Control-Optimized Deep Reinforcement Learning for Artificially Intelligent Autonomous Systems

Oren Fivel[1], Matan Rudman[1], Kobi Cohen[1]

Abstract

Deep reinforcement learning has become a powerful tool for complex decision-making in machine learning and AI. However, traditional methods often assume perfect action execution, overlooking the uncertainties and deviations between an agent's selected actions and the actual system response. In real-world engineering applications—such as robotics, mechatronics, and communication networks—execution mismatches arising from system dynamics, hardware constraints, and latency can significantly degrade performance.

This work advances AI by developing a novel control-optimized deep reinforcement learning framework that explicitly models and compensates for action execution mismatches, a challenge largely overlooked in existing methods. Our approach establishes a structured two-stage process: determining the desired action (e.g., force or torque) and selecting the appropriate control signal (e.g., voltage) to ensure proper execution. It trains the agent while accounting for action mismatches and controller corrections. By incorporating these factors into the training process, the AI agent optimizes the desired action with respect to both the actual control signal and the intended outcome, explicitly considering execution errors. This approach enhances robustness, ensuring that decision-making remains effective under real-world uncertainties.

Our approach offers a substantial advancement for engineering practice by bridging the gap between idealized learning and real-world implementation. It equips intelligent agents operating in engineering environments with the ability to anticipate and adjust for actuation errors and system disturbances during training. We evaluate the framework in five widely used open-source mechanical simulation environments we restructured and developed to reflect real-world operating conditions, showcasing its robustness against uncertainties and offering a highly practical and efficient solution for control-oriented applications.

***Keywords*—** artificial intelligence, autonomous systems, decision making, deep reinforcement learning, optimization and control.

I. Introduction

Deep Reinforcement Learning (DRL) has become a cornerstone in the field of AI, particularly for solving complex decision-making problems in dynamic and uncertain environments. By enabling agents to learn optimal behaviors through interaction with their surroundings, DRL has shown remarkable success across various

[1]School of Electrical and Computer Engineering, Ben-Gurion University of the Negev, Beer-Sheva, Israel (Email:fivel@post.bgu.ac.il; rudman@post.bgu.ac.il; yakovsec@bgu.ac.il).

Open-source code for the algorithm and simulations developed in this paper can be found on GitHub at [1].

TABLE I

LIST OF ABBREVIATIONS

Abbreviation	Description
AI	Artificial Intelligence
BEMF	Back Electromagnetic Force
CO-DRL	Control-Optimized Deep Reinforcement Learning
D3QN	Dueling Double Deep Q-Network
DDPG	Deep Deterministic Policy Gradient
DNN	Deep Neural Network
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
IRL	Integral Reinforcement Learning
KLD	Kullback–Leibler Divergence
LQR	Linear Quadratic Regulator
LQT	Linear Quadratic Tracker
MAB	Multi-Armed Bandit
NN	Neural Network
PD	Proportional-Derivative
PI	Proportional-Integral
PID	Proportional-Integral-Derivative
PPO	Proximal Policy Optimization
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
SARSA	State–Action–Reward–State–Action

applications, including robotics, autonomous vehicles, and mechatronic systems. These systems often operate under uncertain conditions, where the link between an agent's decisions and the resulting actions can be influenced by factors such as delays, noise, and system dynamics. This makes DRL an ideal approach for tasks requiring adaptive and resilient decision-making. As the demand for intelligent, autonomous systems continues to grow, the ability of DRL to handle real-world complexities makes it a powerful tool for advancing the next generation of autonomous technologies. In this paper, we focus on addressing the challenge of uncertainty in action execution by modeling the dynamics of the agent's actuation system, improving the alignment between desired and actual actions in real-world systems.

A. Contributions

To address the challenge of uncertain and imperfect action execution in real-world AI-driven autonomous systems, we propose Control-Optimized Deep Reinforcement Learning (CO-DRL), a novel DRL framework that integrates control theory to enhance decision-making and execution. By incorporating a feedback loop control mechanism, CO-DRL ensures that intended actions are effectively carried out despite inner system dynamics, disturbances, and model uncertainties. CO-DRL distinguishes between determining the desired action (e.g., force/torque) and selecting the control signal (e.g., voltage) needed to achieve it. By incorporating a feedback

loop control mechanism, our approach optimizes both decision-making and execution, ensuring consistency between intended and realized actions.

Our framework extends OpenAI Gym's classic control environments by incorporating a DC motor model with a Proportional-Integral-Derivative (PID) controller for action tracking. This modular implementation allows flexible customization of the actuation system, control architecture, and DRL algorithms, enabling a seamless combination of decision-making and execution strategies. To support both research and engineering applications, we release an open-source Python implementation of CO-DRL, available on GitHub (see [1]). The software is designed with object-oriented principles, facilitating integration into a variety of control-oriented AI systems.

For researchers and developers, CO-DRL offers a practical toolset for improving the robustness and reliability of DRL-based autonomous systems. By explicitly modeling and compensating for execution mismatches, it enables more stable and predictable policy behavior in complex, real-world settings. The provided environments serve as a testbed for developers and engineers to evaluate and refine DRL policies under realistic conditions, supporting applications in robotics, industrial automation, and other domains where execution fidelity is critical.

We evaluate CO-DRL across five widely used OpenAI Gym environments—Acrobot, CartPole, Mountain Car, Continuous Mountain Car, and Pendulum—chosen for their relevance to classic control tasks and their representativeness of common challenges in engineering systems. To demonstrate the generality and adaptability of our framework, we integrate it with a range of DRL algorithms, including Proximal Policy Optimization (PPO) [2], Deep Q-Networks (DQN) [3], Tile Coding [4], and Deep Deterministic Policy Gradient (DDPG) [5]. Experimental results show that CO-DRL consistently improves performance under conditions of actuation uncertainty and system disturbance. These findings highlight its practical utility and robustness, making it a strong candidate for deployment in AI-driven autonomous systems within engineering domains such as robotics, control systems, and automation.

B. Related Work

RL and particularly DRL algorithms have been implemented as part of a control law design for various autonomous dynamic systems, such as the pendulum (including cartpole and acrobot) [6]–[9], intelligent transportation and autonomous vehicles [10], [11], drones and quadrotors [12], [13], and robotic systems [14]–[16]. These methods have been increasingly integrated into control law design for autonomous systems, with frameworks such as Associative Search Element, Adaptive Critic Element [17], Tile Coding [4], Deep Q-Networks (DQN) [18], and DRL-based proportional-derivative (PD) control [6]. In recent studies, DRL-based methods continue to push the envelope in areas like multi-agent systems and optimization tasks. These DRL frameworks generally optimize decision-making by maximizing rewards, assuming perfect action execution. However, real-world systems, such as those with actuators (e.g., motors), introduce discrepancies between desired and actual actions due to inner dynamics, disturbances, and model uncertainties, as considered in this paper.

Classical RL methods, such as multi-armed bandit (MAB) and tabular Q-learning, have long been used in engineering to solve decision-making problems under uncertainty in diverse areas, such as biomedical

engineering, financial investment, robotics and mechanic systems, and communication networks [19]–[25]. Moreover, reinforcement learning has been applied to other specific classic control problems. For example, [26] applies off-policy integral reinforcement learning (IRL) to solve optimal linear quadratic tracker (LQT) problems for continuous-time two-time-scale processes, addressing both slow and fast states with linear quadratic regulator (LQR) and LQT. In [27], the authors investigate an online reinforcement Q-learning algorithm to design a model-free H_∞ tracing controller for unknown discrete-time linear systems, which is demonstrated in a simulation of a single-phase voltage-source UPS inverter. Additionally, [28] explores a reduced-dimensional reinforcement learning approach for linear time-invariant singularly perturbed systems, focusing on the separation of slow and fast states, with the action determined by LQR control subject to the slow states.

While classic approaches are effective mainly in structured environments with low-dimensional state and action spaces, as engineering systems have grown in complexity, DRL has emerged as a powerful extension, combining the strengths of RL with deep learning to handle complex, unstructured problems. This advancement has led to significant progress across a wide range of engineering domains. In AI-based communication networks, DRL has been employed for dynamic spectrum access, resource management, and cognitive radio [29]–[37]. In the context of computer vision, it has been used for image categorization [38], automated classification [39], and intelligent expert-aided classification [40], [41]. Additionally, DRL methods have been adopted for AI-based anomaly detection and active hypothesis testing [42]–[47], as well as health-related applications that require adaptive and robust decision-making [39], [41], [48]. These diverse applications demonstrate the adaptability of DRL in solving complex, real-world problems beyond classical control, further highlighting its growing role in modern engineering systems.

However, despite these advancements, existing DRL-based controllers often lack explicit mechanisms to track and correct execution mismatches caused by actuator inner dynamics and external disturbances. In this work, we address this gap by developing CO-DRL, a novel framework that optimizes both decision-making and execution during training. This approach ensures consistency between intended and realized actions, enhancing robust decision-making and execution in real-world autonomous systems.

II. Methodology

We begin with a formal introduction of the system model and problem statement, followed by a presentation of the proposed control-optimized DRL framework designed to address the problem.

A. System Model and Problem Statement

In a mechanical system environment, the state space $\mathcal{S}$ consists of the continuous set of positions (both linear and angular) and their corresponding velocities for the bodies and particles within the system, determined by its degrees of freedom. Let $s_t \in \mathcal{S}$ represent the system's state at discrete time t.

An agent in the mechanical system corresponds to the actuation component that applies external forces or torques to certain bodies within the system. Let $\mathcal{A}$ denote the action space available to the agent, with $a_t \in \mathcal{A}$

representing an action taken at time t. The action space $\mathcal{A}$ can be either continuous or discrete. Fig. 1 illustrates a basic RL framework.

