drones

Article

Adaptive Nonlinear Model Predictive Horizon Using Deep Reinforcement Learning for Optimal Trajectory Planning

Younes Al Younes and Martin Barczyk *

Department of Mechanical Engineering, University of Alberta, Edmonton, AB T6G 1H9, Canada
* Correspondence: mbarczyk@ualberta.ca

Abstract: This paper presents an adaptive trajectory planning approach for nonlinear dynamical systems based on deep reinforcement learning (DRL). This methodology is applied to the authors' recently published optimization-based trajectory planning approach named nonlinear model predictive horizon (NMPH). The resulting design, which we call 'adaptive NMPH', generates optimal trajectories for an autonomous vehicle based on the system's states and its environment. This is done by tuning the NMPH's parameters online using two different actor-critic DRL-based algorithms, deep deterministic policy gradient (DDPG) and soft actor-critic (SAC). Both adaptive NMPH variants are trained and evaluated on an aerial drone inside a high-fidelity simulation environment. The results demonstrate the learning curves, sample complexity, and stability of the DRL-based adaptation scheme and show the superior performance of adaptive NMPH relative to our earlier designs.

Keywords: trajectory planning; nonlinear model predictive approach; adaptive design; deep reinforcement learning; deterministic policy gradient; soft actor-critic

Citation: Al Younes, Y.; Barczyk, M. Adaptive Nonlinear Model Predictive Horizon Using Deep Reinforcement Learning for Optimal Trajectory Planning. *Drones* **2022**, *6*, 323. https://doi.org/10.3390/drones6110323

Academic Editor: Andrey V. Savkin

Received: 3 October 2022
Accepted: 23 October 2022
Published: 27 October 2022

1. Introduction

Path planning and trajectory tracking control are compelling domains for researchers working with autonomous robotic systems. Some formulations require accurate system dynamics models to design the control and navigation algorithms [1]. However, obtaining accurate models is challenging in practice, especially if the system dynamics are vary by time or task. Changes in system dynamics require updating the system model and/or the associated control and navigation algorithms. For instance, adaptive control designs adjust the controller's parameters in response to changes in the system dynamics and the environment [2]. Adaptive control methods can be traced back to the 1950s and early 1960s [3]. Richard Bellman showed how dynamic programming is related to the different aspects of adaptation [4], and various adaptive flight control systems from this era are reported in [5]. One of the simplest instances of adaptive control is dynamically adjusting the gains of a PID control law; some techniques proposed by researchers for online PID tuning include [6–9].

The world is witnessing rapid progress in the use of artificial intelligence (AI) techniques for self-adaptive systems [10]. In particular, some AI-based techniques have generated great interest for adaptive control designs for mobile robots [11–13]. One of the most productive paradigms in AI is reinforcement learning (RL), which is a learning method for an agent interacting with its environment [14]. In the literature, RL has been used by researchers as an adaptive control strategy, for instance, a Q-learning-based cruise control method was developed by [15] to control a vehicle's speed on curved lanes. Q-learning [16] is an RL algorithm that learns the value of an action for a given state of the system. For online tuning purposes, [17] used the Q-learning method to auto-tune fuzzy PI and PD controllers for both single- and multi-input/output systems, while [18] used an actor-critic RL technique to tune the weights of an LQR controller to adjust to different payloads being carried by a robot arm manipulator.

Recent developments in RL have made it possible to use neural networks as approximators of the RL value and policy functions [14]. In general, RL methods that use neural networks in their structure are called deep reinforcement learning (DRL). One class of DRL methods that support continuous-time system models belongs to the actor-critic family [19], including the deep deterministic policy gradient (DDPG) [20], twin delayed deep deterministic (TD3) [21], soft actor-critic (SAC) [22], and asynchronous advantage actor–critic (A3C) [23]) algorithms. Actor-critic methods simultaneously learn policy and value functions that are maintained independently using separate memory structures [14]. The actor is a policy function that selects the best action for the current observations, and the critic is a value function that criticizes the actions made by the actor. The algorithms listed above have recently begun being used to implement adaptive control. For example, the DDPG algorithm was used by [1] for self-tuning gains of PID controllers onboard mobile robots, while [24] utilized the A3C algorithm to tune the gains of a PID controller used for position control of a two-phase hybrid stepping motor. DRL-based algorithms can also be used to autonomously tune the parameters of algorithms other than control, for instance path planning. This will be the focus of the present paper.

Recently, the authors introduced a path planning methodology called nonlinear model predictive horizon (NMPH) [25], which produces optimal, consistent, collision-free, and computationally efficient trajectories that respect the internal and external constraints of a mobile robot (in our case, an aerial drone). By design, the NMPH algorithm compensates for the system's nonlinearities to reduce or even remove the non-convexity of its underlying optimization problem. This is done by combining the nonlinear plant model with various nonlinear feedback control design methodologies, such as feedback linearization (FBL) [25] and backstepping control (BSC) [26]. The optimization problem embedded within NMPH contains various parameters that affect its cost function, as further explained in Section 2.1. In our previous works, these parameters were selected empirically; however, in the present paper, a new framework is proposed that dynamically adjusts these parameters to optimize the path planning performance in real time. Our approach uses DRL algorithms (DDPG or SAC) to automatically tune the NMPH parameters based on system states and observations from the environment. This framework is called 'adaptive NMPH'.

The research contributions of this paper are as follows:

- Introducing an adaptive NMPH framework that uses a DRL-based method to tune the parameters of the underlying optimization problem of generating the best possible reference trajectories for the vehicle.
- Designing the RL components (the agent, the environment, and the reward scheme) of the proposed system.
- Implementing two different actor-critic DRL algorithms—the deterministic DDPG approach and the probabilistic SAC algorithm—within the adaptive NMPH framework, comparing them in terms of learning speed and stability.
- Evaluating the performance of the overall system with each of the above DRL algorithms in a lifelike simulation environment.

The remainder of this paper is organized as follows: Section 2 describes the various methodologies used in this work. Section 3 presents the adaptive NMPH framework for trajectory planning. Section 4 evaluates the proposed designs in simulation, and Section 5 concludes the paper and proposes future work directions.

2. Methodologies

This section provides a background on the different methodologies used within the adaptive NMPH framework.

2.1. Nonlinear Model Predictive Horizon Based on Backstepping Control

Nonlinear model predictive horizon (NMPH) was originally proposed by the authors in [25]. NMPH is an optimization-based method used to generate reference trajectories for a closed-loop system. Within its optimization problem, NMPH uses a model of the nonlinear plant, a nonlinear control law (here, backstepping control), and a set of constraints representing input limits plus static and dynamic obstacles in the environment. Connecting the nonlinear plant with the control law aims to reduce the nonlinearity of the overall closed-loop system and consequently the non-convexity of the associated optimization problem. This greatly improves the efficiency of the optimization calculations, which enables real-time trajectory generation to run onboard the drone vehicle.

Consider a nonlinear system with state, input, and output vectors $x \in X \subseteq \mathbb{R}^{n_x}$, $u \in U \subseteq \mathbb{R}^{n_u}$, and $\xi \in \Xi \subseteq \mathbb{R}^{n_\xi}$, respectively. The output vector is assumed to be a subset of the system state, $\Xi \subseteq X$. In addition, let $f\big(x(n), u(n)\big) : X \times U \to X$ be the smooth map that represents the plant dynamics, and $g\big(x(n), \xi_{ref}(n)\big) : X \times \Xi \to U$ the smooth nonlinear control law map.

NMPH is designed to generate estimated reference trajectories $\hat{\xi}_{ref} \in \Xi$, which will be tracked by the closed-loop system consisting of the plant and control law. As shown in (1), a copy of these closed-loop dynamics is used by the NMPH optimization problem, where the variables used by NMPH are denoted by a $\tilde{}$ to visually differentiate them from the actual system variables. For instance, within (1), $\tilde{x}$ represents the *predicted system state trajectory*, and $\tilde{\xi}$ is the *predicted output trajectory*.

