Emphasizing the current action's immediate reward and reducing the impact of future actions' rewards help the expected summation of discounted rewards to converge.

B. Policy and Value Function

The agent's behavior is defined by following a policy π, where the policy π defines the probability of taking action a, given a state s, which is denoted as $\pi(a|s)$. Once the agent takes an action, the agent uses a value function to evaluate the action. The agent either uses: 1) a *state-value function* to estimate how good for the agent to be in state s, or 2) a *action-value function* to measure how good it is for the agent to perform an action a in a given state s. The action-value function is defined in terms of the expected summation of the discounted rewards and represents the target Q-value:

$$Q_\pi(s, a) = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right] \quad (3)$$

The agent performs the action with the highest Q-value, which might not be the optimal Q-value. Finding the optimal Q-value requires selecting the best actions that maximize the expected summation of discounted rewards under the optimal policy π. The optimal Q-value $Q_*(s, a)$ as described in equation 4 must satisfy the *Bellman optimality equation*, which is equal to the expected reward R_{t+1}, plus the maximum expected discounted return that can be achieved for any possible next state-action pairs (s', a').

$$Q_*(s, a) = \max_{\pi} Q(s, a) \quad (4)$$

$$Q_*(s, a) = E\left[R_{t+1} + \gamma \max_{a'} Q_*(s', a')\right] \quad (5)$$

This optimal Q-value $Q_*(s, a)$ is used to train the neural network. The Q-value $Q(s, a)$ predicted by the network is subtracted from the optimal Q-value $Q_*(s, a)$ estimated using the Bellman equation and backpropagated through the network. The loss function is defined as follows:

$$\overbrace{E\left[R_{t+1} + \gamma \max_{a'} Q_*(s', a')\right]}^{Target} - \overbrace{E\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right]}^{Predicted} \quad (6)$$

C. Experience Replay

In RL, an experience e can be described as the knowledge produced from the agent performing an action a in a state s causing a new state s' and a generated reward r. The experience can be expressed as a tuple $e(s, a, s', r)$. Lin [3] proposed a technique called *Experience Replay*, where experiences are stored in a replay memory D and used to train the agent. Since experiences are stored in the memory, and some experiences might be of a high importance, they can repeatedly be reused to train the agent what improves convergence.

Although experience replay should help the agent theoretically learn from previous important experiences, it entails sampling experiences uniformly from the replay memory D regardless of their significance. Schaul *et al.* [4] suggested the use of *Prioritized Experience Replay*, which aims to prioritize experiences using Temporal Difference error (TD-error) and replay more frequently experiences that have lower TD-error.

III. REINFORCEMENT LEARNING ALGORITHMS CLASSIFICATION

While most reinforcement learning algorithms use deep neural networks, different algorithms are suited for different environment types. We classify RL algorithms according to the number of the states and action types available in the environment into three main categories: 1) a limited number of states and discrete actions, 2) an unlimited number of states and discrete actions, and 3) an unlimited number of states and continuous actions. The three categories, together with algorithms belonging to those categories, are shown in Figure 2 and discussed in the following subsections.
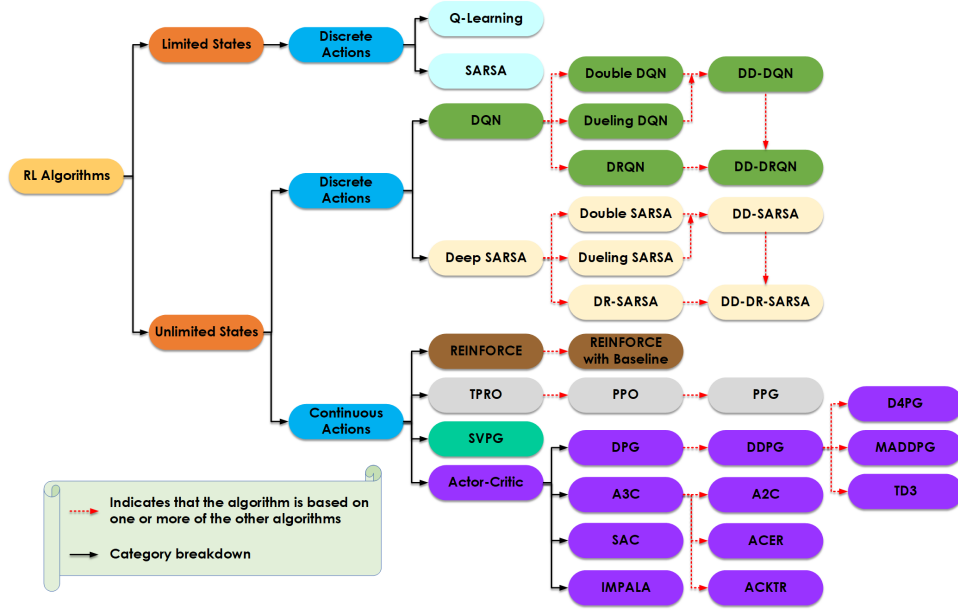
Fig. 2. Reinforcement Algorithms classification based on the environment type

