Optimal UAV Trajectory Planning for Sum Rate Maximization: A Comparative Study of DQN, DDQN and PPO

by

Yawen Li
B.Eng., North China Electric Power University, 2016
M.Eng., North China Electric Power University, 2019

A Report Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF ENGINEERING

in the Department of Electrical and Computer Engineering

© Yawen Li, 2025
University of Victoria

All rights reserved. This report may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Optimal UAV Trajectory Planning for Sum Rate Maximization: A Comparative Study of DQN, DDQN and PPO

by

Yawen Li
B.Eng., North China Electric Power University, 2016
M.Eng., North China Electric Power University, 2019

Supervisory Committee

Dr. Hong-Chuan Yang Supervisor Main, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Kin Fun Li, Departmental Member
(Department of Electrical and Computer Engineering)

ABSTRACT

This report presents a study on the optimization of unmanned aerial vehicle (UAV) trajectory using advanced reinforcement learning (RL) algorithms, specifically Deep Q-Network (DQN), Double DQN (DDQN), and Proximal Policy Optimization (PPO). The primary objective is to maximize the communication sum rate between the UAV and ground users by formulating it into a Markov Decision Process (MDP).

The study introduces an innovative approach of action elimination to enhance the learning efficiency of RL agents by preventing them from selecting actions that do not contribute to the mission's success. This method proved crucial in helping agents achieve higher rewards and reach their destinations on time, thereby avoiding unnecessary explorations. Additionally, the research explores the impact of different reward functions on the learning dynamics and performance of the RL agents. It was found that while DQN and DDQN are flexible with various reward structures, PPO shows a marked preference for cumulative rewards, reflecting its design to capitalize on long-term benefits.

A significant portion of the research was dedicated to hyperparameter tuning within the PPO framework, where variables such as learning rates, clipping ratios, and buffer sizes were meticulously adjusted to refine the learning process. This tuning not only enhanced the performance of the PPO agent but also offered valuable insights into the sensitivity of RL algorithms to their operational parameters.

However, the study acknowledges limitations, including the simplification of environmental factors and the two-dimensional trajectory optimization. Future work is suggested to integrate more complex environmental models and consider three-dimensional trajectory planning to address real-world applicability more effectively.

The findings from this study contribute to the growing body of knowledge in UAV navigation and RL, providing a pathway for future research to build upon the foundational results obtained.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

my spouse, Zhi Li, and my parents, for supporting me in the low moments.

Supervisor Hong-Chuan Yang, for mentoring, support, encouragement, and patience.

I believe I know the only cure, which is to make one's centre of life inside of one's self, not selfishly or excludingly, but with a kind of unassailable serenity-to decorate one's inner house so richly that one is content there, glad to welcome any one who wants to come and stay, but happy all the same in the hours when one is inevitably alone.

Yawen Li

DEDICATION

Just hoping this is useful!

Chapter 1

Introduction

1.1 Study Background and Motivation

Recent advancements in Unmanned Aerial Vehicles (UAVs) have revolutionized various sectors, enabling new capabilities from military applications to civil services [1, 2]. Particularly, UAV-mounted base stations (UAV-BS) have emerged as a versatile solution to enhance wireless network coverage and manage network traffic, especially in scenarios beyond 5G [3, 4]. These aerial platforms offer rapid deployment and cost-effective network support in areas unreachable by traditional infrastructure, often establishing direct line-of-sight communications that can support high data transmission rates [5].

However, integrating UAVs into cellular networks introduces several challenges, such as ensuring reliable connectivity across different terrains and altitudes, managing interference, optimizing handover strategies, and maintaining energy efficiency [6, 7, 8]. Moreover, effective navigation and path planning are critical due to the limited battery capacities of UAVs.

Previous works have explored various aspects of UAV deployment. Al-Hourani et al. (2014) [9] developed a probabilistic line-of-sight (LoS) channel model and optimized UAV altitude to maximize ground coverage, necessitating prior environmental knowledge. Mozaffari et al. (2016) [10] and Ono et al. (2016) [11] proposed location optimizations limited to predetermined points, thereby restricting potential performance gains.

In related work, researchers addressed imbalances in transmission rates between RF and FSO links by designing UAV relay node trajectories that maximize data

throughput at the ground user terminal [12]. They categorized transmission schemes into delay-limited and delay-tolerant types, proposing an iterative algorithm to achieve locally optimal throughput solutions while considering atmospheric conditions, buffer sizes, and delay requirements.