The online NMPH optimization problem to bring the system from a current state x to a terminal stabilization setpoint x_{ss} is shown in Equation (1) [27]. Let t_n, $n = 0, 1, 2, \cdots$ represent successive sampling times. At every sampling instant, the optimization treats the following problem for $\tilde{x}$ and $\hat{\xi}_{ref}$, running for as long as $\|x_{ss} - x(t_n)\| \geq \Delta$, where $\Delta \in \mathbb{R}^+$ is a user-specified tolerance:

$$\underset{\tilde{x}, \hat{\xi}_{ref}}{\operatorname{argmin}} \left(J\big(\tilde{x}, \hat{\xi}_{ref}\big) = E\big(\tilde{x}(t_n + T)\big) + \int_{t_n}^{t_n+T} L\big(\tilde{x}(\tau), \hat{\xi}_{ref}(\tau)\big)\, d\tau \right) \tag{1}$$

$$\begin{aligned}
\text{subject to}\quad & \tilde{x}(t_n) = x(t_n)\,, && \text{(1a)}\\
& \dot{\tilde{x}}(\tau) = f(\tilde{x}(\tau), \tilde{u}(\tau)), && \text{(1b)}\\
& \tilde{u}(\tau) = g\big(\tilde{x}(\tau), \hat{\xi}_{ref}(\tau)\big), && \text{(1c)}\\
& \tilde{x}(\tau) \in \mathcal{X},\ \ \tilde{u}(\tau) \in \mathcal{U},\ \ \tilde{\xi}(\tau), \hat{\xi}_{ref}(\tau) \in \mathcal{Z}, && \text{(1d)}\\
& \mathcal{O}_i(\tilde{x}) \leq 0\,, \quad i = 1,\, 2,\, \ldots,\, p, && \text{(1e)}\\
\text{for}\quad & \tau \in [t_n, t_n + T].
\end{aligned}$$

where $\mathcal{X} \subseteq X$, $\mathcal{U} \subseteq U$, and $\mathcal{Z} \subseteq X$ are the constraint sets for the state, input, and output trajectories, respectively, and each $\mathcal{O}_i(\tilde{x}) \leq 0$ in (1e) is an inequality constraint corresponding to a detected static or dynamic obstacle within the environment [27]. The stage cost L and terminal cost E functions in (1) are assigned as follows:

$$L\big(\tilde{x}(\tau), \hat{\xi}_{ref}(\tau)\big) = \|\tilde{x}(\tau) - x_{ss}\|^2_{W_x} + \|\tilde{\xi}(\tau) - \hat{\xi}_{ref}(\tau)\|^2_{W_\xi} \tag{2a}$$

$$E\big(\tilde{x}(t_n + T)\big) = \|\tilde{x}(t_n + T) - x_{ss}\|^2_{W_T} \tag{2b}$$

where the errors in (2a) and (2b) are weighted by matrices $W_x \in \mathbb{R}^{n_x \times n_x}$, $W_\xi \in \mathbb{R}^{n_\xi \times n_\xi}$, and $W_T \in \mathbb{R}^{n_x \times n_x}$, which in this work will be adaptively tuned using DRL algorithms.

The optimization problem in (1) begins with measuring the current state of the physical system $x(t_n)$ at time t_n. The cost function $J\big(\tilde{x}, \hat{\xi}_{ref}\big)$ is then minimized over the prediction horizon $[t_n, t_n + T]$ subject to constrains (1b), (1c), and (1e) to provide a prediction of the values of $\tilde{x}$ and $\hat{\xi}_{ref}$. Finally, either the estimated reference trajectory $\hat{\xi}_{ref}$ or the predicted output trajectory $\tilde{\xi}$ (as both converge to each other) is input into the closed-loop system for tracking. This process is repeated in real time at a user-specified rate until the plant reaches the desired terminal setpoint. Details about the NMPH approach can be found in our recent works [25,26].

In this work, the nonlinear backstepping control law is used within the NMPH optimization problem as a constraint in (1c). The detailed development and implementation of the BSC technique within NMPH, as well as its advantages over the earlier FBL-based design [25], are described in our recent work [26].

The NMPH trajectory planning algorithm receives terminal points from a modular global motion planner [27]. The global motion planner generates terminal points within unexplored areas of an incrementally built-up volumetric map of the environment [28,29]. These terminal points, along with the current pose of the vehicle, the constraints representing the closed-loop system dynamics and environmental obstacles (which are extracted from the volumetric map), and the entries of the weighting matrices (which in the present design are adjusted online by a DRL algorithm) are sent to the NMPH optimization problem in order to calculate optimal trajectories between the vehicle's current pose and the next terminal point. The results are then used as reference trajectories by the vehicle's low-level flight controller.

2.2. Deep Reinforcement Learning Overview

This section covers the preliminaries of reinforcement learning, then describes the DDPG and SAC algorithms used within the adaptive NMPH frameworks.

2.2.1. Reinforcement Learning Preliminaries

A reinforcement learning (RL) system is composed of an agent that interacts with an environment in a sampling-based manner. Assuming the environment is fully observed, at each time sample the agent observes the environment state $s \in \mathcal{S}$, applies the action $a \in \mathcal{A}$ decided by a policy, and receives a scalar reward $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, where $\mathcal{S}$ and $\mathcal{A}$ are the environment state space and the action space, respectively. In our work, we consider continuous action spaces with a real-valued vector $a \in \mathbb{R}^n$. The main components of an RL framework are depicted in Figure 1.

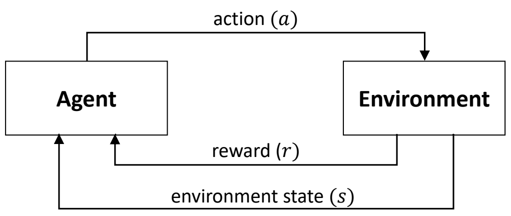


Figure 1. Block diagram of an RL framework.

The agent's policy can be deterministic (denoted by $\mu(s)$), or stochastic (denoted by $\pi(\cdot|s)$). In deep RL, we parameterize the policy and represent it using a universal function approximator realized by a neural network. The parameters (representing the weights and biases of the policy's neural network) are denoted by θ, and the corresponding policies for the deterministic and stochastic cases are denoted by $\mu_\theta(s)$ and $\pi_\theta(\cdot|s)$, respectively.

We consider a stochastic environment with transition probability function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$, where $p(s', r|s, a)$ is the probability of transition from the current state s and action a to the next state s' with reward $r \in \mathcal{R}$. In addition, we define the 'return' as the expected weighted sum of future rewards $R = \sum_{t=0}^{\infty} \gamma^t r(s, a)$, where $r(s, a)$ is the reward function and $0 \leq \gamma \leq 1$ is the discounting factor. The main objective in RL is to find a policy that maximizes the expected sum of rewards $J = \underset{\tau \sim \pi}{\mathbb{E}}[R]$, where $\tau = (s_0, a_0, s_1, a_1, \ldots)$ is the trajectory sequence of states and actions in the RL system.

The state-action value function (also known as the Q-function) specifies the expected return of an agent after performing an action a at a state s by following a policy π or μ. The Q-function can be described by a Bellman equation [14].

Many recent advances in deep reinforcement learning consider a replay buffer (also known as an experience buffer or experience replay) during the learning process. The replay buffer is a memory that collects the previous experience tuples $(s, a, r, s') \in \mathcal{B}$, in which the agent uses them to increase the computational efficiency and speed up learning [30].

We will now review the DDPG and SAC deep reinforcement learning algorithms used within our proposed adaptive NMPH frameworks.

2.2.2. Deep Deterministic Policy Gradient

Deep deterministic policy gradient (DDPG) [20] is a model-free deep reinforcement learning technique that is designed for applications with deterministic action spaces. It uses stored experiences in a replay buffer to concurrently learn a Q-function and a policy. DDPG is classified as an actor-critic technique, where the actor is a policy network that receives the state of the environment and provides continuous action to the system, while the critic is a Q-function network that inputs a state and action pair and outputs a Q-value.

DDPG seeks to find the optimal action-value function $Q^*(s,a)$ followed by the optimal action $a^*(s)$, where $a^*(s) = \arg\max_a Q^*(s,a)$. As a deep reinforcement learning approach, DDPG uses universal function approximators represented by neural networks to learn $Q^*(s,a)$ and $a^*(s)$. Consider a neural network approximator $Q_\phi(s,a)$ (also known as a Q-network) with parameters ϕ, where the objective is to make the approximator as close as possible to the optimal action-value function written in the form of a Bellman equation. The associated mean square Bellman error (MSBE [31]) function is defined as follows:

$$J_Q(\phi,\mathcal{B}) = \mathop{\mathbb{E}}_{(s,a,r,s')\sim\mathcal{B}}\left[\Big(Q_\phi(s,a) - \big(r + \gamma \max_{a'} Q_\phi(s',a')\big)\Big)^2\right] \tag{3}$$

where a random batch of data (s,a,r,s') from the replay buffer $\mathcal{B}$ is used for each update. The goal is to minimize the loss in (3) by performing a gradient descent of the MSBE $J_Q(\phi,\mathcal{B})$.

As shown in (3), the neural network parameters represented by ϕ are used for both the action-value function approximator $Q_\phi(s,a)$ *and* the network that estimates $Q_\phi(s',a')$, which uses the *next* states and actions. Unfortunately, this makes it impossible for the gradient descent to converge. To tackle this issue, a time delay is added to the network parameters ϕ for $Q_\phi(s',a')$. The adjusted network is called the target Q-network $Q_{\phi_{\text{targ}}}(s',a')$ with parameters ϕ_{targ}. A copy of the Q-network $Q_\phi(s',a')$ is used for the target Q-network $Q_{\phi_{\text{targ}}}(s',a')$, where the latter uses the weighted average of the model parameters $\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$ to stabilize Q-function learning [32]. It should be noted that the parameters of the target Q-network are not trained. However, they are periodically synchronized with the original Q-network's parameters.