A. Environments with Limited States and Discrete Actions

The environments with discrete actions and limited states are relatively simple environments where the agent can select from pre-defined actions and be in pre-defined known states. For example, when an agent is playing a tic-tac-toe game, the nine boxes represent the states, and the agent can choose from two actions: X or O, and update the available states.

Q-Learning [5] algorithm is commonly used to solve problems in such environments. This algorithm finds the optimal policy in a Markov Decision Process (MDP) by maintaining a Q-Table table with all possible states and actions and iteratively updating the Q-values for each state-action pair using the Bellman equation until the Q-function converges to the optimal Q-value.

State–Action–Reward–State–Action (SARSA) [6] is another algorithm from this category: it is similar to Q-learning except it updates the current $Q(s, a)$ value in a different way. In Q-learning, in order to update the current $Q(s, a)$ value, we need to compute the next state-action $Q(s', a')$ value, and since the next action is unknown, then Q-learning takes a greedy action to maximize the reward [7]. In contrast, when SARSA updates the current state-action $Q(s, a)$ value, it performs the next action a' [7].

B. Environments with Unlimited States and Discrete Actions

In some environments, such as playing a complex game, the states can be limitless; however, the agent's choice is limited to a finite set of actions. In such environments, the agent mainly consists of a Deep Neural Network (DNN), usually a Convolutional Neural Network (CNN), responsible for processing and extracting features from the state of the environment and outputting the available actions. Different
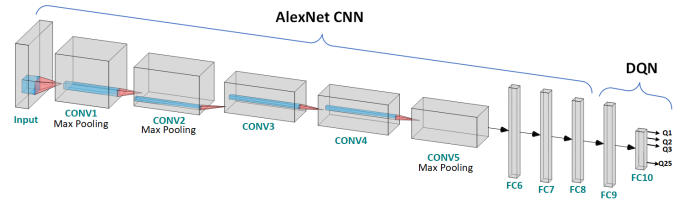

Fig. 3. DQN using AlexNet CNN

algorithms can be used with this environment type, such as Deep Q-Networks (DQN), Deep SARA, and their variants.

1) Deep Q-Networks (DQN):

Deep Q-Learning, also referred to as Deep Q-Networks (DQN), is considered the main algorithm used in environments with unlimited states and discrete actions, and it inspires other algorithms used for a similar purpose. DQN usually combines convolutional and pooling layers, followed by fully connected layers that produce Q-values corresponding to the number of actions. Figure 3 [8] shows AlexNet CNN followed by two fully connected layers to produce Q-value. The current scene from the environment represents the environment's current state; once it is passed to the network, it produces Q-value representing the best action to take. The agent acts and then captures the changes in the environment's current state and the reward generated from the action.

A significant drawback of the DQN algorithm is overestimating the action-value (Q-value), where the agent tends to choose a non-optimal action because it has the highest Q-value [9].

a) Double and Dueling DQN (DD-DQN):

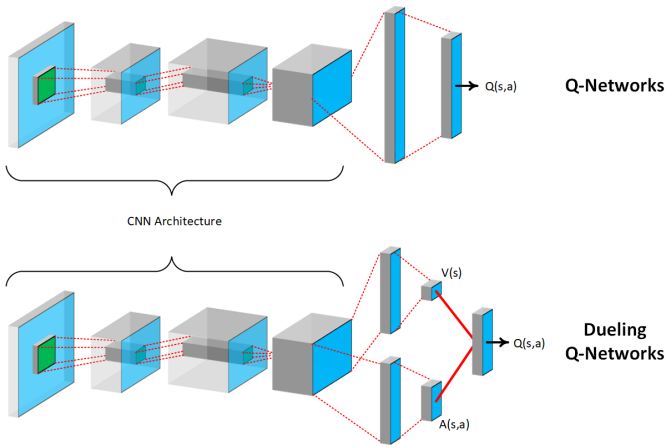Double DQN uses two networks to solve this overestimation