Despite these advances, many methods still rely on complex computations and predefined environmental parameters. For example, Zeng (2021) [13] developed a novel UAV navigation algorithm using Double Deep Q-Network (DDQN) with multi-step learning to predict outage probabilities across various locations, aiming to minimize the total mission completion time and expected communication outage time. However, these approaches often overlook the constraints imposed by UAV battery life, which can limit their practical utility.

To optimize the trajectory of UAV, we have employed model-free reinforcement learning (RL) algorithms, namely Deep Q-Network (DQN), DDQN, and Proximal Policy Optimization (PPO). These algorithms enable the UAV to optimize its data transmission strategy by dynamically navigating closer to target users, thereby establishing robust communication links. Moreover, this strategy ensures the UAV can efficiently manage its power and return to its charging station before depleting its battery.

1.2 Project Contributions and Findings

This project contributes to the field by focusing on optimal trajectory design and transmit sum rate maximization. We have employed advanced RL approaches—specifically DQN, DDQN, and PPO—to effectively address the complex challenges associated with UAV navigation and communication:

- **Markov Decision Process Formulation:** We have formulated the UAV trajectory optimization problem as the Markov Decision Process (MDP), concentrating on maximizing communication sum rates. This formulation allows for a structured and systematic approach to decision-making under uncertainty, enhancing the strategic depth of UAV navigation and communication strategies.

- **Action Elimination Efficacy:** Our study demonstrates the effectiveness of action elimination in refining the learning process. By enabling the UAV to avoid inefficient paths, this approach significantly enhances overall performance, ensuring more reliable and efficient mission outcomes.

- **Comparative Analysis of Reward Functions:** We have compared the efficacy of different reward functions across the employed RL algorithms. Our findings indicate that while DQN and DDQN exhibit adaptability across various reward configurations, PPO shows a distinct preference for cumulative rewards, aligning closely with its algorithmic design which favors long-term benefit maximization.
- **Hyperparameter Impact on PPO:** The project has thoroughly investigated the impact of various hyperparameters on the learning efficacy of PPO. Through extensive experimentation, we provide a detailed guide that outlines the critical hyperparameters influencing learning dynamics, offering invaluable insights for future applications of RL in similar domains.

These advancements not only enhance the theoretical understanding of RL application in UAV systems but also provide practical insights that can be leveraged to improve real-world UAV operations. The successful implementation of these techniques showcases the potential of RL to revolutionize the efficiency and effectiveness of UAV-based communication networks.

1.3 Report Structure

The report is organized as follows to systematically present the research and validate the claims made:

Chapter 1 Introduction—outlining the research background, challenges, and scope.

Chapter 2 System Model and Problem Formulation—detailing the technical setup and theoretical models.

Chapter 3 Methodology—describing the RL algorithms and proposed solutions.

Chapter 4 Experiments and Analysis—presenting comparative studies on action elimination and reward function efficacy.

Chapter 5 Results and Discussion—evaluating the outcomes and optimizing PPO hyperparameters.

Chapter 6 Conclusions and Future Work—summarizing the research contributions and suggesting directions for further investigation.

Chapter 2

System Model and Problem Formulation

2.1 System Model

Our model includes a UAV-assisted wireless network depicted in Figure 2.1, wherein a UAV serves as a mobile base station (BS) to cater to a group of K terrestrial users. The UAV maintains a constant altitude H during its flight duration T. We utilize a Cartesian coordinate system where the position of each user k, belonging to the set $\mathcal{K} \triangleq 1, \ldots, K$, is denoted as $\mathbf{w}_k = (x_k, y_k)^T$ and is part of $\mathbb{R}^{2\times 1}$. Initially, these user locations are not known to the UAV. The UAV's mission is to travel from its initial location to a designated final destination, strategically optimizing data transmission by moving closer to key users within specified time constraints.

The starting and ending horizontal positions of the UAV are defined as $\mathbf{q}_I = (x_I, y_I)^T$ and $\mathbf{q}_F = (x_F, y_F)^T$ respectively. To model the UAV's flight, we break down the total flight time T into N equal intervals, each lasting $\delta_t = \frac{T}{N}$, assuming that δ_t is small enough to consider the UAV's position nearly constant within each interval. During each interval, the UAV may remain stationary or travel at a consistent velocity V in one of four cardinal directions. The UAV's horizontal position at the start of each interval n is given by $\mathbf{q}_n = (x_n, y_n)^T \in \mathbb{R}^{2\times 1}$, where $n \in \mathcal{N} \triangleq 1, \ldots, N$. The movement of the UAV over time is described by:

$$\mathbf{q}_1 = \mathbf{q}_I, \quad \mathbf{q}_{N+1} = \mathbf{q}_F, \tag{2.1}$$

$$\mathbf{q}_{n+1} = \mathbf{q}_n + V\delta_t \mathbf{v}_n, \quad n = 1, \ldots, N, \tag{2.2}$$