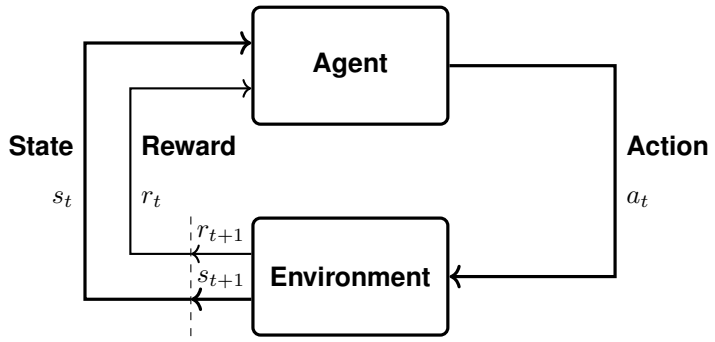


Fig. 1. An illustration of a basic reinforcement learning framework.

At each iteration, the state s_t and reward r_t are provided to the Agent block. The agent selects an action aimed at maximizing the future accumulated reward. This action is then passed to the Environment block, which generates the next state s_{t+1} and the corresponding reward r_{t+1}.

In practical mechanical systems, each action a_t is mapped to an external force/torque as follows:

$$F_t = \texttt{ActionFeature}(a_t) = F\{a_t\}, \tag{1}$$

where $\texttt{ActionFeature}(\cdot)$ (or $F\{\cdot\}$ for brevity) is the function that maps an action a_t to an external force/torque F_t. For a continuous action space, it is often convenient to define the mapping as a simple linear relationship, where the action directly represents the applied force/torque: $F_t = a_t$. In the discrete case, it is often convenient to define the action space as a finite set of indexed actions $\{0, 1, \ldots, N-1\}$, where each index k corresponds to a predefined external force/torque value $F\{k\} \in \mathbb{R}$.

At each time t, a new state s_{t+1} is generated by

$$s_{t+1} = \texttt{EnvStateUpdate}(s_t, F_t), \tag{2}$$

where $\texttt{EnvStateUpdate}(s_t, F_t)$ is a function of the current state s_t and the force/torque $F_t = F\{a_t\}$ associated with the current action a_t. Let p_t and v_t denote the position and velocity of the mechanical object, respectively. Applying Newton's Second Law, a typical next-state update is performed as follows:

$$v_{t+1} = v_t + \texttt{Acceleration}(s_t, F_t)\Delta t, \tag{3}$$

$$p_{t+1} = p_t + v_t \Delta t, \tag{4}$$

where $\texttt{Acceleration}(\cdot,\cdot)$ is a function of the current state s_t and the force/torque F_t, representing the acceleration of the bodies, and Δt is the integration time step. The following integration methods are used in the five openAI Gym environments analyzed in this work. In MountainCar (both discrete and continuous), the next position p_{t+1} is updated using semi-implicit Euler integration, meaning it is computed based on the next velocity v_{t+1} rather than the current velocity v_t. In Pendulum, the next position is updated via regular Euler integration, using the current velocity v_t. In CartPole, either regular Euler integration or semi-implicit Euler integration can be selected. In Acrobot, integration is performed using the Runge-Kutta method.

After the state update, the next reward r_{t+1} is updated as follows:

$$r_{t+1} = \texttt{Reward}(s_{t+1}, s_t, F_t), \tag{5}$$

where `Reward` is a function of the updated state s_{t+1}, the pre-update state s_t, and the applied force/torque F_t corresponding to the action. The reward function can take various forms, such as quadratic or boolean. The next observations o_{t+1} are given by:

$$o_{t+1} = \texttt{getObs}(s_{t+1}, s_t, F_t). \tag{6}$$

In general, the observations o_t and the states s_t are not identical. For instance, in Pendulum, the state consists of the pendulum's angle and angular velocity (dimension $= 2$), whereas the observation represents the angle using its sine and cosine components instead (dimension $= 3$).

The objective of the algorithm is to enable an agent to make effective sequential decisions in a dynamic environment. The DRL algorithm receives observations of the environment and, in some cases, the underlying state, along with the corresponding rewards. Based on this information, the algorithm outputs an action to be executed. The goal is to solve the decision-making problem by learning an optimal policy that adapts to the environment's dynamics and selects actions that maximize cumulative rewards over time. The learning is done by training a deep neural network (DNN) to approximate the value function or the policy, which is updated iteratively based on the agent's interactions with the environment, allowing it to improve decision-making over time.

Unfortunately, the design and implementation of DRL algorithms for applying the optimal force/torque as an intended action in Gym's classic control environments do not account for the agent's uncertainty caused by the actuator's internal dynamics. Our approach is to generalize the action mechanism so that the force/torque in (1) incorporates the uncertainty introduced by the inner dynamics of an actuation system, such as a DC motor. Additionally, we extend the DRL block diagram in Fig. 1 to be based on a closed-loop tracking control framework, as illustrated Fig. 2.

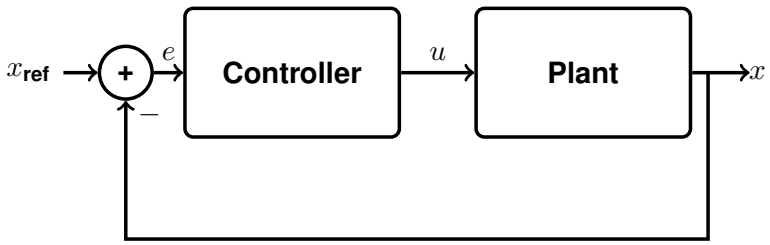


Fig. 2. An illustration of a reinforcement learning framework with closed-loop tracking control considered in this paper.

In control theory notation, the state vector x is fed back and compared with a reference signal x_{ref}. The resulting error, $e = x_{\text{ref}} - x$, is input to the Controller block (analogous to the Agent), which generates a control signal u (corresponding to the Action). This control signal u is then applied to the Plant block (analogous to the Environment), producing a new state x, and the loop continues.

B. The Control-Optimized Deep Reinforcement Learning (CO-DRL) Framework for AI-Driven Autonomous Systems

We model the agent's uncertainty by defining the force/torque in (1) as the *desired* force/torque rather than the actual one. This desired force/torque, along with the actuated body's velocity, is input into a subsystem comprising a DC motor electrical circuit and a PID controller.

1) Integration of DC Motor and PID Controller into the CO-DRL Framework: For simplicity and as a proof of concept, we assume a linear relationship between force and electrical current, as well as between the voltage drop across the DC motor (the BEMF) and the velocity of the body (in m/sec). These assumptions are motivated by fundamental mechanical relationships:

$$\tau = r \times F, \tag{7}$$

$$v = \omega \times r, \tag{8}$$

where τ represents torque, F denotes force, v is the linear velocity, ω is the angular velocity, and r is the arm parameter or the rotation radius. In the DC motor model, the following linear relationships hold:

$$\tau = k_T i, \tag{9}$$

$$V_{BEMF} = k_E \omega, \tag{10}$$

where i represents the electrical current fed into the DC motor, V_{BEMF} is the voltage drop on the DC motor, and k_T and k_E are the torque constant and the BEMF constant, respectively. The discrete time version of the electrical current i_t in a DC motor is given by:

$$i_{t+1} = i_t + \frac{u_t - V_t^{BEMF} - Ri_t}{L}\Delta t \tag{11}$$

where Δt denotes the integration time, u_t the voltage control input, R is the resistor, and L is the inductor. The voltage control input u_t is constructed by a PID controller subject to the electrical current error (defined later in (19)). A typical PID controller in discrete time has the following structure:

$$u_t = k_P e_t + k_I I_{e_t} + k_D D_{e_t}, \tag{12}$$

where

$$I_{e_t} = I_{e_{t-1}} + e_t \Delta t, \tag{13}$$

$$D_{e_t} = \frac{e_t - e_{t-1}}{\Delta t}. \tag{14}$$

Here, I_{e_t} represents the recursive approximation of the integral of the error, D_{e_t} is the approximation of the error's derivative, and k_P, k_I, and k_D are the PID gains for the proportional, integral, and derivative terms, respectively. However, we modify the standard PID control structure by introducing the $k_E\omega$ term to compensate for the BEMF, enabling pre-analysis of the DC electrical circuit without a motor. Additionally, we apply a saturation to the control signal, i.e.,

$$
\begin{aligned}
u_t^{\text{temp}} &= k_P e_t + k_I I_{e_t} + k_D D_{e_t} + k_E \omega, \\
u_t &= \text{sat}(u_t^{\text{temp}}, -u_{\text{max}}, u_{\text{max}}),
\end{aligned}
\tag{15}
$$

where,

$$
\text{sat}(u, u_{\text{min}}, u_{\text{max}}) = \begin{cases} u_{\text{min}}, & u < u_{\text{min}}, \\ u, & u_{\text{min}} \leq u \leq u_{\text{max}}, \\ u_{\text{max}}, & u > u_{\text{max}}. \end{cases}
$$

For each input u_t in (12), a new electrical current i_{t+1} is calculated using (11), which generates the actual torque (9) on the mechanical system. Thus, the actual torque (and force) is assumed to be linearly related to the subsequent electrical current:

$$
F_{t+1} = k_T i_{t+1}. \tag{16}
$$

2) Architecture of the Complete CO-DRL Framework for Autonomous Systems: We now introduce the complete CO-DRL framework for autonomous systems. To define the agent's actions in terms of a desired force/torque, we reformulate (1) as follows:

$$
F_t^{\text{desired}} = \texttt{ActionFeature}(a_t) = F^{\text{desired}}\{a_t\}, \tag{17}
$$

where the superscript *desired* refers to a desired force/torque. Using (9), we compute the desired electrical current as a reference value

$$
i_t^{\text{ref}} = F_t^{\text{desired}} / k_T, \tag{18}
$$

and the error

$$
e_t = i_t^{\text{ref}} - i_t. \tag{19}
$$

To construct the electrical model of the DC motor, we extract the velocity of the actuated body from the current state s_t as it corresponds to the motor speed. Let

$$
w_t = \texttt{ActuatedVelocity}(s_t) \tag{20}
$$

represent the velocity of the actuated body. The error e_t and the actuated velocity w_t are fed into a DC motor sub-environment, which then generates the next actual force/torque based on (11), (13)-(16):