The MSBE function given in (3) contains a maximization term for the Q-value. One way to perform this maximization is to apply the optimal action $a^*(s)$. This can be achieved by creating another approximator for the policy $\mu_\theta(s)$ with parameters θ and maximizing the associated Q-function with regard to the replay buffer $\mathcal{B}$. This new policy also requires a time delay to stabilize its learning. Therefore, a target policy $\mu_{\theta_{\text{targ}}}(s)$ is introduced to maximize $Q_{\phi_{\text{targ}}}$. The Bellman equation, MSBE, and policy learning function are respectively given by

$$y(r,s') = r + \gamma \overbrace{Q_{\phi_{\text{targ}}}(s', \underbrace{\mu_{\theta_{\text{targ}}}(s')}_{\text{target policy network}})}^{\text{target Q-network}} \tag{4}$$

$$J_Q(\phi,\mathcal{B}) = \mathop{\mathbb{E}}_{(s,a,r,s')\sim\mathcal{B}}\left[\big(\underbrace{Q_\phi(s,a)}_{\text{Q-network}} - y(r,s')\big)^2\right] \tag{5}$$

$$J_\mu(\theta,\mathcal{B}) = \mathop{\mathbb{E}}_{s\sim\mathcal{B}}\left[Q_\phi(s,\mu_\theta(s))\right] \tag{6}$$

Practically, for a random sample $B = \{(s, a, r, s')\}$ from the replay buffer $\mathcal{B}$ with cardinality $|B|$, Equations (5) and (6) can be expressed as

$$J_Q(\phi, B) = \frac{1}{|B|} \sum_{(s,a,r,s') \in B} \left(Q_\phi(s,a) - y(r,s')\right)^2 \tag{7}$$

$$J_\mu(\theta, B) = \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s)) \tag{8}$$

During training, Ornstein–Uhlenbeck noise is added to the action vector to enhance the exploration of the DDPG policy [31]. The pseudo-code summarizing the DDPG process is given in Algorithm 1.

Algorithm 1 Deep Deterministic Policy Gradient.

```
1:  Initialize: $\theta$, $\phi$, $\mathcal{B} \leftarrow \emptyset$
2:  Set $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3:  repeat
4:      Observe the state $s$
5:      Find and apply noise to the action $a = \mu_\theta(s) + \eta_{\text{OU-noise}}$
6:      Apply $a$ by the agent
7:      Observe the next state $s'$ and calculate the reward $r$
8:      Store $(s, a, r, s')$ in the replay buffer $\mathcal{B}$
9:      for a given number of episodes do
10:         Obtain a random sample $B = \{(s, a, r, s')\}$ from $\mathcal{B}$
11:         Compute Bellman function $y(r, s')$
12:         Update the Q-function by applying gradient descent to MSBE: $\nabla_\phi J_Q(\phi, B)$
13:         Update the policy by applying gradient ascent to (8): $\nabla_\theta J_\mu(\theta, B)$
14:         Update the parameters of the target networks: $\begin{cases} \phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi \\ \theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta \end{cases}$
15: until convergence
```

The hyperparameters used for the DDPG algorithm are the number of training episodes, target update factor (ρ), actor and critic network learning rates, replay buffer size, random batch size, and discount factor value. The sensitivity to the hyperparameter values and the interaction between the Q-value and policy approximator $\mu_\theta(s)$ make analyzing the stability and convergence of DDPG difficult tasks [33], especially when using high-dimensional nonlinear universal function approximators [34]. Moreover, DDPG is expensive in terms of its sample complexity, which is measured by the number of training samples needed to complete the learning process.

An alternative approach that overcomes the issues of the DDPG algorithm is soft actor-critic (SAC) [22,34], a probabilistic DRL algorithm, which is considered next.

2.2.3. Soft Actor-Critic

Soft actor-critic (SAC) is a model-free deep reinforcement learning technique that obtains a stochastic policy by maximizing its expected return and entropy [22]. Maximizing the expected entropy in the policy leads to broader exploration in complicated domains, which enhances the sampling efficiency, increases robustness, and guards against convergence to a local maximum [31]. SAC is a probabilistic framework that builds on Soft Q-learning within an actor-critic formulation.

SAC involves simultaneously learning two Q-functions Q_{ϕ_1}, Q_{ϕ_2} using two different Q-networks, as well as a stochastic policy π_θ using a policy network. Both Q-functions use a modified MSBE (known as soft-MSBE), to be presented in (10), where the minimum Q-value of both functions is used to update the policy [21]. SAC employs a 'target network' associated with each Q-network to enhance the stability of the learning process, where both target Q-networks are copies of the corresponding Q-network, but employ weighted averaging on the network parameters during training. Because of the policy's stochastic nature, SAC uses the current policy to obtain the next state-action values without needing to have a target policy [31]. In addition, the stochastic nature of the exploration process means it's not necessary to artificially introduce noise, as was done in the deterministic DDPG.

The objective of SAC is to maximize the sum of the expected return and entropy. The Bellman equation within its Q-value function thus includes the expected entropy of the policy as follows:

$$Q_\pi(s,a) \approx r + \gamma\big(Q_\pi(s'_\mathcal{B}, a'_\pi) - \alpha \log \pi(a'_\pi|s'_\mathcal{B})\big) \tag{9}$$

where α is the coefficient that regulates the trade-off between the expected entropy and return, $s'_\mathcal{B}$ indicates that the replay buffer is used to obtain the expectation of the future states, and $a'_\pi \sim \pi(\cdot|s')$ indicates that the current policy is used to obtain future actions. For simplicity of notation, we will denote $s'_\mathcal{B}$ by s' and a'_π by a' in the sequel.

Two Bellman residuals are used within SAC [22], referred to as soft-MSBEs. In addition to the policy network π_θ, each soft-MSBE includes a Q-network and two target Q-networks in its calculation as follows:

$$J_Q(\phi_i, \mathcal{B}) = \mathop{\mathbb{E}}_{(s,a,r,s',a')\sim\mathcal{B}}\left[\Big(Q_{\phi_i}(s,a) - y(r,s',a')\Big)^2\right], \qquad i = 1,2 \tag{10}$$

and their Bellman equation forms are

$$y(r,s',a') = r + \gamma\left(\min_{j=1,2} Q_{\phi_{\text{targ},j}}(s',a') - \alpha \log \pi_\theta(a'|s')\right), \quad a' \sim \pi_\theta(\cdot|s') \tag{11}$$

Similar to DDPG, the Q-functions are updated using gradient descent, while gradient ascent is utilized to update the policy network.

The policy should maximize the state-value function $V_\pi(s)$, defined as follows:

$$V_\pi(s) = \mathop{\mathbb{E}}_{a\sim\pi}\Big[Q_\pi(s,a) - \alpha \log \pi(a|s)\Big] \tag{12}$$

which represents the expected return when starting from a state s and following a policy π.

For the optimal value of the action, we can employ reparameterization [22,31] to obtain a continuous action from a deterministic function that represents the policy. The function is expressed by the state and additive Gaussian noise as follows:

$$a_\theta(s,\xi) = \tanh(\mu_\theta(s) + \sigma_\theta(s)\,\xi), \qquad \xi \sim \mathcal{N}\big(0, \text{diag}(1,\ldots,1)\big). \tag{13}$$

The policy optimization can be performed by maximizing the Q-function, which implicitly maximizes the entropy of the trajectory. Using the computed value of the action from (13), the function to be maximized is

$$J_\pi(\theta, \mathcal{B}) = \mathop{\mathbb{E}}_{s\sim\mathcal{B},\, \xi\sim\mathcal{N}}\left[\min_{j=1,2} Q_{\phi_j}(s, a_\theta(s,\xi)) - \alpha \log \pi_\theta(a_\theta(s,\xi)|s)\right] \tag{14}$$

and the optimum policy can be obtained by finding $\arg\max_\theta\ J_\pi(\theta,\mathcal{B})$ using gradient ascent. For a random sample $B = \{(s,a,r,s',a')\}$ from the buffer $\mathcal{B}$, Equations (10) and (14) can be expressed as follows:

$$J_Q(\phi_i, B) = \frac{1}{|B|} \sum_{(s,a,r,s')\in B} \big(Q_{\phi_i}(s,a) - y(r,s',a')\big)^2, \qquad i = 1,2 \tag{15}$$

$$J_\mu(\theta, B) = \frac{1}{|B|} \sum_{s\in B} \left(\min_{j=1,2} Q_{\phi_j}\big(s, a_\theta(s,\xi)\big) - \alpha \log \pi_\theta\big(a_\theta(s,\xi)|s\big)\right) \tag{16}$$

The pseudo-code for the SAC algorithm is provided in Algorithm 2.