Fig. 4. DQN vs. Dueling DQN

problem in DQN. The first network, called the Policy Network, optimizes the Q-value, and the second network, the Target Network, is a replica of the policy network, and it is used to produce the estimated Q-value [10]. The target network parameters are updated after a certain number of time steps by copying the policy network parameters rather than using the backpropagation.

Another improvement on DQN is Dueling DQN illustrated in Figure 4 [11]. Dueling DQN tries to define a better way to evaluate the Q-value by explicitly decomposing the Q-value function into two functions:

- State-Value function $V(s)$ measures how good is for the agent to be in state s.
- Advantage-Value function $A(s, a)$ captures how good is an action compared to other actions at a given state.

The two functions shown in Figure 4 [11], are combined via a special aggregation layer to produce an estimate of the state-action value function [11]. The value of this function is equal to the summation of the two values produced by the two functions:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a)\right) \quad (7)$$

The subtracted term $\frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a)$ represents the mean, where $|\mathcal{A}|$ represents the size of the vector A. This term helps with identifiability, and it does not change the relative rank of the A (and hence Q) values. Additionally, it increases the stability of the optimization as the advantage function only needs to change as fast as the mean [11].

Double Dueling DQN (DD-DQN) is another DQN algorithm: it combines Dueling DQN with Double DQN to find the optimal Q-value as suggested originally by Wang *et al.* [11] where the output from the Dueling DQN is passed to Double DQN.

b) Deep Recurrent Q-Networks (DRQN):
Deep Recurrent Q-Network (DRQN) [12] is an extension of the DQN algorithm, replacing the first fully connected layer with a recurrent LSTM layer of the same size. Adding the LSTM layer requires changing the input size from a single state of the environment to multiple states (frames) as a single input, which helps to integrate information through time [12].

2) Deep SARSA:
Basic SARSA uses Q learning and is suitable for limited states and discrete actions environments, as described in subsection III-A. On the other hand, Deep SARSA for unlimited states uses a deep neural network similar to DQN: the main difference is that SARSA computes $Q(s', a')$ by performing the next action a', which is required to calculate the current state-action $Q(s, a)$. As shown in Figure 2, extensions of Deep SARSA are the same as extensions of DQN with the main difference on how to calculate the next action-state value $Q(s', a')$.

C. Environments with Unlimited States and Continuous Actions

Although discrete actions are sufficient to move a car or UAV in a virtual environment, such actions do not provide a realistic object movement in real-life scenarios. Continuous actions describe the quantity of movement in different directions where the agent does not choose from a list of predefined actions. For example, a realistic UAV movement specifies the quantity of required change in roll, pitch, yaw, and throttle values to navigate the environment while avoiding obstacles, rather than moving UAV using one step in predefined directions: up, down, left, right, and forward.

Continuous action space requires the agent to learn a parameterized policy π_θ to maximize the expected summation of the discounted rewards because it is impossible to calculate action-value for all continuous actions at different states. The problem is a maximization problem and can be solved using gradient descent algorithms to find the optimal θ. The value of θ is updated as follows:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \quad (8)$$

where α is the learning rate and ∇ is the gradient.

The reward function J objective is to maximize the expected reward using a parameterized policy π_θ as follows [13]:

$$\begin{aligned}
J(\pi_\theta) &= \sum_{s \in S} \rho_{\pi_\theta}(s) \, V^{\pi_\theta}(s) \\
&= \sum_{s \in S} \rho_{\pi_\theta}(s) \sum_{a \in A} Q^{\pi_\theta}(s, a) \, \pi_\theta(a|s)
\end{aligned} \quad (9)$$