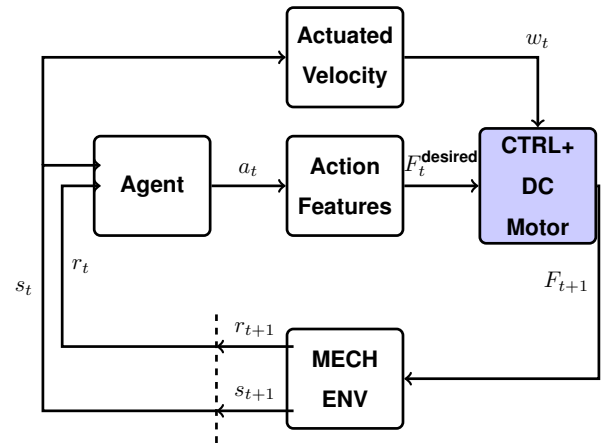
$$
F_{t+1} = \texttt{DCMotorENVStep}(e_t, w_t), \tag{21}
$$

and this computed version of the actual force/torque F_{t+1} is then used in the state update equation (2). Notably, in this formulation, F_{t+1} depends on a_t rather than F_t.

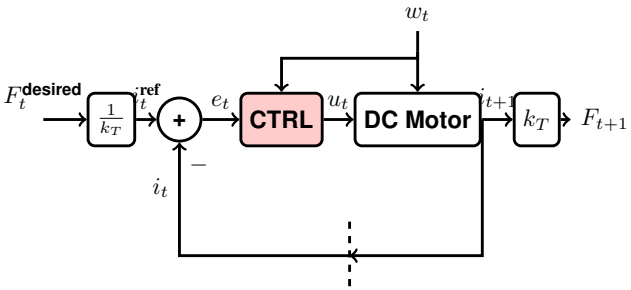
Our proposed approach is illustrated in Fig. 3. Fig. 3(a) illustrates a novel extension of the DRL framework to address uncertainty in action execution by optimizing system operation through key components: *Action Features*, *Actuated Velocity*, and *CTRL+DC Motor*. Specifically, in addition to feeding the current state s_t and reward r_t into the *Agent* component, s_t is also processed by the *Action Features* component, which extracts the velocity associated with the actuated body via the DC motor (20). This velocity is then fed into the *CTRL+DC*

Motor component. The agent's current action a_t is mapped to a desired force/torque via the *Action Features* component (17), which is subsequently passed to *CTRL+DC Motor*. The *CTRL+DC Motor* component is a key enhancement that determines the actual force/torque applied to the main mechanical environment (*MECH ENV*), which in turn generates the next state s_{t+1} (as per (2)) and the corresponding reward r_{t+1} (as per (5)). Further details on the *CTRL+DC Motor* component are provided in Fig. 3(b) and elaborated on in the following discussion.

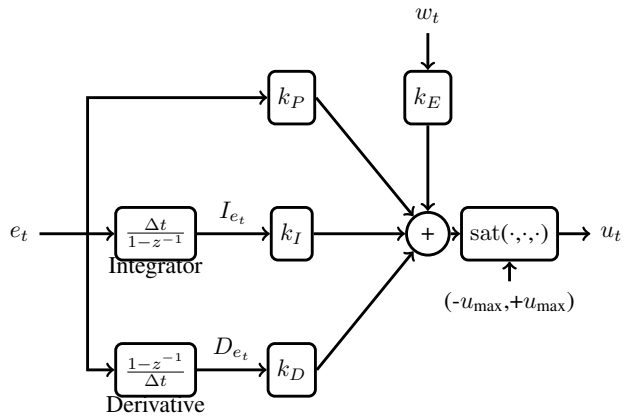
In Fig. 3(b), we illustrate the implementation of the feedback tracking control loop for tracking the applied force/torque. The DC Motor component models the electrical equation for updating the electrical current (11), while the CTRL component governs the control process based on the error between the reference electrical current i_t^{ref} and the actual current i_t (19). The actuated velocity w_t is fed into both the DC Motor and CTRL components, though with distinct roles. When fed into the DC Motor component, w_t influences the motor speed (an internal state) and generates an actual BEMF voltage within the electrical circuit. Conversely, when fed into the CTRL component, w_t is measured (or estimated) and incorporated into the control law by multiplying it by the constant k_E. This artificially generates the BEMF (based on (10)), which is then integrated into the PID control structure (15). Further details on the PID control law structure within the CTRL component are provided in Fig. 3(c) and discussed in the following section.



(a) An illustration of the proposed CO-DRL framework.



(b) An illustration of the CTRL+DC Motor component in the CO-DRL framework.



(c) An illustration of the CTRL component within the CTRL+DC Motor component.

Fig. 3. A block diagram of the proposed CO-DRL framework for autonomous systems.

In Fig. 3(c), we illustrate the CTRL component and the PID control law structure (15), incorporating the time-shift relation in the z-domain:

$$z^{-1}\{x_t\} = x_{t-1}. \tag{22}$$

We represent the integral using the transfer function:

$$\frac{\Delta t}{1 - z^{-1}}, \tag{23}$$

and the derivative by the transfer function:

$$\frac{1-z^{-1}}{\Delta t}. \tag{24}$$

The proportional, integral and derivative parts are multiplied by the coefficients k_P, k_I, and k_D, respectively, and summed with the the term $k_E w_t$. The final control signal u_t is then passed through the Sat component to ensure it remains within predefined bounds.

3) Pseudocode of the CO-DRL Algorithm: The CO-DRL algorithm is summarized in the following pseudocode in Algorithm 1. Let s_0 represent the initial state of the mechanical and original environment, and i_0 the initial electrical current supplied to the DC motor. The algorithm starts at discrete time $t = 0$ and loops until termination, which is indicated by the Boolean variable *terminated*. Termination is determined by the environment's conditions, such as maximum time iterations or the success/failure of meeting specific state and action criteria. These conditions are defined by the Gym library for each environment (prior to our development). At each step, an action a_t is chosen according to the DRL algorithm designed for the environment. This action is then converted into a desired force/torque F_t^{desired} through the ActionFeature. While the ActionFeature is implemented within each Gym environment, we refer to it as the desired force/torque, rather than the actual force/torque. The desired force/torque is converted into a reference electrical current i_t^{ref}, and the error $e_t = i_t^{\text{ref}} - i_t$ is computed. From the current state s_t, the velocity of the actuated body w_t, which is related to the DC motor speed, is obtained through the ActuatedVelocity function. The actual force/torque F_{t+1} is generated based on the error e_t and the velocity w_t by the PID controller and the DC motor electrical equations, as described in the DCMotorENVStep(e_t, w_t) function below the main loop (our new development for each environment). Using the actual force/torque F_{t+1}, we update the state s_{t+1}, reward r_{t+1}, observations o_{t+1}, time step, and the termination flag *terminated*.

Algorithm 1: The CO-DRL Algorithm

Initialization s_0, i_0 ;

t=0;

while *Not terminated* **do**

Choose action: a_t;

Map action to desired force: $F_t^{\text{desired}} = \texttt{ActionFeature}(a_t)$;

Set reference electrical current: $i_t^{\text{ref}} = F_t^{\text{desired}}/k_T$;

Compute error: $e_t = i_t^{\text{ref}} - i_t$;

Extract actuated body velocity: $w_t = \texttt{ActuatedVelocity}(s_t)$;

Get actual force: $F_{t+1} = \texttt{DCMotorENVStep}(e_t, w_t)$;

Update state: $s_{t+1} = \texttt{EnvStateUpdate}(s_t, F_{t+1})$;

Get new reward: $r_{t+1} = \texttt{Reward}(s_{t+1}, s_t, F_{t+1})$;

Get new observations: $o_{t+1} = \texttt{getObs}(s_{t+1}, s_t, F_{t+1})$;

$t = t + 1$;

Update *terminated* flag as necessary;

end

Function `DCMotorENVStep(`e_t, w_t`)`**:**

Get previous error e_{t-1}, and previous error integral $I_{e_{t-1}}$ saved in `DCMotorENV`;

Update integral: $I_{e_t} = I_{e_{t-1}} + e_t \Delta t$;

Update derivative: $D_{e_t} = \frac{e_t - e_{t-1}}{\Delta t}$;

Compute voltage input:

$u_t = \text{sat}(k_P e_t + k_I I_{e_t} + k_D D_{e_t} + k_E w_t, -u_{\text{max}}, u_{\text{max}})$;

Update current: $i_{t+1} = i_t + \frac{u_t - V_t^{BEMF} - R i_t}{L} \Delta t$;

Compute actual force: $F_{t+1} = k_T i_{t+1}$;

return F_{t+1};

4) Open Source Software: For the benefit of researchers and developers in related fields, we developed an open source software implementation of the CO-DRL framework for autonomous systems. Practitioners in related fields are welcome to integrate our implementation in their working environment. Our implementation was developed using Python and is available at GitHub (see link in [1]). The implementation details of the open source software are described in the Appendix.

III. Experimental Results

In this section, we present a comprehensive experimental study to evaluate the performance of the CO-DRL algorithm, which accounts for uncertainties in action execution due to the internal dynamics of AI-driven autonomous systems. We conduct simulations using various fixed values for the PID controller's coefficients, representing a manual tuning approach. These simulations reflect scenarios where the PID controller within the DC motor functions as a black box from the DRL designer's perspective.