Algorithm 2 Soft Actor-Critic.

```
1: Initialize: $\theta$, $\phi_i$, $\alpha$, $\mathcal{B} \leftarrow \emptyset$,     $i = 1,2$
2: Set $\phi_{\text{targ},i} \leftarrow \phi_i$
3: **repeat**
4:     Observe the state $s$
5:     Find the action $a \sim \pi_\theta(\cdot|s)$, and apply it through the agent
6:     Observe the next state $s'$ and the reward $r$
7:     Find the next action $a' \sim \pi_\theta(\cdot|s')$
8:     Store $(s, a, r, s', a')$ in the replay buffer $\mathcal{B}$
9:     **for** a given number of episodes **do**
10:        Obtain a random sample $B = \{(s, a, r, s', a')\}$ from $\mathcal{B}$
11:        Compute Bellman functions $y(r, s', a')$ in (11) and find the soft-MSBEs (10)
12:        Apply gradient descent on the soft-MSBEs: $\nabla_{\phi_i} J_Q(\phi_i, B)$
13:        Reparametrize the action: $a_\theta(s, \xi) = \tanh(\mu_\theta(s) + \sigma_\theta(s)\,\xi)$
14:        Apply gradient ascent on the policy: $\nabla_\theta J_\mu(\theta, B)$
15:        Apply gradient descent to tune $\alpha$: $\nabla_\alpha J(\alpha)$
16:        Update target networks: $\phi_{\text{targ},i} \leftarrow \rho\phi_{\text{targ},i} + (1-\rho)\phi_i$
17: **until** convergence
```

3. Adaptive Trajectory Planning Framework

In this section, we present the DRL-based adaptive framework used to adjust the gains of the NMPH trajectory planning algorithm. First, we will describe the agent and environment involved in the DRL problem; then, we will present two adaptive NMPH architectures based on the DDPG and the SAC algorithm, respectively.

3.1. Agent and Environment Representations

Figure 2 shows the main components of the adaptive NMPH system. The environment is an autonomous drone that flies within an incrementally built-up 3D volumetric map of the surroundings. The drone uses the NMPH algorithm for planning local trajectories between the current pose and a terminal setpoint provided by the exploration algorithm presented in [27]. As covered in Section 2.1, the NMPH optimization process (blue box in Figure 2) contains models of the nonlinear system dynamics and nonlinear control law, as well as constraints representing actuation limits and environmental obstacles. The onboard flight control system tracks the optimum reference trajectories generated by the NMPH.

From an RL perspective, at each episode the drone is commanded to fly through k terminal setpoints. Hence, each episode consists of k iterations. Following each iteration, three observations are sent to the agent: initial velocity v_o, angle φ between the initial velocity vector v_o and the vector $\vec{r} = p_{ss} - p_o$ running from the initial point p_o to the terminal point p_{ss}, and the distance $|\vec{r}|$.

A sketch of the observations $\{v_o,\ \varphi,\ |\vec{r}|\}$ for one iteration is given in Figure 3.

Our objective is to tune the NMPH parameters online by using reinforcement learning to maximize the total reward. This reward is a function of the tracking performance by the drone of the reference path generated by the NMPH algorithm, which consists of three indicators:

- Trajectory tracking reward, which reflects how well the flight trajectory matches the generated reference. The trajectory tracking reward is calculated as follows:

$$r_{traj} = \begin{cases} -\frac{r_{t,max}}{r_{t,th}} e_{t,\text{RMS}} + r_{t,max}, & \text{for } e_{t,\text{RMS}} \le r_{t,th} \\ 0, & \text{otherwise} \end{cases} \quad (17)$$

where $e_{t,\text{RMS}}$ is root-mean-square (RMS) error between the generated and flight trajectories, and $r_{t,max}$ and $r_{t,th}$ are the maximum and threshold values of the trajectory tracking reward, respectively.

- *Terminal setpoint reward*, which reflects how close the ending point of the flight trajectory is to the terminal setpoint of the reference trajectory. The terminal setpoint reward is calculated as follows:

$$r_{ss} = \begin{cases} -\frac{r_{s,max}}{r_{s,th}} e_{ss} + r_{s,max}, & \text{for } e_{ss} \leq r_{s,th} \\ 0, & \text{otherwise} \end{cases} \tag{18}$$

where $e_{ss} = \|p_{ss} - \hat{\xi}^{pos}_{ref}(t_n + T)\|$ is the error between the terminal point and the final point of the reference trajectory generated by the NMPH, and $r_{s,max}$, $r_{s,th}$ are the maximum and threshold values of this reward, respectively.

- *Completion reward*, which reflects how far the drone travels along its prescribed flight trajectory in the associated time interval. This is given by the following:

$$r_c = \begin{cases} -\frac{r_{c,max}}{r_{c,th}} e_c + r_{c,max}, & \text{for } e_c \leq r_{c,th} \\ -5, & \text{otherwise} \end{cases} \tag{19}$$

where $e_c = \|p_{ss} - p|_{t_n+T}\|$ is the error between the drone's position at $t_n + T$ and the flight trajectory's endpoint, while $r_{c,max}$, $r_{c,th}$ are the maximum and threshold values of the completion reward, respectively. We place more importance on this factor by reducing the total reward ($r_c < 0$) whenever the error e_c exceeds the assigned threshold value $r_{c,th}$. Consequently, the overall algorithm will give priority to ensuring the drone reaches the desired setpoint in the allotted timeframe.

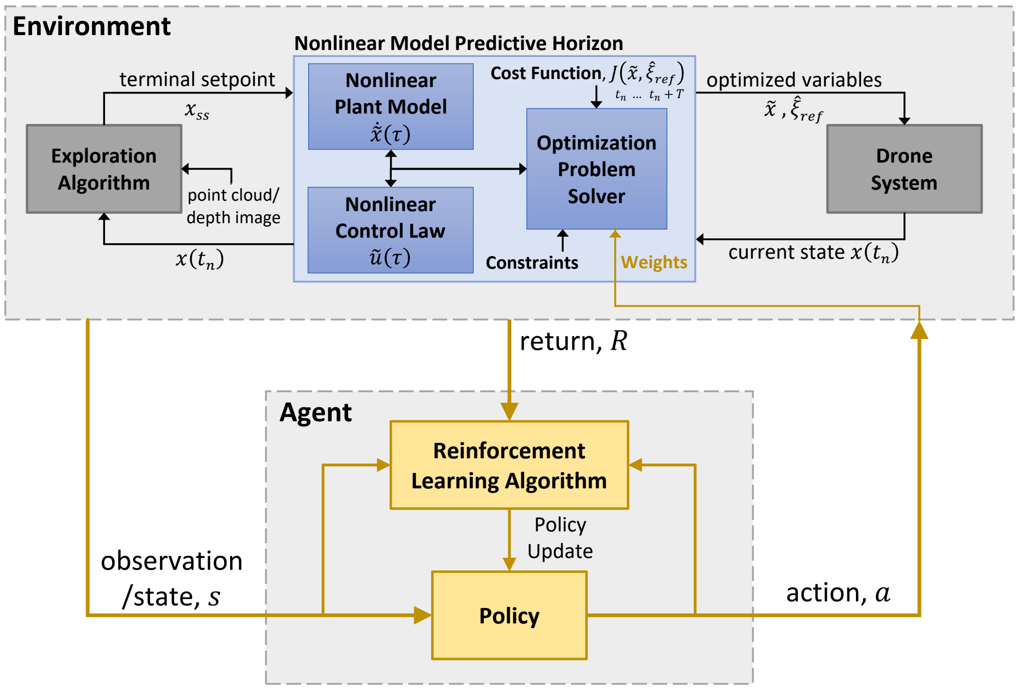


Figure 2. Adaptive NMPH architecture.

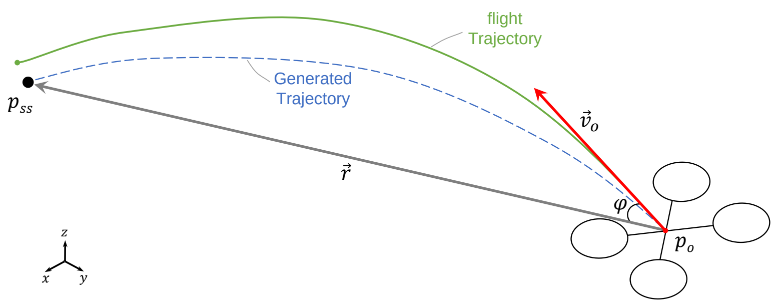


Figure 3. Observations from the environment for one Iteration.

3.2. DRL-Based Adaptive NMPH Architecture

The objective of adaptive NMPH is to integrate deep learning—in this case, an actor-critic method (DDPG or SAC)—within the NMPH optimization problem to adaptively tune the NMPH parameters and thus provide the best possible reference flight trajectories for the drone.

The structures of the NMPH-DDPG and NMPH-SAC algorithms are illustrated in Figures 4 and 5, respectively. Both DRL structures contain two parts, the actor and the critic. The actor contains the policy network, which selects the action that maximizes the total reward (a function of the state of the vehicle and environment) and subsequently improves the policy based on feedback from the critic. A target policy network is used in DDPG to obtain a stable learning process, while SAC does not need a target network because of its probabilistic nature. The critic is responsible for policy evaluation; within DDPG, it consists of a Q-network and a target Q-network, while in SAC, it is composed of two Q-networks, two target Q-networks, and an optimization problem for α tuning. Both DDPG and SAC employ a replay buffer to store previous experiences, which are used to refine the actor and critic networks. The policy evaluation and improvement processes within DDPG and SAC are explained in Sections 2.2.1 and 2.2.2 and depicted in Figures 4 and 5, respectively.