Here $\rho_{\pi_\theta}(s)$ defines the stationary probability of π_θ starting from state s_0 and transitioning to future states following the policy π_θ for t time steps. Finding the optimal θ that

maximizes the function $J(\pi_\theta)$ requires finding the gradient $\nabla_\theta J(\theta)$:

$$\nabla_\theta J(\theta) = \nabla_\theta \left(\sum_{s \in S} \rho_{\pi_\theta}(s) \sum_{a \in A} Q^{\pi_\theta}(s,a) \ \pi_\theta(a|s) \right)$$
$$\propto \sum_{s \in S} \mu(s) \sum_{a \in A} Q^{\pi_\theta}(s,a) \ \nabla_\theta \pi_\theta(a|s) \quad (10)$$

Equation 10 can be further rewritten in continuous episodes since $\sum_{s \in S} \eta(s) = 1$ as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s \sim \rho^{\pi_\theta}, a \sim \pi_\theta} \left[Q^{\pi_\theta}(s,a) \ \nabla_\theta \ln \pi_\theta(a_t|s_t) \right] \quad (11)$$

When the training sample is collected according to the target policy $s \sim \rho^{\pi_\theta}$ and the expected return is generated for the same policy π_θ, the algorithm is referred to as *on-policy algorithm*. On the other hand, in *off-policy algorithms*, the training sample follows a behavior policy $\beta(a|s)$, which is different than the target policy $\pi_\theta(a|s)$ [14], while the expected reward is generated using the target policy π_θ. Off-policy algorithms do not require full rejectories (episodes) for the training sample and they can reuse past trajectories. Equation 12 [14] shows how the policy is adjusted to the ratio between the target policy $\pi_\theta(a|s)$ and behaviour policy $\beta(a|s)$.

$$\nabla_\theta J(\theta) = \mathbb{E}_{s \sim \rho^\beta, a \sim \beta} \left[\frac{\pi_\theta(a|s)}{\beta_\theta(a|s)} Q^{\pi_\theta}(s,a) \ \nabla_\theta \ln \pi_\theta(a_t|s_t) \right] \quad (12)$$

The policy gradient theorem shown in equation 9 [15] considered the fundamental base of distinct Policy Gradients (PG) algorithms such as REINFORCE [16], Actor-Critic algorithms [17], Trust Region Policy Optimization (TRPO) [18], and Phasic Policy Gradient [19], Stein Variational Policy Gradient [20], Proximal Policy Optimization (PPO) [21], and many others.

1) REINFORCE:
REINFORCE is an acronym for **RE**ward **I**ncrement = **N**onnegative **F**actor × **O**ffset **R**einforcement × **C**haracteristic **E**ligibility [16]. REINFORCE is a Monte-Carlo policy gradient algorithm that works with the episodic case. It requires a complete episode to obtain a sample proportional to the gradient, and updates the policy parameter θ with the step size α. Because $\mathbb{E}_\pi[G_t|S_t, A_t] = Q^\pi(s,a)$, REINFORCE can be defined as [13]:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[G_t \ \nabla_\theta \ln \pi_\theta(A_t|S_t) \right] \quad (13)$$

REINFORCE uses the Monte Carlo method, which suffers from high variance and, consequently, has slow learning [16]. Adding a baseline to REINFORCE reduces the variance and speeds up learning while keeping the bias unchanged by subtracting the baseline value from the expected return G_t [13].

2) Trust Region Policy Optimization (TPRO):
Trust Region Policy Optimization (TRPO) [18] is a PG algorithm that improves the performance of gradient descent by taking more extensive steps within trust regions defined by a constraint of KL-Divergence and performs the policy update after each trajectory rather than after each state. Proximal Policy Optimization (PPO) [21] can be considered an extension of TRPO; it imposes the constraint as a penalty and clips the objective to ensure that the optimization is carried out within the predefined range [22].

Phasic Policy Gradient (PPG) [23] extends PPO by including a periodic auxiliary phase which distills features from the value function into the policy network to improve training. This auxiliary phase enables feature sharing between the policy and value function while decoupling their training.

3) Stein Variational Policy Gradient (SVPG):
Stein Variational Policy Gradient (SVPG) [20] applies the Stein variational gradient descent (SVGD) [24] to update the policy parameterized by θ, which reduce variance and improves convergence. SVPG improves the average return and data efficiency when used on top of REINFORCE and advantage actor-critic algorithms [20].