Table II provides a summary of all Gym environments used in our implementation, and Fig. 4 provides the illustrations of those Gym environments. It details the following aspects: the type of desired action (continuous

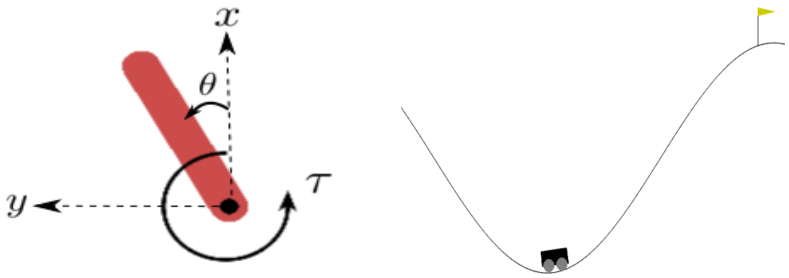
or discrete), the DRL algorithm employed to determine the agent's actions (e.g., DDPG, DQN, etc.), and the Python platform used for implementing the DRL algorithm (e.g., TensorFlow, PyTorch).

The parameters of the DC motor's electrical circuit remain consistent across all environments with the following values: resistance $R = 1\ \Omega$, inductor $L = 0.1$ H, torque constant $k_T = 1$ Volt per meter per second (or radian per second).

TABLE II

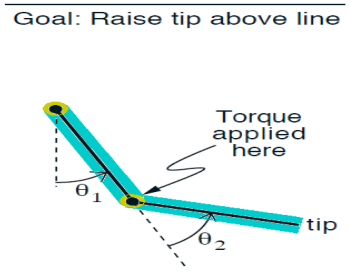
SUMMARY OF IMPLEMENTED GYM'S ENVIRONMENTS

Gym ENV	**Disc./Cont. Action**	**Algorithm**	**Platform**
Pendulum	Continuous	DDPG	TesnsorFlow
Mountain Car (Disc.)	Discrete	Ep. Semi-Grad SARSA +Tile Coding	tile3 [4], [49]
Mountain Car (Cont.)	Continuous	DDPG	Pytorch
Acrobot	Discrete	DQN	Pytorch
Cartpole	Discrete	PPO	TesnsorFlow

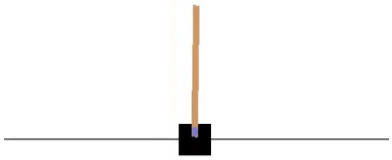


(a) Pendulum environment

(b) Mountain Car environment



(c) Acrobot environment



(d) Cartpole environment

Fig. 4. Illustrations of the OpenAI Gym environments that have been restructured and augmented in this work to capture the complexities of real-world action uncertainties.

A. Solving the Pendulum Environment

In the Pendulum environment, a pendulum is attached to a fixed point at one end, while the other end is free. which is allowed to move along the x-axis, as illustrated in Fig. 4(a).

The goal of this environment is to apply torque to the free end of the pendulum to swing it upright, positioning its center of gravity above the fixed point. The state space consists of the pendulum's angle $\theta \in [-\pi, \pi]$ (radians) and angular velocity $\dot{\theta} \in [-8, 8]$ (rad/sec). The observation space represents the pendulum's position in Cartesian coordinates, where $x = \cos(\theta)$ and $y = \sin(\theta)$, with $x, y \in [-1, 1]$. The action space is a continuous torque input ranging from $[-2, 2]$ Nm. In our approach, action feature extraction is defined by:

$$F_t^{\text{desired}} = \text{sat}(a_t, -\texttt{max\_torque}, +\texttt{max\_torque}), \tag{25}$$

where a_t is the action input, `max_torque`=2 denotes the nominal value of the maximal desired torque to be applied, and F_t^{desired} the result of the desired torque to be fed into the PID control to generate the actual torque F_{t+1}. The reward is defined by:

$$r_{t+1} = -\theta_t^2 - 0.1\dot{\theta_t}^2 - 0.001F_{t+1}^2. \tag{26}$$

In Gym, the reward is updated before the state update. The minimum reward is -16.27, and the maximum is zero (when the pendulum is upright, with zero velocity and no torque). The episode is truncated after 200 time steps, each lasting 0.05 seconds.

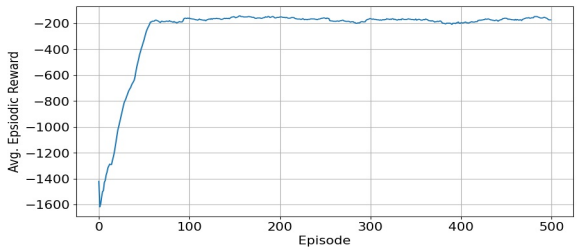
The agent's desired action architecture is DDPG. The actor's neural network (NN) consists of: an input layer (2 units), two hidden layers (64 units each, with Rectified Linear Unit (ReLU) activation), and an output layer (1 unit, with Tanh activation, multiplied by 2). The critic's NN has the following structure: State input layer (2 units), state output layer (1 unit, ReLU), action input layer (1 unit), action output layer (32 units, ReLU), followed by concatenated state and action layers, two hidden layers (256 units each, ReLU), and a final output layer (1 unit). Other parameters include: learning rates $\alpha_{\text{actor}} = 0.001$ and $\alpha_{\text{critic}} = 0.002$, discount factor $\gamma = 0.99$, target NN update parameter $\tau = 0.005$, 500 episodes, and the ADAM optimizer.

The results are shown in Fig. 5. With PID coefficients $K_P = 1$, $K_I = 20$, and $K_D = 1e-6$, the desired torque was tracked almost perfectly (Fig. 5(b)), yielding similar agent performance, reward (Fig. 5(a)), and pendulum observations (Figs. 5(c) to 5(f)) compared to the ideal environment. Within 10 seconds, the agent successfully held the pendulum vertically ($x = 1$, $y, \theta, \dot{\theta} = 0$).

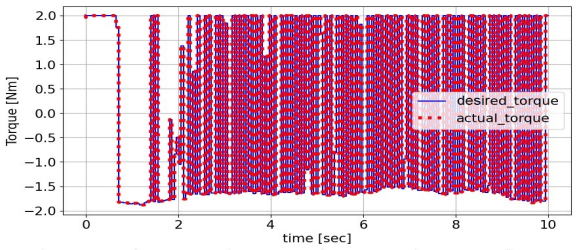
B. Solving the Mountain Car Environment

This subsection covers solutions for both versions of the Mountain Car environment (Fig. 4(b)): the discrete version (Mountain Car in Gym) and the continuous version (Mountain Car Continuous in Gym).

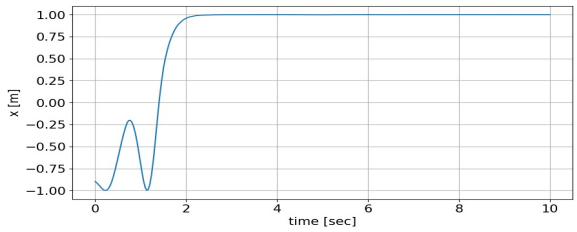
The goal is to drive the car to the top of the right hill, starting from the bottom of the left hill. The state (and observation) space is continuous, including the car's position along the x-axis $x \in [-1.2, 0.6]$ (in meters) and velocity $v \in [-0.07, 0.07]$ (in m/sec). The action space applies a force (in N) between -1 and $+1$. In the continuous version, the action space is $[-1, 1]$, defining the action feature as $F_t^{\text{desired}} = \text{sat}(a_t, -1, +1)$. In the



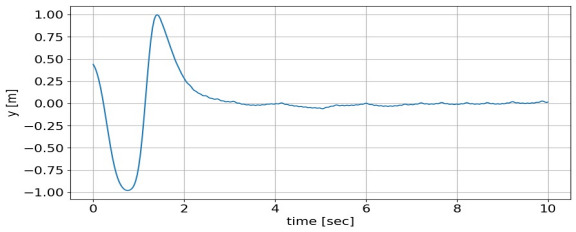
(a) Average Episodic Reward



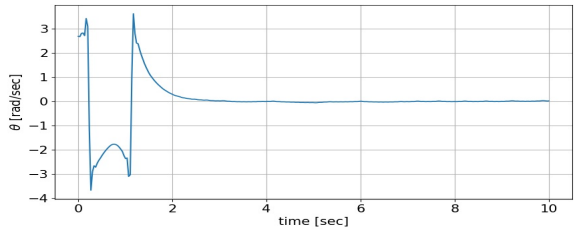
(b) Desired Torque and Actual Torque



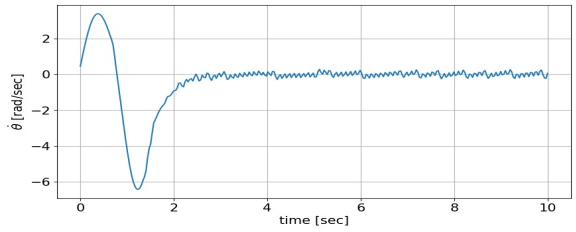
(c) Vertical Position $x = \cos(\theta)$ (Positive Direction: Up)



(d) Horizontal Position $y = \sin(\theta)$ (Positive Direction: Left)



(e) Angle $\theta = \text{atan2}(y, x)$



(f) Angular Velocity $\dot{\theta}$

Fig. 5. Simulation results for the Pendulum environment. Simulation parameters: K_P =1, K_I =20, K_D =1e-6.