The action produced by the actor is a vector of positive values representing the entries of the weighting matrices used in the NMPH optimization problem. Using these, NMPH calculates its stage and terminal cost functions used to perform its optimization and generates the estimated reference trajectory $\hat{\xi}_{ref}$. This result is used by the drone's flight control system, and the vehicle's resulting trajectory is used to calculate the observations $\{v_o,\ \varphi,\ |\vec{r}|\}$ and the total reward $r_t = r_{traj} + r_{ss} + r_c$ sent to the replay buffer to be used in the learning process.

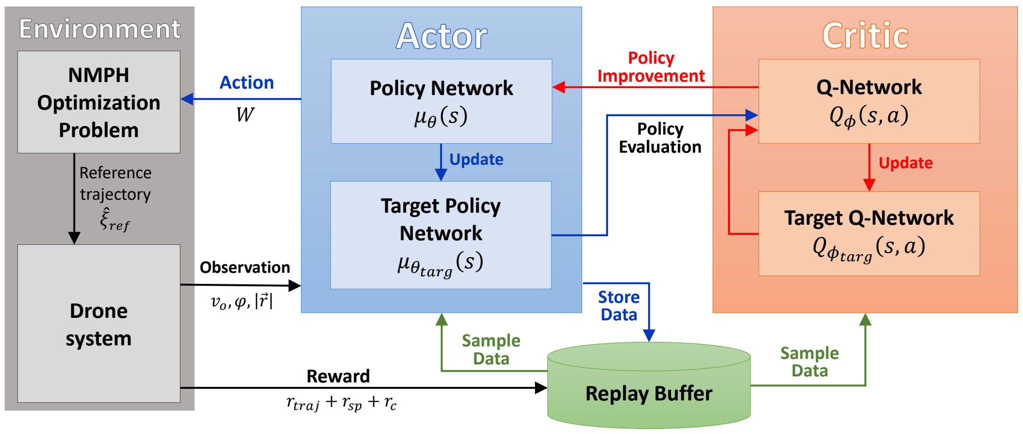

Environment
NMPH Optimization Problem
Reference trajectory $\hat{\xi}_{ref}$
Drone system
Action
W
Observation
$v_o, \varphi, |\vec{r}|$
Reward
$r_{traj} + r_{sp} + r_c$
Actor
Policy Network $\mu_\theta(s)$
Update
Target Policy Network $\mu_{\theta_{targ}}(s)$
Policy Improvement
Policy Evaluation
Critic
Q-Network $Q_\phi(s,a)$
Update
Target Q-Network $Q_{\phi_{targ}}(s,a)$
Store Data
Sample Data
Sample Data
Replay Buffer

Figure 4. Adaptive NMPH-DDPG structure.

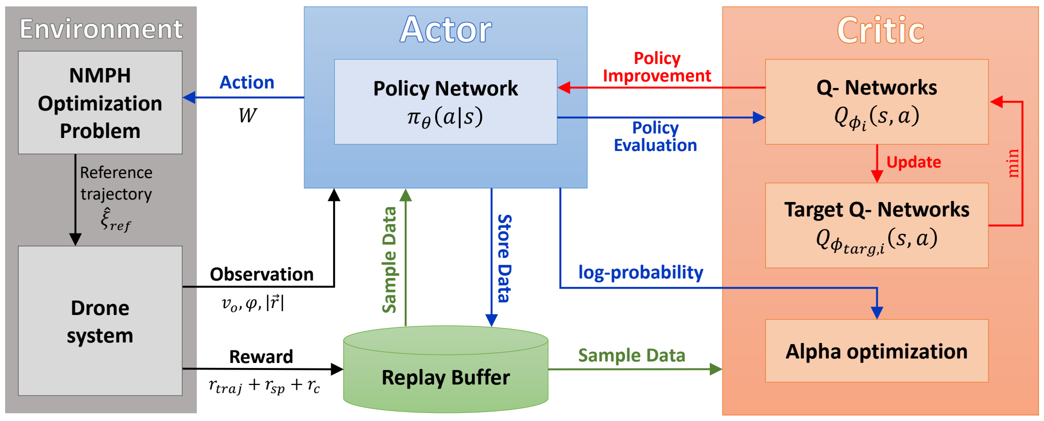


Figure 5. Adaptive NMPH-SAC structure.

4. Implementation and Evaluation

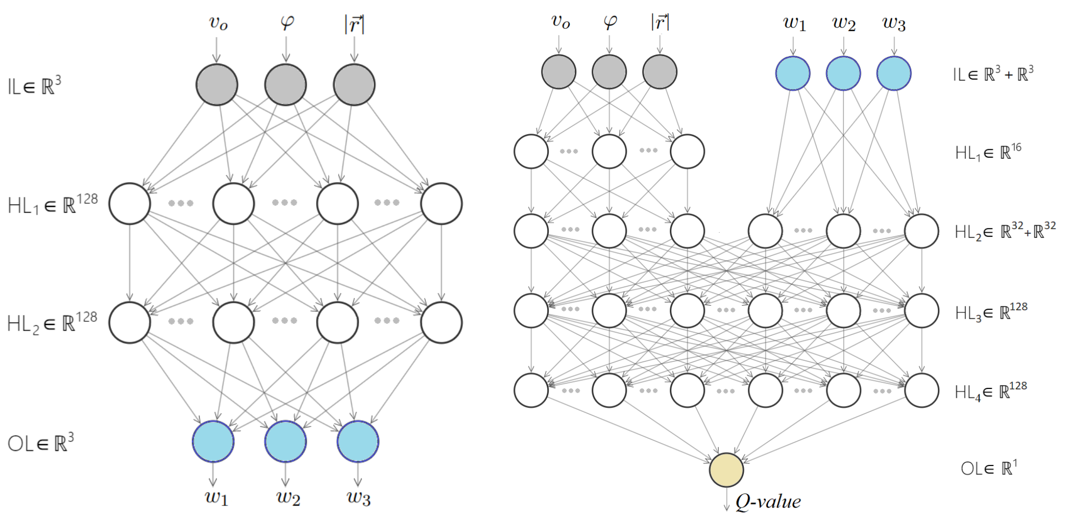
This section evaluates the effectiveness of tuning the NMPH parameters in real time via two DRL algorithms (DDPG and SAC). It also assesses the sample complexity and stability of both methods.

The overall architecture is implemented within the robot operating system (ROS) [35], which handles the interactions between the various subsystems, including physics simulation, optimization calculations, and DRL algorithm. The AirSim open-source simulator [36] is used to simulate the physics of the drone and provides photo-realistic environment data. For optimization, the ACADO Toolkit [37] is used to solve the NMPH's optimization problem in real time. The TensorFlow [38] and Keras [39] libraries are used to train the deep neural networks within the DDPG and SAC algorithms. In addition, the TensorLayer library [40] was used to tailor the SAC algorithm to our application. TensorLayer is a TensorFlow-based package that offers various RL and DRL modules for learning system implementations.

As stated in Section 3.2, three observations of the system are fed back to the individual neural networks: v_o, φ, and $|\vec{r}|$. DDPG is very sensitive to hyperparameters when the action space has a high dimension, in which case achieving stable learning becomes challenging. Therefore, we employ only three actions corresponding to the weights of the NMPH optimization dealing with position states. The learning process for the three weight factors $\{w_1 = w_x, w_2 = w_y, w_3 = w_z\}$ is performed using DDPG and SAC in parallel for comparison purposes.

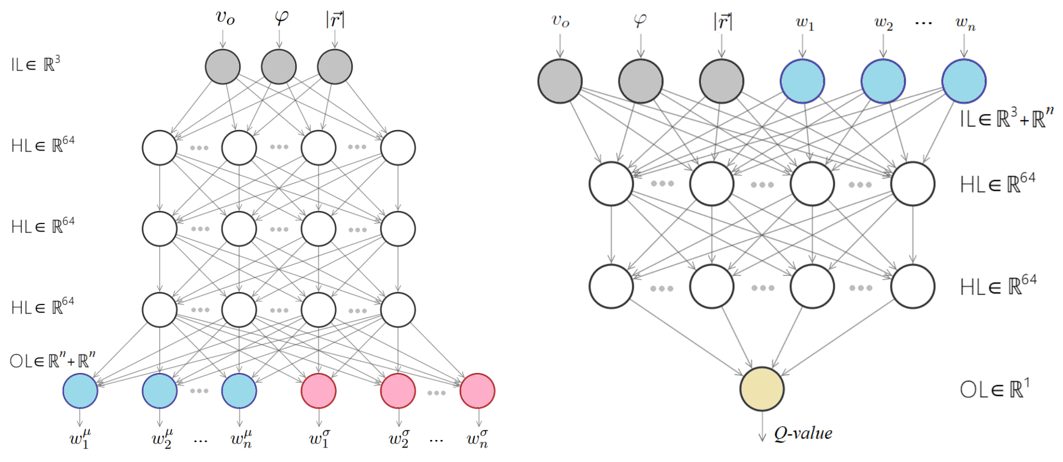
Each episode is composed of a sequence of iterations, where each iteration represents a trajectory between two endpoints (terminal points). At the start of each iteration, the velocity vector of the drone v_o, the angle φ between the velocity and endpoint-to-endpoint vectors, and the distance $|\vec{r}|$ between endpoints are calculated, followed by the errors $\{e_{t,\text{RMS}}, e_{ss}, e_c\}$ and the total reward at the end of the iteration. All this data is stored in the replay buffer. In order to cover a wider portion of the state and action spaces of the system, the initial velocity is randomly selected at the beginning of each episode.