4) Actor-Critic:
Actor-Critic algorithms are a set of algorithms based on policy gradients theorem that consist of two components:

1) An Actor responsible for adjusting the parameter θ of the policy π_θ
2) A Critic which employs a parameterized vector w to estimate the value-function $Q^w(s_t, a_t) \approx Q^\pi(s_t, a_t)$ using a policy evaluation algorithm such as temporal-difference learning [14]

The actor can be described as the network trying to find the probability of all available actions and select the action with the highest value, while the critic can be described as a network evaluating the selected action by estimating the value of the new state resulted from performing the action. Different algorithms fall under the actor-critic category; the main ones are described in the following subsections.

a) Deterministic Policy Gradients (DPG) Algorithms:
All deterministic policy gradients algorithms model the policy as a deterministic policy $\mu(s)$, rather than stochastic policy $\pi(s,a)$ that is modeled over the action's probability distribution. We described earlier in Equation 9, the objective function under a selected policy $J(\pi_\theta)$ to be $\sum_{s \in S} \rho_{\pi_\theta}(s) \ V^{\pi_\theta}(s)$; however, a deterministic policy is a special case of stochastic policy, where the objective function of the target policy is averaged over the state distribution of the behaviour policy as described in equation 14 [14].

$$J_\beta(\mu_\theta) = \int_S \rho^\beta(s) \ V^\mu(s) \ ds$$
$$= \int_S \rho^\beta(s) \ Q^\mu(s, \mu_\theta(s)) \ ds \quad (14)$$

In the off-policy approach with a stochastic policy, importance sampling is often used to correct the mismatch between behaviour and target policies. However, because the deterministic policy gradient removes the integral over actions, we can avoid importance sampling and the gradient becomes:

$$
\begin{aligned}
\nabla_\theta J_\beta(\mu_\theta) &\approx \int_S \rho^\beta(s) \, \nabla_\theta \, \mu_\theta(a|s) \, Q^\mu(s,\mu_\theta(s)) \, ds \\
&= \mathbb{E}_{s\sim\rho^\beta} \left[\nabla_\theta \, \mu_\theta(s) \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)} \right]
\end{aligned}
\tag{15}
$$

Different algorithms build on DPG with improvements; for example, Deep Deterministic Policy Gradient (DDPG) [25] adapts DQN to work with continuous action space and combines it with DPG. On the other hand, Distributed Distributional DDPG (D4PG) [26] adopts distributed settings for DDPG with additional improvements such as using N-step returns and prioritized experience replay [26]. Multi-agent DDPG (MADDPG) [27] is another algorithm that extends DDPG to work with multi-agents, where it considers action policies of other agents and learns policies that require multi-agent coordination [27].

Twin Delayed Deep Deterministic (TD3) [28] builds on Double DQN and applies to DDPG to prevent the overestimation of the value function by taking the minimum value between a pair of critics [28].

b) Advantage Actor-Critic (A3C):
Asynchronous Advantage Actor-Critic (A3C) [29] is a policy gradient algorithm that uses multi-threads, also known as agents or workers, for parallel training. Each agent maintains a local policy $\pi_\theta(a_t|s_t)$ and an estimate of the value function $V_\theta(s_t)$. The agent synchronizes its parameters with the global network having the same structure.

The agents work asynchronously, where the value of the network parameters flows in both directions between the agents and the global network. The policy and the value function are updated after t_{max} actions or when a final state is reached [29].

Advantage Actor-Critic (A2C) [29] is another policy gradient algorithm similar to A3C, except it has a coordinator responsible for synchronizing all agents. The coordinator waits for all agents to finish their work either by reaching a final state or by performing t_{max} actions before it updates the policy and the value function in both direction between the agents and the global network.

Actor-Critic with Experience Replay (ACER) is an off-policy actor-critic algorithm with experience replay that uses a single deep neural network to estimate the policy $\pi_\theta(a_t|s_t)$ and the value function $V_{\theta_v}^\pi(s_t)$ [30]. The three main advantages of ACER over A3C [29]: 1) it improves the truncated importance sampling with the bias correction, 2) it uses stochastic dueling network architectures, and 3) it applies a new *trust region policy optimization* method [30].