discrete version, there are three action indices $a_t \in \{0, 1, 2\}$, mapping to forces $F_t^{\text{desired}} = a_t - 1 \in \{-1, 0, 1\}$. State updates in Gym's Mountain Car environment follow the dynamic equations below [50]:

$$v_{t+1} = v_t + C_f F_{t+1} - C_g \cos(3x_t), \tag{27}$$

$$x_{t+1} = x_t + v_{t+1}. \tag{28}$$

The new velocity v_{t+1} and position x_{t+1} are clipped to $[-0.07, 0.07]$ and $[-1.2, 0.6]$, respectively. The actual applied force is F_{t+1} ($F_{t+1} = F_t^{\text{desired}}$ in the original environment). Constants include C_f for force (`self.force=0.001` in the discrete version, `self.power=0.0015` in the continuous version) and C_g for gravity (`self.gravity=0.0025`). Since position updates (28) mix terms with different units—position in [m] and velocity in [m/sec]—Gym assumes a 1-second integration time, which may be too coarse for state updates and PID control. To refine this, we adopt a continuous formulation using integral equations:

$$v(t + \Delta t) = v(t) + \int_t^{t+\Delta t} \left[C_f F(t') - C_g \cos(3x(t')) \right] dt', \tag{29}$$

$$x(t + \Delta t) = x(t) + \int_t^{t+\Delta t} v(t' + \Delta t) dt'. \tag{30}$$

Integration occurs over the interval $[t, t + \Delta t]$ with $\Delta t = 1$ second (as in the original environment), but the integrals are approximated using a Riemann sum with a finer partition $\Delta t' < 1$ (e.g., 0.05). This approach,

detailed in Algorithm 2, introduces an inner loop. The state s_t updates every Δt, while integration occurs at intervals of $\Delta t' = \Delta t/N$ for some $N \in \mathbb{N}$. Before entering the loop, the desired force F_t^{desired} is set as a reference, and the current state is stored in a temporary variable $s' = s_t$. The loop iterates from 0 to Δt (excluding Δt) with step $\Delta t'$ (similar to `numpy.arange` in Python). Within each iteration, the temporary velocity w', actual force F', and next state s' are updated. The `GetActualForce` function encapsulates force computation, including conversion to reference current, error calculation, PID control, and current-to-force conversion. To keep the algorithm concise, step-by-step details are minimized (see Algorithm 2 for further explanation).

Algorithm 2: Inner Loop Integration for State Update

Initialization s_0, Δt;
$\Delta t' = \Delta t/N$;
t=0;
while *Not terminated* **do**
  Choose action: a_t;
  $F_t^{\text{desired}} = \texttt{ActionFeature}(a_t)$;
  $s' = s_t$;
  for $t' \in$ *arrange(start=0, stop=*Δt*, step=*$\Delta t'$*)* **do**
    $w' = \texttt{ActuatedVelocity}(s')$;
    $F' = \texttt{GetActualForce}(F_t^{\text{desired}}, w')$;
    $s' = \texttt{EnvStateUpdate}(s', F')$;
    /* e.g., s'+=func(s',F')Δ t' */
  end
  $s_{t+1} = s'$;
  $F_{t+1} = F'$;
  $r_{t+1} = \texttt{Reward}(s_{t+1}, s_t, F_{t+1})$;
  $o_{t+1} = \texttt{getObs}(s_{t+1}, s_t, F_{t+1})$;
  $t = t + 1$;
  Update *terminated* flag as necessary;
end

In the discrete Mountain Car environment, the reward is -1 per time step ($\Delta t = 1$ second) with no additional reward for reaching the goal. In the continuous version, the reward is a negative quadratic function of the applied force: $r_{t+1} = -0.1F_{t+1}^2$, where the -0.1 factor follows Gym's original design for consistency. Additionally, a reward of $+100$ is given if the cart reaches the goal. The reward function is defined as:

$$r_{t+1} = 100 \cdot [x_{t+1} \geq x_{\text{Goal}}] - 0.1F_{t+1}^2, \tag{31}$$

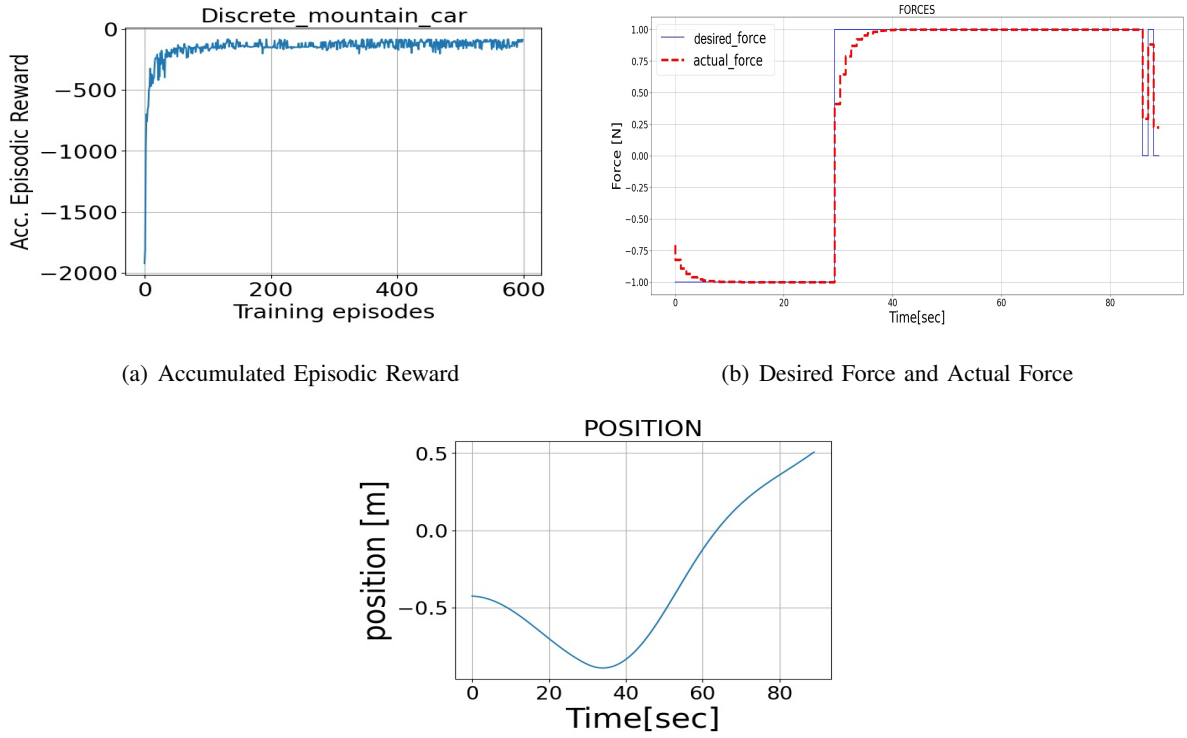
where $x_{\text{Goal}} = 0.45$ is the goal position at the top of the right hill, and $[x_{t+1} \geq x_{\text{Goal}}]$ is a Boolean indicator (1 if true, 0 otherwise).

The episode ends if the car's position exceeds the goal ($x_{\text{Goal}} = 0.45$ in the continuous version, $x_{\text{Goal}} = 0.5$ in the discrete) or if the episode reaches 999 steps (continuous) or 200 steps (discrete).

For the discrete case, the agent's desired action is modeled using Tile Coding and Episodic Semi-Gradient SARSA. Tile Coding parameters include 8 tilings, 3 actions, 2 state variables, and a total of 1,944 tiles, with a learning rate of 0.0375. The Episodic Semi-Gradient SARSA parameters are a discount factor $\gamma = 1.00$, an epsilon-greedy parameter $\varepsilon = 0.001$, and 600 episodes.

For the continuous case, the agent's desired action is modeled using DDPG. The actor network consists of an input layer of size 2 (number of observations), followed by two fully connected hidden layers with 64 units each and ReLU activation, and an output layer with one unit and Tanh activation. The critic network processes the state through an input layer of size 2 and a fully connected output layer with 64 units and ReLU activation. The action is processed through a separate input layer of size 1 before being concatenated with the state output. This is followed by a fully connected hidden layer with 64 units and ReLU activation, and a final output layer with one unit. Other parameters include a learning rate of 1e-3 for the actor and 4e-5 for the critic, a discount factor $\gamma = 0.85$, target network update parameter $\tau = 0.45$, 200 episodes, and the ADAM optimizer.

For the discrete version (Fig. 6), results with $K_P = 1$, $K_I = 1$, and $K_D = 1e - 2$ show that while force tracking (Fig. 6(b)) was imperfect and took time to converge, the car reached the goal in under 85 seconds (Fig. 6(c)), outperforming both the ideal environment (160 sec) and a setup with perfect force tracking (100 sec for $K_P = 2$, $K_I = 10$, $K_D = 1e - 6$). The average reward over the last 50 episodes was approximately -114 (Fig. 6(a)).