The structures of the actor-critic DRL (policy and Q-networks) for DDPG and SAC algorithms are presented in Figures 6 and 7, respectively. Each network is composed of an input layer, multiple hidden layers, and an output layer. Figures 6 and 7 depict our neural network designs in terms of the layer structure of each network and the number of nodes in each layer. The policy networks in the actor are responsible for generating actions that maximize the total reward based on observations of the environment, while the Q-networks in the critic compute a Q-value that is used for policy improvement. For DDPG, four networks are used: a policy network, a Q-network (depicted in Figure 6), a target policy network, and a target Q-network. The target networks are replicas of the policy and Q-networks with a delay added to their parameters. Meanwhile, SAC consists of five networks: a policy network, two Q-networks, and two target Q-networks. The SAC's policy and Q-network structures are shown in Figure 7.


v_o φ $|\vec{r}|$
$\mathrm{IL} \in \mathbb{R}^3$
$\mathrm{HL}_1 \in \mathbb{R}^{128}$
$\mathrm{HL}_2 \in \mathbb{R}^{128}$
$\mathrm{OL} \in \mathbb{R}^3$
w_1 w_2 w_3
v_o φ $|\vec{r}|$ w_1 w_2 w_3
$\mathrm{IL} \in \mathbb{R}^3 + \mathbb{R}^3$
$\mathrm{HL}_1 \in \mathbb{R}^{16}$
$\mathrm{HL}_2 \in \mathbb{R}^{32} + \mathbb{R}^{32}$
$\mathrm{HL}_3 \in \mathbb{R}^{128}$
$\mathrm{HL}_4 \in \mathbb{R}^{128}$
$\mathrm{OL} \in \mathbb{R}^1$
Q-value

(**a**) Actor network (policy network) (**b**) Critic network (Q-network)

Figure 6. Neural networks used by DDPG. IL: input layer; HL: hidden layer; OL: output layer.


v_o φ $|\vec{r}|$
$\mathrm{IL} \in \mathbb{R}^3$
$\mathrm{HL} \in \mathbb{R}^{64}$
$\mathrm{HL} \in \mathbb{R}^{64}$
$\mathrm{HL} \in \mathbb{R}^{64}$
$\mathrm{OL} \in \mathbb{R}^n + \mathbb{R}^n$
w_1^μ w_2^μ … w_n^μ w_1^σ w_2^σ … w_n^σ
v_o φ $|\vec{r}|$ w_1 w_2 … w_n
$\mathrm{IL} \in \mathbb{R}^3 + \mathbb{R}^n$
$\mathrm{HL} \in \mathbb{R}^{64}$
$\mathrm{HL} \in \mathbb{R}^{64}$
$\mathrm{OL} \in \mathbb{R}^1$
Q-value

(**a**) Actor network (policy network) (**b**) Critic network (Q-network)

Figure 7. Neural networks used by SAC. IL: input layer; HL: hidden layer, OL: output layer.

Figure 8 shows the average episodic reward during the training processes of the DDPG and SAC architectures. In this comparison, each framework is learning to optimize the values of only three actions, which represent the entries of the weight matrix corresponding to the position states within the NMPH optimization problem.

To enhance DDPG performance in terms of sample complexity and its sensitivity to hyperparameters, we propose and apply a 'pre-exploration' technique, which traverses the RL problem spaces before the training process is started. Pre-exploration is performed by applying a set of predefined actions and considering a random system state for each action. The collected experiences of the pre-exploration process are then stored in the replay buffer, which is used during the training process. It was found that using this technique helps DDPG to improve convergence and stability over the case without pre-exploration, as can be seen from Figure 8. Conversely, a number of episodes must be spent for pre-exploration, which delays the learning process in the real-time adaptation. Note that the results shown in Figure 8 also show that SAC generally outperforms DDPG (either with or without pre-exploration) in terms of learning speed. In addition, during the training process, SAC showed noticeably better learning stability relative to DDPG with regard to the process of selecting the hyperparameter values for each algorithm.

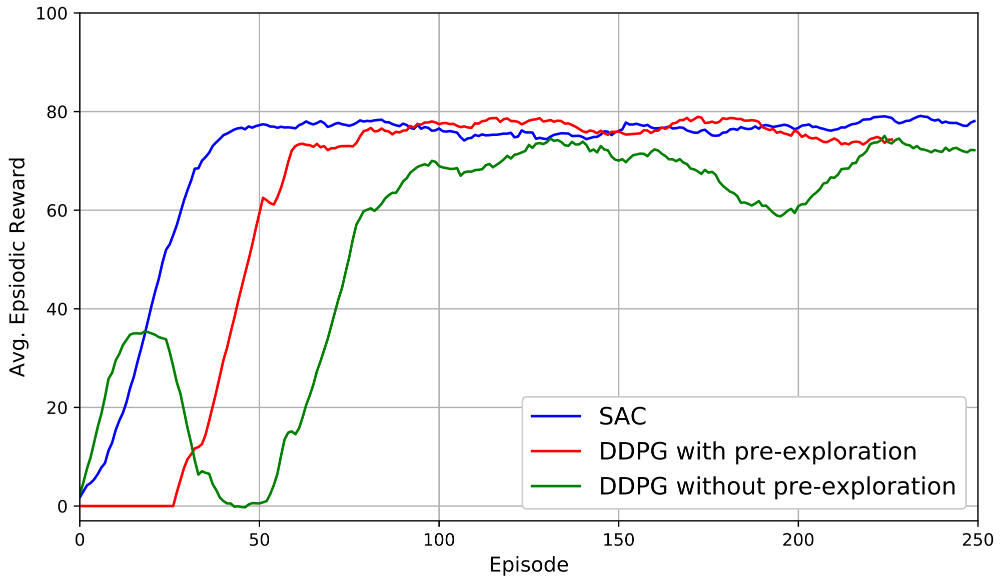


Figure 8. Training curves of SAC, DDPG with pre-exploration, and DDPG without pre-exploration for adaptively tuning three NMPH parameters.

To test the performance of the SAC approach in a higher-dimensional setting, the number of actions was increased to 12 to estimate the weight matrix entries corresponding to the position, velocity, and acceleration states $\{w_x, w_y, w_z, w_\psi, w_{\dot{x}}, w_{\dot{y}}, w_{\dot{x}}, w_{\dot{\psi}}, w_{\ddot{x}}, w_{\ddot{y}}, w_{\ddot{x}}, w_{\ddot{\psi}}\}$ within the NMPH optimization problem. Figure 9 shows the resulting training curve of SAC; DDPG failed to complete the learning process in this case. The effect of increasing the number of NMPH parameters being tuned can be seen by comparing the SAC training curves in Figures 8 and 9 in terms of the average episodic reward. In the 12-parameter trial, SAC has better training performance than in the 3-parameter case, which is because the former covers a larger action space and consequently provides better solutions of the NMPH optimization problem.

To test the trajectory planning performance of NMPH with and without the proposed adaptation scheme, four different flight tests were performed within the AirSim simulation environment. For the second case, the weighting matrices within NMPH used fixed parameters, which were used as the initial values in the DRL-based adaptation method. Table 1 provides a comparison between the conventional NMPH design with fixed parameter values and the adaptive NMPH-SAC design. The comparison is based on the average of the error metrics discussed in Section 3.1, namely $e_{t,\text{RMS}}$, e_{ss}, and e_c. Each flight trajectory consists of ten trials, and each trial includes five iterations. The initial velocity and drone orientation were selected randomly at the beginning of each trial. The first trial uses a zigzag pattern, which consists of five paths, each with length of 5.6 m. For the second trial (square pattern), the side length was 5 m. For the third trial (ascending square pattern), the elevation gain was set to 1 m. The fourth trial involved a set of position setpoints provided by a graph-based exploration algorithm (see [27] for the complete details). As shown in Table 1, the flight performance obtained with the adaptive NMPH is much better than the one from the non-adaptive (conventional) NMPH. The reason for this is that real-time adaptation of NMPH parameters works better than using a single set of fixed values when performing a variety of different flying trajectories.