ACER uses an improved Retrace algorithm as described in Equation 16 [31] by applying truncated importance sampling with bias correction technique and using the value Q^{ret} as

the target value to train the critic by minimizing the L2 error term [30]. In ACER, the gradient $\hat{g}_t^{acer}$ is computed by truncating the importance weights by a constant c, and subtracting $V_{\theta_v}(s_t)$ to reduce variance: this is denoted in Equation 17 [30].

$$
\begin{aligned}
Q^{ret}(s_t,a_t) &= r_t + \gamma\bar{\rho}_{t+1}\big[Q^{ret}(s_{t+1},a_{t+1}) - Q(s_{t+1},a_{t+1})\big] \\
&\quad + \gamma V(s_{t+1})
\end{aligned}
\tag{16}
$$

$$
\begin{aligned}
\hat{g}_t^{acer} &= \bar{\rho}_t \nabla_\theta \ln \pi_\theta(a_t|s_t)\big[Q^{ret}(s_t,a_t) - V_{\theta_v}(s_t)\big] \\
&\quad + \mathop{\mathbb{E}}_{a\sim\pi}\Big(\big[\frac{\rho_t(a)-c}{\rho_t(a)}\big]\nabla_\theta \ln \pi_\theta(a_t|s_t) \\
&\quad \big[Q_{\theta_v}(s_{,t},a_t) - V_{\theta_v}(s_t)\big]\Big)
\end{aligned}
\tag{17}
$$

Actor-Critic using Kronecker-Factored Trust Region (ACKTR) [32] is another extension of A3C [29], which optimizes both the actor and critic by using Kronecker-factored approximation curvature (K-FAC) [33]. It provides an improved computation of the natural gradients by allowing the covariance matrix of the gradient to be efficiently inverted [32].

c) Soft Actor-Critic (SAC):
Soft Actor-Critic (SAC) aims to maximize the expected reward while maximizing the entropy [34]. SAC ameliorates the maximum expected sum of rewards defined through accumulating the reward over states transitions $J(\pi) = \sum_{t=1}^T \mathbb{E}_{s\sim\rho^\pi,a\sim\pi}\big[r(s_t,a_t)\big]$ by adding the expected entropy of the policy over $\rho_\pi(s_t)$ [34]. Equation 18 shows a generalized entropy objective, where the temperature parameter α controls the stochasticity of the optimal policy through defining the relevance of the entropy $\mathcal{H}(\pi(.|s_t))$ term to the reward [34].

$$
J(\pi) = \sum_{t=1}^T \mathbb{E}_{s\sim\rho^\pi,a\sim\pi}\Big[r(s_t,a_t) + \alpha\mathcal{H}(\pi(.|s_t))\Big]
\tag{18}
$$

SAC uses two separate neural networks for the actor and critic, and applies function approximators to estimate a soft Q-function $Q_\theta(s_t,a_t)$ parameterized by θ, a state value function $V_\psi(s_t)$ parameterized by ψ, and an adjustable policy $\pi_\phi(a_t|s_t)$ parameterized by ϕ.

d) Importance Weighted Actor-Learner Architecture (IM-PALA):
Importance Weighted Actor-Learner Architecture (IMPALA) [35] is an off-policy learning algorithm that decouples acting from learning and can be used in two different setups: 1) single learner and 2) multiple synchronous learners.

Using a single learner and multiple actor setup, each actor generates trajectories and sends each trajectory to the learner, and receives the updated policy before starting a new trajectory. The learner learns from the actors simultaneously by saving the received trajectories from the actors in a queue and generating the updated policy. Nevertheless, actors might learn an older model because actors are not aware of each other

and because of the lag between the actors and the learner. To resolve this issue, IMPALA uses a novel v-trace correction method that considers a truncated importance sampling (IS), which is the ratio between the learner policy π and the actor current policy μ. Similarly, in multiple synchronous learners, the policy parameters are distributed across multiple learners that work synchronously through a master learner [35].

IV. CONCLUSION

Deep Reinforcement Learning has shown advancement in solving sophisticated problems in real-life scenarios. The environment type of the application has a vital role in selecting an appropriate RL algorithm that provides good results and performance. In this work, we have identified three environment types based on the number of actions and states: 1) Limited states and discrete actions, 2) Unlimited states and discrete actions, and 3) Unlimited states and continuous actions.

Environments with a limited number of states and limited actions are considered austere environments and can be solved using Q-learning and SARSA. Complex environments have unlimited states representing the environment, and applying the appropriate algorithm depends on the number of actions. If the actions are limited (discrete), the value-based algorithms such as DQN and its variations would be the choice. However, if the actions are continuous, the policy gradient algorithms are appropriate as they can learn a parameterized policy that approximates the solution. This classification helps researchers and practitioners select appropriate RL algorithms for their studies and applications.