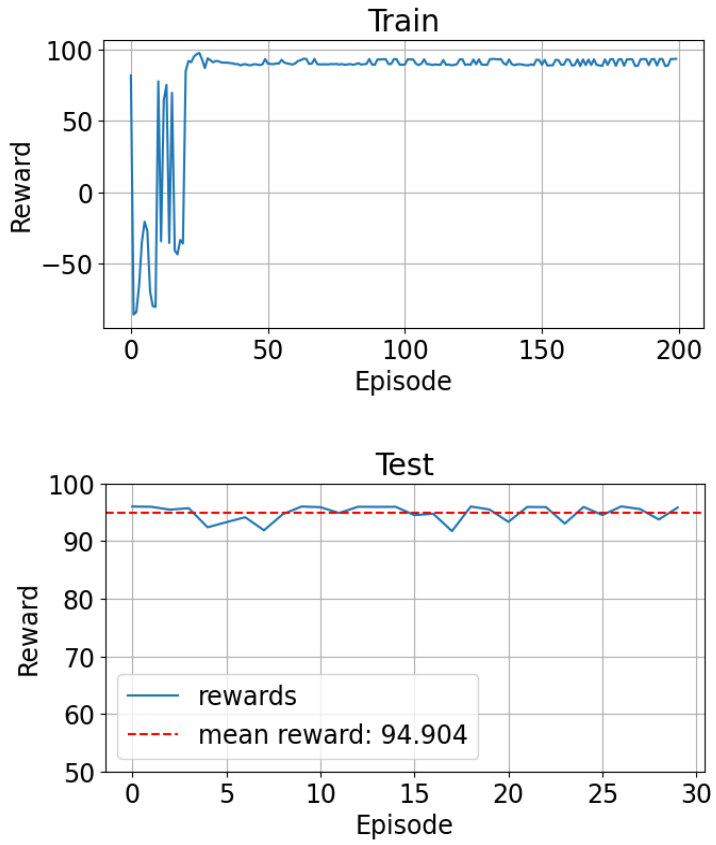
(a) Accumulated Episodic Reward

(b) Desired Force and Actual Force

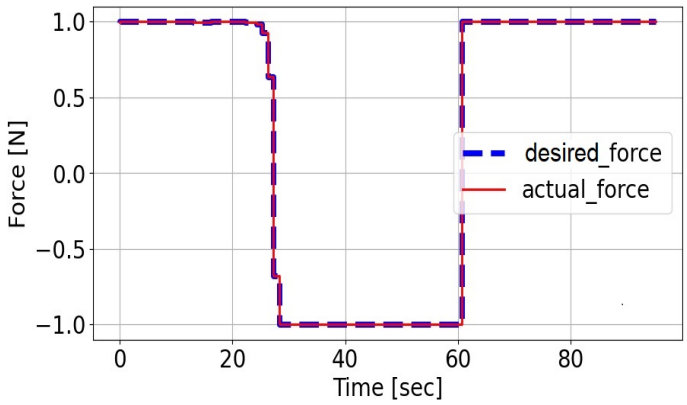
(c) Horizontal Position x

Fig. 6. Simulation results for the Discrete Mountain Car environment. Simulation parameters: K_P =1, K_I =1, K_D =1e-2.

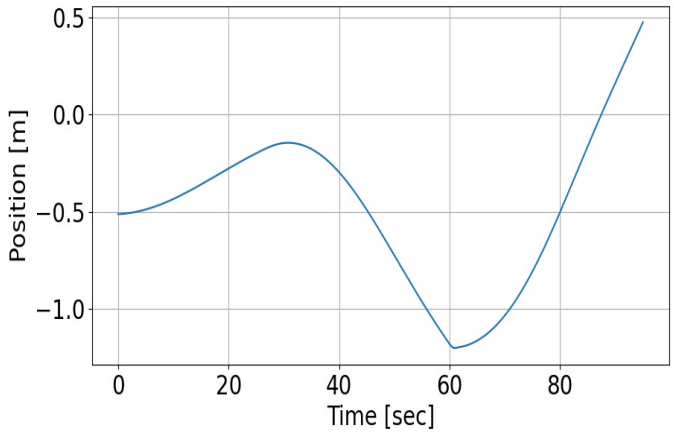
For the continuous version (Fig. 7, using $K_P = 1e-1$, $K_I = 10$, and $K_D = 1e-3$, force tracking was nearly perfect (Fig. 7(b)), reward converged to ~ 95 (Fig. 7(a)), and the car reached the goal in 75 seconds (Fig. 7(c)).



(a) Accumulated Episodic Reward



(b) Desired Force and Actual Force



(c) Horizontal Position x

Fig. 7. Simulation results for the Continuous Mountain Car environment. Simulation parameters: K_P =1e-1, K_I =10, K_D =1e-3.

C. Solving the Acrobot Environment

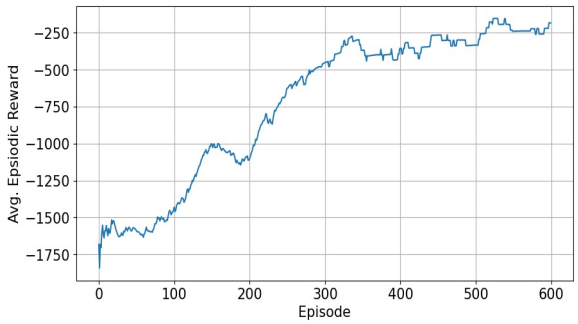
Acrobot Environment (Fig. 4(c)) is a mechanical system consists of two connected links forming a chain with one end fixed. An external torque (in [Nm]) actuates the joint between the links, aiming to swing the free end upright.

The State Space includes the first joint's angle θ_1 and angular velocity $\dot{\theta_1}$, along with the relative angle θ_2 and its velocity $\dot{\theta_2}$. The Observation Space represents angles using $\cos\theta_1$, $\sin\theta_1$, $\cos\theta_2$, and $\sin\theta_2$.

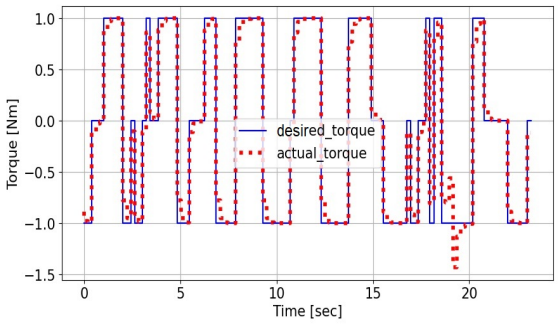
The action space is discrete with three options, $a_t \in \{0, 1, 2\}$, defining the desired torque as $F_t^{\text{desired}} = a_t - 1 \in \{-1, 0, 1\}$. This torque is processed by a PID controller and DC motor to generate a continuous actual torque F_{t+1}. The state updates follow a 4th-order Runge-Kutta method with a 0.2-second interval. We implement a 10-step integration, as in Algorithm 2, similar to the Mountain Car Environment, using a step size of 0.02 seconds.

The agent's desired action is determined using a DQN with a NN comprising an input layer of size 6 (equal to the number of observations), two fully connected hidden layers with 64 and 128 units respectively, both using ReLU activation, and an output layer with 3 units (equal to the number of actions). The model parameters include a learning rate of $\alpha = 1e-3$, a discount factor of $\gamma = 0.99$, 500 training episodes, and the ADAM optimizer.

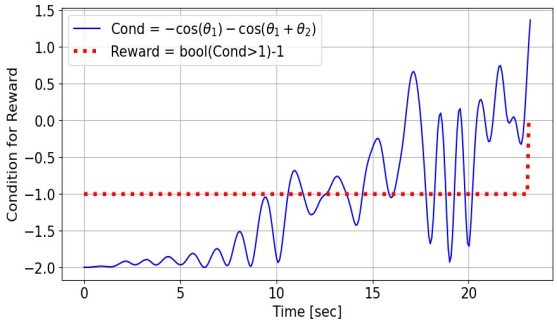
Results are shown in Fig. 8. For $K_P = 1$, $K_I = 10$, and $K_D = 1e-6$, the desired torque is well-tracked (Fig. 8(b)), the reward converges to approximately -200 (Fig. 8(a)), and the Acrobot reaches the goal in about 25 seconds (Fig. 8(c)).



(a) Average Episodic Reward



(b) Desired Torque and Actual Torque



(c) $-\cos(\theta_1) - \cos(\theta_1 + \theta_2)$

Fig. 8. Simulation results for the Acrobot environment. Simulation parameters: K_P =1, K_I =10, K_D =1e-6.

D. Solving the Cartpole Environment

In the Cartpole environment (Fig. 4(d)), a pole is attached to a cart that moves along the x-axis. The state space includes the cart's position x and velocity $\dot{x}$, as well as the pole's angle θ and angular velocity $\dot{\theta}$. At each time step, the agent applies a force of either -10 N (left) or +10 N (right):

$$F_t^{\text{desired}} = \begin{cases} -10, & a_t = 0, \\ +10, & a_t = 1. \end{cases} \tag{32}$$

Although the action space is discrete, mapping to desired forces of ± 10 N, the actual force F_{t+1} from the DC motor is continuous, regulated by a PID controller. The pendulum starts upright, and the objective is to keep it balanced. The agent earns a reward of $+1$ for each time step the pole remains upright. The cart's position is constrained to $[-2.4, 2.4]$ meters, and the pole's angle to $[-0.2095, 0.2095]$ radians (or $[-12°, 12°]$); exceeding these limits results in failure. The reward function is given by:

$$r_{t+1} = \begin{cases} 1, & x_{t+1} \in [-2.4, 2.4] \text{ and } \theta_{t+1} \in [-0.2095, 0.2095] \\ 0, & x_{t+1} \notin [-2.4, 2.4] \text{ or } \theta_{t+1} \notin [-0.2095, 0.2095]. \end{cases} \tag{33}$$

The episode ends when the pole falls ($r_{t+1} = 0$) or remains upright for N time steps—500 in cartpole-v1, 200 in cartpole-v0—each lasting 0.02 seconds. The maximum reward per episode is N.

To evaluate our algorithm, we modified the cartpole environment and PID controller. First, we adjusted the arm length r in the torque-force and velocity relations to keep the desired action signal within a reasonable range. Instead of tracking a torque of ± 10 Nm (with $r = 1$ m), we set $r = 0.15$ m, resulting in a lower torque of ± 1.5 Nm. Second, we reshaped the reward function by penalizing the agent (reducing the reward from 1 to 0) whenever the cart's position exceeded $|x| > 0.1$, without terminating the episode. This encourages the agent to keep the cart near $x = 0$ while avoiding the termination zone at $|x| > 2.4$. Lastly, we introduced a feedforward term $K_{\text{ff}} i_t^{\text{ref}}$ into the control signal u_t to improve tracking of the fast-switching reference current. The control law was modified accordingly, incorporating $K_{\text{ff}} = 0.6$, along with $K_P = 4.3$, $K_I = 1$, and $K_D = 1e{-}6$.