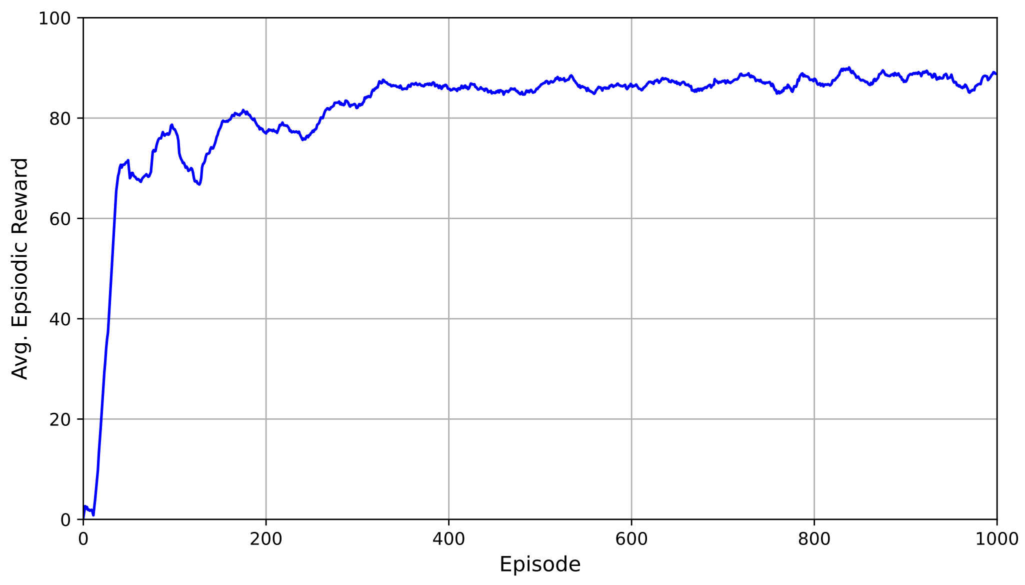


Figure 9. Training curve of SAC adaptively tuning 12 parameters of the NMPH optimization.

Table 1. Comparison between the conventional NMPH design (fixed values of the NMPH parameters) and the adaptive NMPH-SAC approach, for different flight trials.

	Average Error	**Zigzag Pattern**	**Square Pattern**	**Ascending Square Pattern**	**Random Setpoints (Exploration)**
Fixed NMPH parameters	$e_{t,\mathrm{RMS}}$	0.11353	0.09758	0.10741	0.09646
	e_{ss}	0.08659	0.07547	0.07663	0.07339
	e_c	0.12033	0.06426	0.07413	0.07739
Adaptive NMPH-SAC	$e_{t,\mathrm{RMS}}$	0.08877	0.08495	0.09212	0.06749
	e_{ss}	0.01029	0.00919	0.01046	0.01150
	e_c	0.04400	0.04419	0.04952	0.05874

To show how the values of the NMPH parameters are adjusted online using SAC, Figures 10 and 11 present the results of a flight through 20 randomly generated setpoints. Figure 10 depicts the values of the observations v_o, φ, and $|\vec{r}|$ at the beginning of each iteration, and Figure 11 shows the changing values of the NMPH weighting matrix entries. An animation of this test showing the vehicle's flight trajectory and corresponding online calculation outputs is available as a supplementary video file.

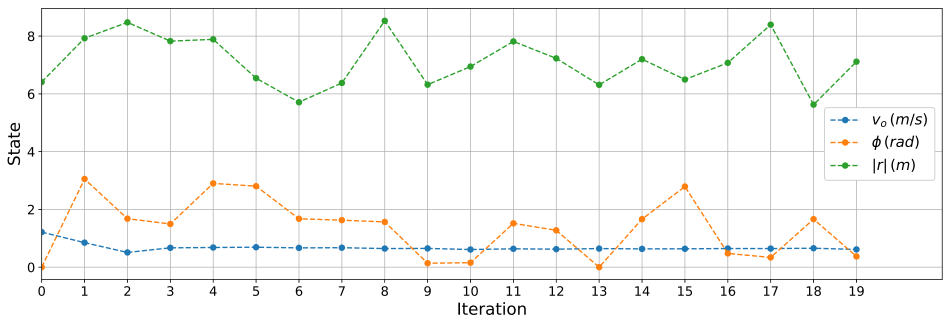


Figure 10. Observations at start of iterations.

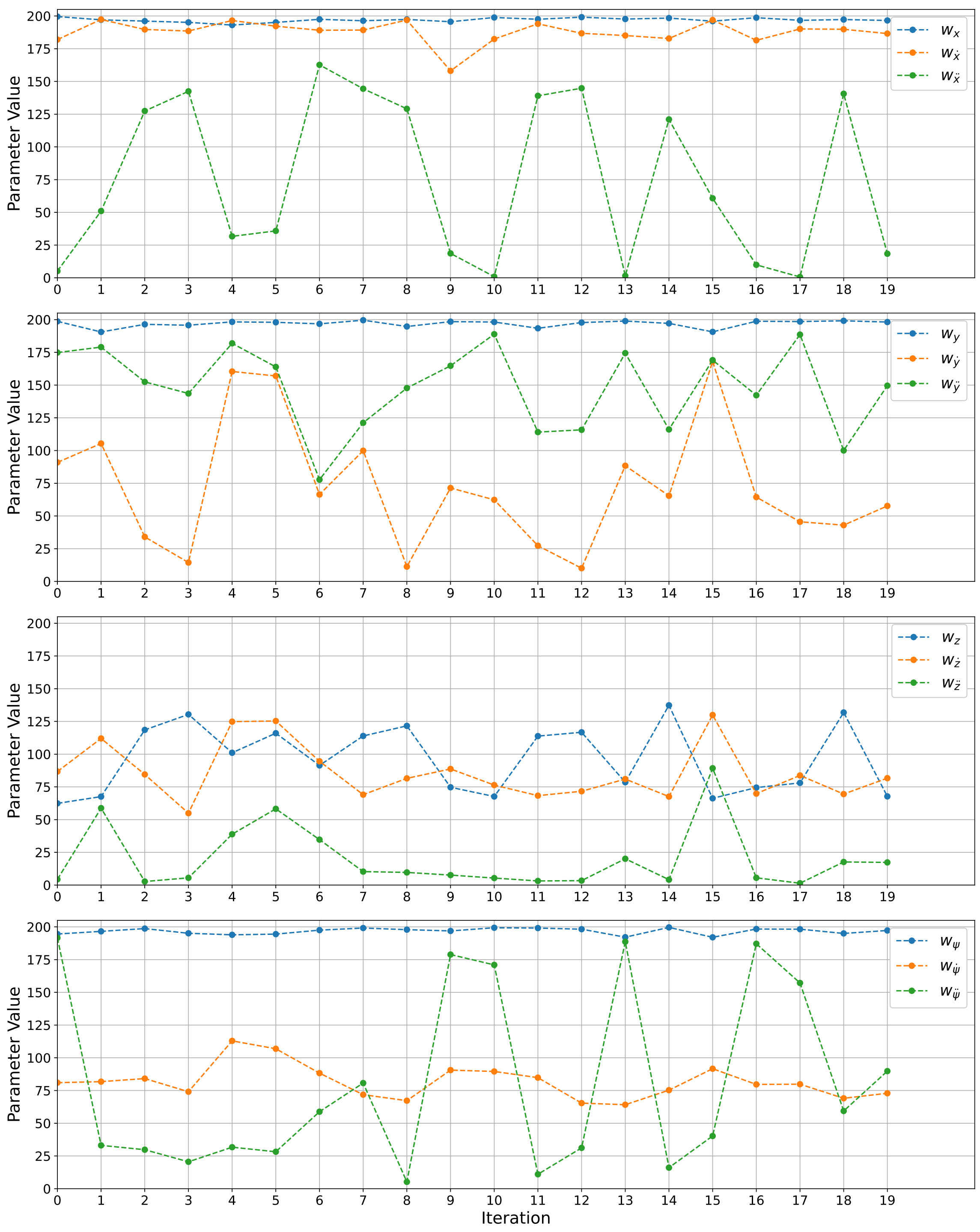


Figure 11. Values of NMPH weighting matrix entries being adjusted online by SAC.

5. Conclusions and Future Work

This paper presented a DRL-based adaptive scheme to tune the optimization parameters of our previously proposed NMPH trajectory planning approach. The overall design aims to provide the best-performing flight trajectory generation for an aerial drone across a wide range of flight patterns and environments by tuning these parameters in real-time flights instead of selecting them a priori. The adaptation scheme was implemented through two different actor-critic DRL algorithms—the deterministic DDPG and the probabilistic SAC.

The two variants of DRL-based NMPH were trained and tested on an aerial drone in a simulation environment. The results showed a marked improvement in flight performance when using the adaptive NMPH-DDPG and NMPH-SAC over the conventional NMPH. Comparisons between DDPG and SAC showed that the latter outperforms the former in terms of learning speed, ability to handle a larger set of tuning parameters, and overall flight performance.

The pros, cons, and limitations of this study are summarized as follows:

- Pros:
    - The proposed design is able to dynamically adjust the parameters of the optimization problem online during flight, which is preferable to tuning them before flight and evaluating the resulting performance afterwards.
    - The DRL model can adapt the gains of the optimization problem in response to changes in the vehicle, such as new payload configurations or replaced hardware components.
- Cons:
    - DRL algorithms employ a large number of hyperparameters. While SAC is less sensitive to hyperparameters than DDPG, finding the best combination of these parameters to achieve fast training is a challenging task.
- Limitations:
    - The present study was performed entirely within a simulation environment and does not include hardware testing results.