Further investigation of algorithms performance in different use case scenarios is needed: the algorithms should be compared in respect to accuracy, convergence, computational resources, and ease of use. Moreover, diverse use cases and requirements should be considered in the evaluation.

REFERENCES

[1] T. Okuyama, T. Gonsalves, and J. Upadhay, "Autonomous driving system based on deep q learnig," in *2018 International Conference on Intelligent Autonomous Systems (ICoIAS)*. IEEE, 2018, pp. 201–205.

[2] S. O. Chishti, S. Riaz, M. BilalZaib, and M. Nauman, "Self-driving cars using cnn and q-learning," in *IEEE 21st International Multi-Topic Conference*, 2018, pp. 1–7.

[3] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine learning*, vol. 8, no. 3-4, pp. 293–321, 1992.

[4] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv:1511.05952*, 2015.

[5] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[6] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*. University of Cambridge, 1994, vol. 37.

[7] D. Zhao, H. Wang, K. Shao, and Y. Zhu, "Deep reinforcement learning with experience replay based on sarsa," in *IEEE Symposium Series on Computational Intelligence*, 2016, pp. 1–6.

[8] A. Anwar and A. Raychowdhury, "Autonomous navigation via deep reinforcement learning for resource constraint edge nodes using transfer learning," *IEEE Access*, vol. 8, pp. 26 549–26 560, 2020.

[9] H. Van Hasselt, "Double Q-learning," *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010*, pp. 1–9, 2010.

[10] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-Learning," *AAAI Conference on Artificial Intelligence*, pp. 2094–2100, 2016.

[11] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Frcitas, "Dueling Network Architectures for Deep Reinforcement Learning," *International Conference on Machine Learning*, vol. 4, no. 9, pp. 2939–2947, 2016.

[12] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," *arXiv:1507.06527*, 2015.

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

[14] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *International conference on machine learning*. PMLR, 2014, pp. 387–395.

[15] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[16] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[17] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Advances in neural information processing systems*, 2000, pp. 1008–1014.

[18] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*, 2015, pp. 1889–1897.

[19] R. Raileanu and R. Fergus, "Decoupling value and policy for generalization in reinforcement learning," *arXiv:2102.10330*, 2021.

[20] Y. Liu, P. Ramachandran, Q. Liu, and J. Peng, "Stein variational policy gradient," *arXiv:1704.02399*, 2017.

[21] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv:1707.06347*, 2017.

[22] S.-Y. Shin, Y.-W. Kang, and Y.-G. Kim, "Obstacle avoidance drone by deep reinforcement learning and its racing with human pilot," *Applied Sciences*, vol. 9, no. 24, p. 5571, 2019.

[23] K. Cobbe, J. Hilton, O. Klimov, and J. Schulman, "Phasic policy gradient," *arXiv:2009.04416*, 2020.

[24] Q. Liu and D. Wang, "Stein variational gradient descent: A general purpose bayesian inference algorithm," *arXiv:1608.04471*, 2016.

[25] 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:1509.02971*, 2015.

[26] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. Tb, A. Muldal, N. Heess, and T. Lillicrap, "Distributed distributional deterministic policy gradients," *arXiv:1804.08617*, 2018.

[27] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," *arXiv:1706.02275*, 2017.

[28] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International Conference on Machine Learning*. PMLR, 2018, pp. 1587–1596.

[29] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, 2016, pp. 1928–1937.

[30] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," *arXiv:1611.01224*, 2016.

[31] R. Munos, T. Stepleton, A. Harutyunyan, and M. G. Bellemare, "Safe and efficient off-policy reinforcement learning," *arXiv:1606.02647*, 2016.

[32] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba, "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation," in *Advances in neural information processing systems*, 2017, pp. 5279–5288.

[33] J. Martens and R. Grosse, "Optimizing neural networks with kronecker-factored approximate curvature," in *International conference on machine learning*. PMLR, 2015, pp. 2408–2417.

[34] 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.

[35] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward *et al.*, "IMPALA: Scalable distributed deep-rl with importance weighted actor-learner architectures," in *International Conference on Machine Learning*. PMLR, 2018, pp. 1407–1416.