The agent's desired action is determined using PPO-clip with a NN consisting of an input layer of size 4 (number of observations), two fully connected hidden layers with 64 units each and Tanh activation, and an output layer distinguishing between the actor and critic. The actor has two units computing logits, with actions sampled from a categorical distribution based on log probabilities. The critic has one unit estimating the value function. Key parameters include a learning rate of 3×10^{-4} for the actor and 1×10^{-3} for the critic, a discount factor $\gamma = 0.99$, Generalized Advantage Estimation parameter $\lambda = 0.97$, clipping ratio $\epsilon = 0.2$, 60 training epochs, 4000 time steps per epoch, a target Kullback–Leibler Divergence (KLD) of 0.01, and optimization using ADAM.

The learning process runs for 60 epochs with up to 4000 time steps per epoch. A new episode begins upon termination (pole falling) or reaching the time limit, whichever comes first, with policy and value function updates occurring within episodes. Each epoch records the number of episodes, average episodic return, and average episode length, which differ due to the penalty imposed for keeping the cart within $[-0.1, 0.1]$. After

training, the model was tested for 500 time steps (10 seconds) or until termination, with results presented in Fig. 9.

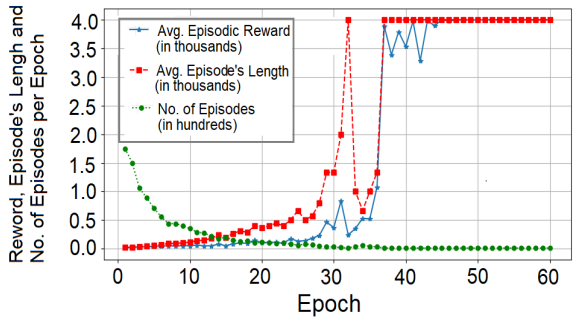
From the learning graphs of average episodic reward, episode length, and number of episodes per epoch (Fig. 9(a)), we observe that initially, there are many episodes as the agent tries to keep the pole upright, with higher rewards and episode lengths. As training progresses, the reward and episode length averages increase up to the 4000 step limit, while the number of episodes decreases to 1. This indicates that the agent has kept the pole upright for the full epoch, though the average reward may not reach 4000. For example, in epoch 32, the episode length jumps to 4000, but the reward averages 240 due to occasional zero rewards when the cart is near the edges of the allowed region. After epoch 35, the average reward approaches 4000 within a single episode. In the action graph (Fig. 9(b)), the actual torque (red) closely tracks the desired torque (blue), with the error (green) bounded within ± 0.1 [Nm]. From the graphs of the pole's angle (Fig. 9(c)) and the cart's position (Fig. 9(d)), we see that the agent successfully kept the cartpole within the required bounds (cart position between [-2.4, 2.4] and pole angle between [-0.2095, 0.2095]). Additionally, through our reward shaping (penalizing the agent with zero reward for cart positions outside [-0.1, 0.1]), the agent was encouraged to keep the cart near the origin rather than an arbitrary point within the range.

IV. Conclusion

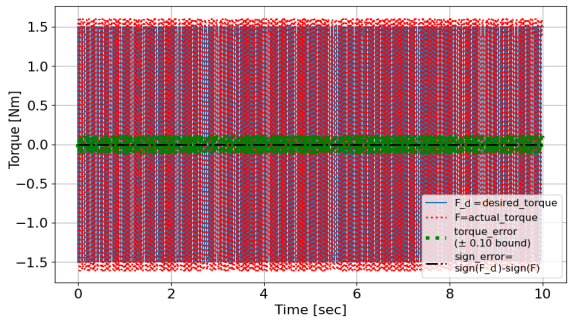
In this research, we introduced a novel DRL framework that incorporates control theory to address uncertainties in action execution. By distinguishing between the Planner, which determines the desired action, and the Controller, which selects the control signal to align execution, our approach optimizes both processes in a unified manner.

Through simulations in various mechanical environments, we demonstrated the framework's robustness in handling execution uncertainties. Integrating control mechanisms into DRL enhances adaptability and performance, particularly in automation and robotics, where precise execution is crucial. Our results highlight the importance of addressing real-world uncertainties to improve the reliability of AI-driven decision-making systems.

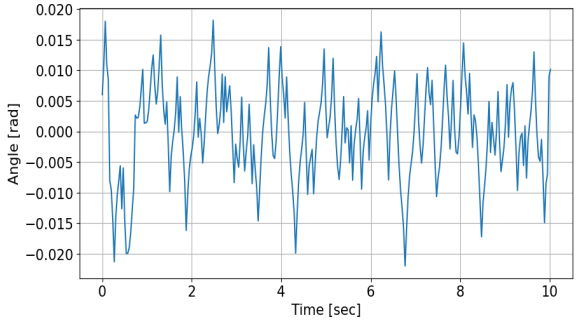
Our findings bridge the gap between high-level decision-making and low-level execution, offering a practical solution for deploying DRL in real-world applications. The open-source implementation provides a valuable resource for practical adoption, and this work has the potential to contribute to the broader success of DRL in automation and control. By enhancing robustness and adaptability, this approach paves the way for more effective deployment of AI-driven systems in dynamic and uncertain environments.



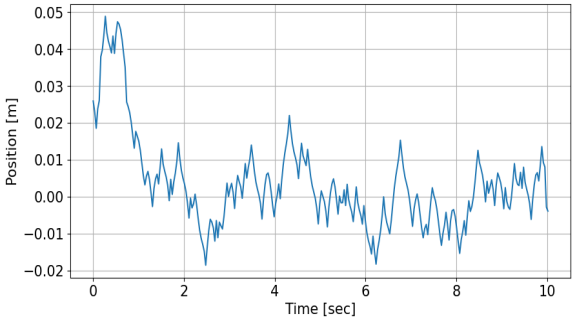
(a) Simulation results of Average Episodic Reward (blue star), Average of Episode's Length (red square) and Number of Episodes (green dot) in each epoch.



(b) Desired Torque and Actual Torque



(c) Pole's angle, θ



(d) Cart's position, x

Fig. 9. Simulation results for the Cartpole environment. Simulation parameters: K_P=4.3, K_I=1 and K_D=1e-6. Additionally, the force-to-torque ratio is 0.15 (torque = 0.15×force), and the control voltage u_t includes an additive feedforward term of $0.6i_t^{\text{ref}}$.

V. Appendix

A. Implementation Details of the Open Source Software

This appendix summarizes the implementation details of our open-source Python software ([1]). For each Gym classic control environment, we provide a Python notebook implementing feedback control on the desired action using a PID controller and a DC motor model (see Fig. 10 for the list of notebooks). Each notebook includes:

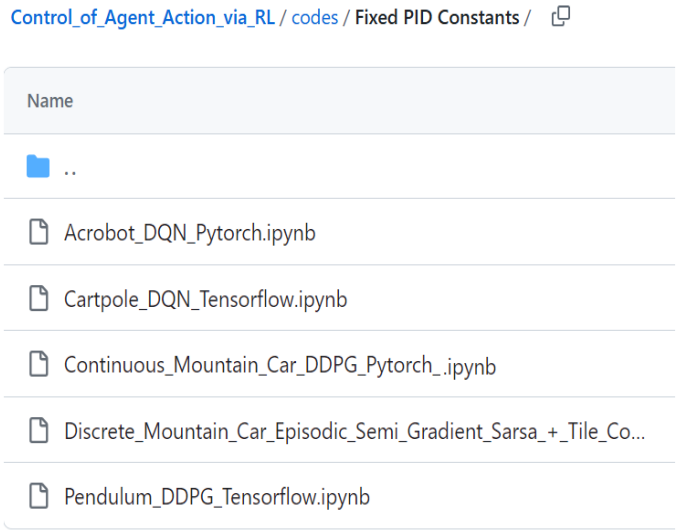


Fig. 10. The project's directory of the feedback control of the desired action in GitHub [1]

1) A customized version of Gym's classic control environment class, incorporating a DC motor model. This model calculates the actual force/torque applied based on the desired input, the actuated body's velocity, and a PID-controlled DC motor field. Users can modify system parameters or adjust reward shaping.

2) A new `ElectricalDCMotorEnv()` class, defining DC motor parameters (e.g., resistor, inductor, torque and voltage constants) and PID control settings. It also offers optional approximations for integral and derivative components via low-pass and high-pass filters. The motor model updates electrical current at each step and outputs the applied force/torque.

3) A DRL implementation for selecting the desired action. For instance, we use DDPG in TensorFlow for the Pendulum environment, but users can employ different frameworks (e.g., PyTorch) or test custom DRL algorithms.

4) Training and test results, showing accumulated rewards over episodes and comparisons of desired vs. actual force/torque, along with state/observation graphs.

REFERENCES

[1] O. Fivel, M. Rudman, and K. Cohen, "Co-drl: Control-optimized deep reinforcement learning." `https://github.com/OrenFivel/Control_of_Agent_Action_via_RL`, 2025. Open-source implementation of CO-DRL (accessed 2025).

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

[3] 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.

[4] R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," *Advances in neural information processing systems*, vol. 8, 1995.

[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] G. Puriel-Gil, W. Yu, and H. Sossa, "Reinforcement learning compensation based PD control for inverted pendulum," in *2018 15th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*, pp. 1–6, IEEE, 2018.