Future work will include implementing NMPH-SAC onboard our hardware drone and testing its performance in a variety of real-world environments, as well as using the DRL algorithms for disturbance and parameter estimation.

Supplementary Materials: The following supporting information can be downloaded at: https://www.mdpi.com/article/10.3390/drones6110323/s1.

Author Contributions: Conceptualization, Y.A.Y. and M.B.; methodology, Y.A.Y.; software, Y.A.Y.; validation, Y.A.Y.; formal analysis, Y.A.Y.; investigation, Y.A.Y.; resources, M.B.; data curation, Y.A.Y.; writing—original draft preparation, Y.A.Y.; writing—review and editing, M.B.; visualization, Y.A.Y.; supervision, M.B.; project administration, M.B.; funding acquisition, M.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by NSERC Alliance-AI Advance Program grant number 202102595. The APC was funded by NSERC Alliance-AI Advance Program.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Carlucho, I.; De Paula, M.; Acosta, G.G. An adaptive deep reinforcement learning approach for MIMO PID control of mobile robots. *ISA Trans.* **2020**, *102*, 280–294.
2. Åström, K.J. Theory and applications of adaptive control—A survey. *Automatica* **1983**, *19*, 471–486.
3. Åström, K. History of Adaptive Control. In *Encyclopedia of Systems and Control*; Baillieul, J., Samad, T., Eds.; Springer-Verlag: London, UK, 2015; pp. 526–533.
4. Bellman, R. *Adaptive Control Processes*; A Guided Tour; Princeton University Press: Princeton, NJ, USA, 1961.

5. Gregory, P. *Proceedings of the Self Adaptive Flight Control Systems Symposium*; Technical Report 59-49; Wright Air Development Centre: Boulder, CO, USA, 1959.
6. Panda, S.K.; Lim, J.; Dash, P.; Lock, K. Gain-scheduled PI speed controller for PMSM drive. In Proceedings of the IECON'97 23rd International Conference on Industrial Electronics, Control, and Instrumentation (Cat. No. 97CH36066), New Orleans, LA, USA, 14 November 1997; Volume 2, pp. 925–930.
7. Huang, H.P.; Roan, M.L.; Jeng, J.C. On-line adaptive tuning for PID controllers. *IEE Proc.-Control. Theory Appl.* **2002**, *149*, 60–67.
8. Gao, F.; Tong, H. Differential evolution: An efficient method in optimal PID tuning and on–line tuning. In Proceedings of the First International Conference on Complex Systems and Applications, Wuxi, China, 10–12 September 2006.
9. Killingsworth, N.J.; Krstic, M. PID tuning using extremum seeking: Online, model-free performance optimization. *IEEE Control Syst. Mag.* **2006**, *26*, 70–79.
10. Gheibi, O.; Weyns, D.; Quin, F. Applying machine learning in self-adaptive systems: A systematic literature review. *ACM Trans. Auton. Adapt. Syst. (TAAS)* **2021**, *15*, 1–37.
11. Jafari, R.; Dhaouadi, R. Adaptive PID control of a nonlinear servomechanism using recurrent neural networks. In *Advances in Reinforcement Learning*; Mellouk, A., Ed.; IntechOpen: London, UK, 2011; pp. 275–296.
12. Dumitrache, I.; Dragoicea, M. Mobile robots adaptive control using neural networks. *arXiv* **2015**, arXiv:1512.03345.
13. Rossomando, F.G.; Soria, C.M. Identification and control of nonlinear dynamics of a mobile robot in discrete time using an adaptive technique based on neural PID. *Neural Comput. Appl.* **2015**, *26*, 1179–1191.
14. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*; MIT Press: Cambridge, MA, USA, 2018.
15. Hu, B.; Li, J.; Yang, J.; Bai, H.; Li, S.; Sun, Y.; Yang, X. Reinforcement learning approach to design practical adaptive control for a small-scale intelligent vehicle. *Symmetry* **2019**, *11*, 1139.
16. Watkins, C. Learning from Delayed Rewards. Ph.D. Thesis, King's College, University of Cambridge, Cambridge, UK, 1989.
17. Boubertakh, H.; Tadjine, M.; Glorennec, P.Y.; Labiod, S. Tuning fuzzy PD and PI controllers using reinforcement learning. *ISA Trans.* **2010**, *49*, 543–551.
18. Subudhi, B.; Pradhan, S.K. Direct adaptive control of a flexible robot using reinforcement learning. In Proceedings of the 2010 International Conference on Industrial Electronics, Control and Robotics, Rourkela, India, 27–29 December 2010; pp. 129–136.
19. Barto, A.G.; Sutton, R.S.; Anderson, C.W. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Trans. Syst. Man, Cybern.* **1983**, *13*, 834–846.
20. Lillicrap, T.P.; Hunt, J.J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; Wierstra, D. Continuous control with deep reinforcement learning. *arXiv* **2015**, arXiv:1509.02971.
21. Fujimoto, S.; Hoof, H.; Meger, D. Addressing function approximation error in actor-critic methods. In Proceedings of the International Conference on Machine Learning, PMLR, Stockholm, Sweden, 10–15 July 2018; pp. 1587–1596.
22. Haarnoja, T.; Zhou, A.; Abbeel, P.; Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Proceedings of the International Conference on Machine Learning, PMLR, Stockholmsmässan, Sweden, 10–15 July 2018; pp. 1861–1870.
23. Mnih, V.; Badia, A.P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In Proceedings of the International Conference on Machine Learning, PMLR, Baltimore, MD, USA, 17–23 July 2016; pp. 1928–1937.
24. Sun, Q.; Du, C.; Duan, Y.; Ren, H.; Li, H. Design and application of adaptive PID controller based on asynchronous advantage actor–critic learning method. *Wirel. Netw.* **2021**, *27*, 3537–3547.
25. Al Younes, Y.; Barczyk, M. Nonlinear Model Predictive Horizon for Optimal Trajectory Generation. *Robotics* **2021**, *10*, 90.
26. Al Younes, Y.; Barczyk, M. A Backstepping Approach to Nonlinear Model Predictive Horizon for Optimal Trajectory Planning. *Robotics* **2022**, *11*, 87.
27. Younes, Y.A.; Barczyk, M. Optimal Motion Planning in GPS-Denied Environments Using Nonlinear Model Predictive Horizon. *Sensors* **2021**, *21*, 5547.
28. Dang, T.; Mascarich, F.; Khattak, S.; Papachristos, C.; Alexis, K. Graph-based path planning for autonomous robotic exploration in subterranean environments. In Proceedings of the 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), The Venetian Macao, Macau, 4–8 November 2019; pp. 3105–3112.
29. Oleynikova, H.; Taylor, Z.; Fehr, M.; Siegwart, R.; Nieto, J. Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning. In Proceedings of the 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Vancouver, BC, Canada, 24–28 September 2017; pp. 1366–1373.
30. Liu, R.; Zou, J. The effects of memory replay in reinforcement learning. In Proceedings of the 2018 56th annual allerton conference on communication, control, and computing (Allerton), Monticello, IL, USA, 2–5 October 2018; pp. 478–485.
31. Achiam, J. Spinning Up in Deep Reinforcement Learning. 2018. Available online: https://github.com/openai/spinningup (accessed on 26 October 2022).
32. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; MIT Press: Cambridge, MA, USA, 2016.
33. Duan, Y.; Chen, X.; Houthooft, R.; Schulman, J.; Abbeel, P. Benchmarking deep reinforcement learning for continuous control. In Proceedings of the International Conference on Machine Learning, PMLR, New York, NY, USA, 20–22 June 2016; pp. 1329–1338.

34. Haarnoja, T.; Zhou, A.; Hartikainen, K.; Tucker, G.; Ha, S.; Tan, J.; Kumar, V.; Zhu, H.; Gupta, A.; Abbeel, P.; et al. Soft actor-critic algorithms and applications. *arXiv* **2019**, arXiv:1812.05905v2.
35. Quigley, M.; Conley, K.; Gerkey, B.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; Ng, A.Y. ROS: An open-source Robot Operating System. In Proceedings of the ICRA Workshop on Open Source Software in Robotics, Kobe, Japan, 12–17 May 2009.
36. Shah, S.; Dey, D.; Lovett, C.; Kapoor, A. AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles. In *Field and Service Robotics*; Hutter, M., Siegwart, R., Eds.; Springer: Cham, Switzerland, 2018; pp. 621–635.
37. Houska, B.; Ferreau, H.; Diehl, M. ACADO Toolkit – An Open Source Framework for Automatic Control and Dynamic Optimization. *Optim. Control. Appl. Methods* **2011**, *32*, 298–312.
38. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. Available online: https://tensorflow.org (accessed on 26 October 2022).
39. Chollet, F. Keras. 2015. Available online: https://keras.io (accessed on 26 October 2022).
40. Lai, C.; Han, J.; Dong, H. Tensorlayer 3.0: A Deep Learning Library Compatible with Multiple Backends. In Proceedings of the 2021 IEEE International Conference on Multimedia & Expo Workshops (ICMEW), Shenzhen, China, 5–9 July 2021; pp. 1–3.