[7] H.-K. Lim, J.-B. Kim, C.-M. Kim, G.-Y. Hwang, H.-b. Choi, and Y.-H. Han, "Federated reinforcement learning for controlling multiple rotary inverted pendulums in edge computing environments," in *2020 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, pp. 463–464, IEEE, 2020.

[8] D. Livne and K. Cohen, "Pops: Policy pruning and shrinking for deep reinforcement learning," *IEEE Journal of Selected Topics in Signal Processing*, vol. 14, no. 4, pp. 789–801, 2020.

[9] P. N. Dao and Y.-C. Liu, "Adaptive reinforcement learning strategy with sliding mode control for unknown and disturbed wheeled inverted pendulum," *International Journal of Control, Automation and Systems*, vol. 19, no. 2, pp. 1139–1150, 2021.

[10] A. Haydari and Y. Yılmaz, "Deep reinforcement learning for intelligent transportation systems: A survey," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 1, pp. 11–32, 2022.

[11] T. Guangwen, L. Mengshan, H. Biyu, Z. Jihong, and G. Lixin, "Achieving accurate trajectory predicting and tracking for autonomous vehicles via reinforcement learning-assisted control approaches," *Engineering Applications of Artificial Intelligence*, vol. 135, p. 108773, 2024.

[12] A. T. Azar, A. Koubaa, N. Ali Mohamed, H. A. Ibrahim, Z. F. Ibrahim, M. Kazim, A. Ammar, B. Benjdira, A. M. Khamis, I. A. Hameed, and G. Casalino, "Drone deep reinforcement learning: A review," *Electronics*, vol. 10, no. 9, p. 999, 2021.

[13] J. Hwangbo, I. Sa, R. Siegwart, and M. Hutter, "Control of a quadrotor with reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 2, no. 4, pp. 2096–2103, 2017.

[14] U. Kartoun, H. Stern, and Y. Edan, "A human-robot collaborative reinforcement learning algorithm," *Journal of Intelligent & Robotic Systems*, vol. 60, pp. 217–239, 2010.

[15] A. Taitler and N. Shimkin, "Learning control for air hockey striking using deep reinforcement learning," in *2017 International Conference on Control, Artificial Intelligence, Robotics & Optimization (ICCAIRO)*, pp. 22–27, IEEE, 2017.

[16] A. Elyasaf, A. Sadon, G. Weiss, and T. Yaacov, "Using behavioural programming with solver, context, and deep reinforcement learning for playing a simplified robocup-type game," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 243–251, IEEE, 2019.

[17] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-13, no. 5, p. 834–846, 1983.

[18] X. Li, H. Liu, and X. Wang, "Solve the inverted pendulum problem base on DQN algorithm," in *2019 Chinese Control And Decision Conference (CCDC)*, pp. 5115–5120, IEEE, 2019.

[19] T. Gafni and K. Cohen, "Learning in restless multiarmed bandits via adaptive arm sequencing rules," *IEEE Transactions on Automatic Control*, vol. 66, no. 10, pp. 5029–5036, 2020.

[20] T. Gafni, M. Yemini, and K. Cohen, "Learning in restless bandits under exogenous global markov process," *IEEE Transactions on Signal Processing*, vol. 70, pp. 5679–5693, 2022.

[21] O. Amar, I. Sarfati, and K. Cohen, "An online learning approach to shortest path and backpressure routing in wireless networks," *IEEE Access*, vol. 11, pp. 57253–57267, 2023.

[22] F. Busacca, S. Palazzo, R. Raftopoulos, and G. Schembra, "Mad-fellows: A multi-armed bandit framework for energy-efficient, low-latency job offloading in robotic networks," in *IEEE INFOCOM 2024-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 1–6, IEEE, 2024.

[23] T. Gafni and K. Cohen, "Distributed learning over markovian fading channels for stable spectrum access," *IEEE Access*, vol. 10, pp. 46652–46669, 2022.

[24] F. Busacca, L. Galluccio, S. Palazzo, A. Panebianco, and R. Raftopoulos, "A distributed multi-armed bandit approach for modulation adaptation in underwater networks," in *2025 IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 1–6, IEEE, 2025.

[25] D. B. Ami, K. Cohen, and Q. Zhao, "Client selection for generalization in accelerated federated learning: A multi-armed bandit approach," *IEEE Access*, vol. 13, pp. 33697–33713, 2025.

[26] W. Xue, J. Fan, V. G. Lopez, Y. Jiang, T. Chai, and F. L. Lewis, "Off-policy reinforcement learning for tracking in continuous-time systems on two time scales," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 10, pp. 4334–4346, 2020.

[27] Y. Peng, Q. Chen, and W. Sun, "Reinforcement Q-learning algorithm for H_∞ tracking control of unknown discrete-time linear systems," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 50, no. 11, pp. 4109–4122, 2020.

[28] S. Mukherjee, H. Bai, and A. Chakrabortty, "Reduced-dimensional reinforcement learning control using singular perturbation approximations," *Automatica*, vol. 126, p. 109451, 2021.

[29] O. Naparstek and K. Cohen, "Deep multi-user reinforcement learning for distributed dynamic spectrum access," *IEEE Transactions on Wireless Communications*, vol. 18, no. 1, pp. 310–323, 2019.

[30] C. Zhong, M. C. Gursoy, and S. Velipasalar, "Deep reinforcement learning-based edge caching in wireless networks," *IEEE Transactions on Cognitive Communications and Networking*, vol. 6, no. 1, pp. 48–61, 2020.

[31] B. Gahtan, R. Cohen, A. M. Bronstein, and G. Kedar, "Using deep reinforcement learning for mmwave real-time scheduling," in *2023 14th International Conference on Network of the Future (NoF)*, pp. 71–79, 2023.

[32] M. Abbasi, A. Shahraki, M. J. Piran, and A. Taherkordi, "Deep reinforcement learning for qos provisioning at the mac layer: A survey," *Engineering Applications of Artificial Intelligence*, vol. 102, p. 104234, 2021.

[33] Y. Bokobza, R. Dabora, and K. Cohen, "Deep reinforcement learning for simultaneous sensing and channel access in cognitive networks," *IEEE Transactions on Wireless Communications*, vol. 22, no. 7, pp. 4930–4946, 2023.

[34] R. Paul, K. Cohen, and G. Kedar, "Multi-flow transmission in wireless interference networks: A convergent graph learning approach," *IEEE Transactions on Wireless Communications*, vol. 23, no. 4, pp. 3691–3705, 2023.

[35] S. Bai, X. Wang, M. C. Gursoy, G. Jiang, and S. Xu, "Deep reinforcement learning for rechargeable uav-assisted data collection from dense mobile sensor nodes," *IEEE Access*, 2025.

[36] X. Li, J. Fang, W. Cheng, H. Duan, Z. Chen, and H. Li, "Intelligent power control for spectrum sharing in cognitive radios: A deep reinforcement learning approach," *IEEE Access*, vol. 6, pp. 25463–25473, 2018.

[37] Y. Cohen, T. Gafni, R. Greenberg, and K. Cohen, "Sinr-aware deep reinforcement learning for distributed dynamic channel allocation in cognitive interference networks," *IEEE Transactions on Wireless Communications*, vol. 24, no. 1, pp. 228–243, 2025.

[38] D. R. Yerramreddy, J. Marasani, S. V. G. Ponnuru, D. Min, *et al.*, "Harnessing deep reinforcement learning algorithms for image categorization: A multi algorithm approach," *Engineering Applications of Artificial Intelligence*, vol. 136, p. 108925, 2024.

[39] M. R. Dubayan, S. Ershadi-Nasab, M. Zomorodi, P. Plawiak, R. Tadeusiewicz, and M. B. Roui, "Automated classification of thyroid disease using deep learning with neuroevolution model training," *Engineering Applications of Artificial Intelligence*, vol. 146, p. 110209, 2025.

[40] A. Puzanov and K. Cohen, "Deep reinforcement one-shot learning for change point detection," in *2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 1047–1051, 2018.

[41] A. Puzanov, S. Zhang, and K. Cohen, "Deep reinforcement one-shot learning for artificially intelligent classification in expert aided systems," *Engineering Applications of Artificial Intelligence*, vol. 91, p. 103589, 2020.

[42] D. Kartik, E. Sabir, U. Mitra, and P. Natarajan, "Policy design for active sequential hypothesis testing using deep learning," in *2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 741–748, IEEE, 2018.

[43] C. Zhong, M. C. Gursoy, and S. Velipasalar, "Deep actor-critic reinforcement learning for anomaly detection," in *2019 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6, 2019.

[44] H. Szostak and K. Cohen, "Decentralized anomaly detection via deep multi-agent reinforcement learning," in *2022 58th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 1–4, 2022.

[45] G. Stamatelis and N. Kalouptsidis, "Deep reinforcement learning for active hypothesis testing with heterogeneous agents and cost constraints," *TechRxiv*, 2023.

[46] H. Szostak and K. Cohen, "Deep multi-agent reinforcement learning for decentralized active hypothesis testing," *IEEE Access*, 2024.

[47] N. Kalouptsidis and G. Stamatelis, "Neural predictor aided policy optimization for adversarial controlled sensing," *Signal Processing*, pp. 110–115, 2025.

[48] X. Liu, Z. Zhao, F. Yang, F. Liang, and L. Bo, "Environment adaptive deep reinforcement learning for intelligent fault diagnosis," *Engineering Applications of Artificial Intelligence*, vol. 151, p. 110783, 2025.

[49] M. Bogdanski, "1001 episodic semi gradient sarsa." https://marcinbogdanski.github.io/rl-sketchpad/RL_An_Introduction_2018/1001_Episodic_Semi_Gradient_Sarsa.html. Accessed: 2023-7-24.

[50] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT press, 2 ed., 2018.