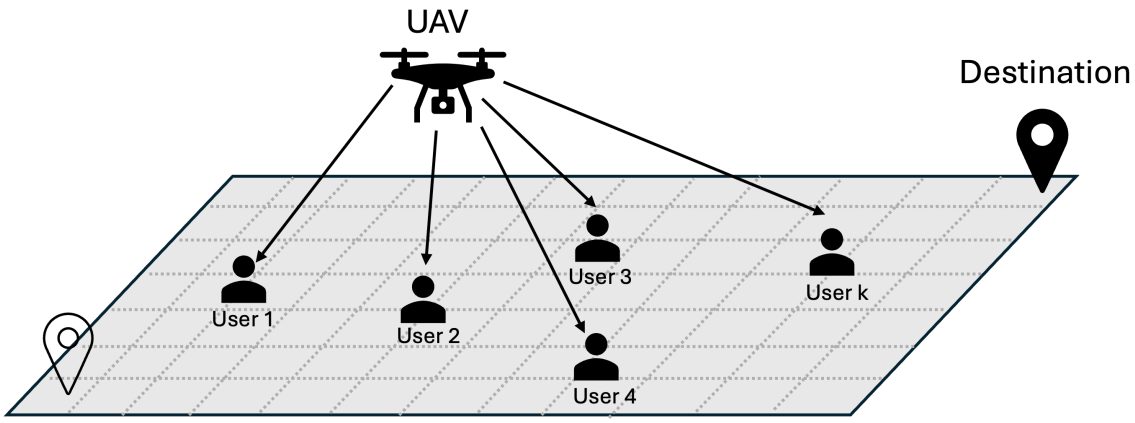


Figure 2.1: An illustration of downlink data transmission in a UAV-assisted wireless network with multiple users

where $\mathbf{v}_n \in (1,0)^T, (-1,0)^T, (0,1)^T, (0,-1)^T, (0,0)^T$ indicates the direction of the UAV's movement during each time slot n. This movement pattern defines the UAV's trajectory.

Channel gains are crucial for assessing the quality of communication between the UAV and each ground user. The actual distance, which accounts for both horizontal displacement and altitude, is expressed as:

$$d_{k,n} = \sqrt{\|\mathbf{w}_k - \mathbf{q}_n\|^2 + H^2}, \tag{2.3}$$

Following the free-space path loss model[14], the channel gain $h_{k,n}$ between the UAV and a ground user k at a particular time slot n is thus modeled by:

$$h_{k,n} = \frac{\beta_0^{\text{linear}}}{d_{k,n}^2}, \tag{2.4}$$

where β_0^{linear} represents the basic gain of the communication channel under ideal conditions.

We adopt Frequency Division Multiple Access (FDMA)[15] in this work. In FDMA, the total available bandwidth is equally divided among multiple users, allowing each to transmit simultaneously without interference within their designated frequency bands. Accordingly, the system bandwidth B is equitably distributed among all K users, providing each with an equal slice of the bandwidth. This allocation is

mathematically represented as:

$$B(k) = \frac{B}{K} \tag{2.5}$$

where $B(k)$ denotes the bandwidth allocated to the k-th user.

The Signal-to-Noise Ratio (SNR) for each user at each time slot is calculated as:

$$\mathrm{SNR}_{k,n} = \frac{P_{\text{linear}} \cdot h_{k,n}}{N_0 \cdot B(k)}, \tag{2.6}$$

where P_{linear} is the transmit power in Watts, and N_0 is the noise power spectral density. Applying Shannon's theorem[16], the achievable rate for each user k at time slot n is given by:

$$R_{k,n} = B(k) \cdot \log_2(1 + \mathrm{SNR}_{k,n}), \tag{2.7}$$

and the cumulative sum rate at each time slot is:

$$R_{\text{sum},n} = \sum_{k=1}^{K} R_{k,n}. \tag{2.8}$$

This dynamic multi-user scenario, where the UAV's trajectory impacts the sum rate calculation, illustrates the need for real-time path adjustments to maximize communication rates.

2.2 Problem Formulation

In this project, our goal is to maximize the cumulative data rate by optimizing the trajectory of the UAV, which is determined by the directional choices $\mathbf{v}_n$ for each time slot.

We define the set of all possible directions as $\mathbf{V} \triangleq \{\mathbf{v}_n, \forall n\}$, and set up our

optimization problem as follows:

$$
\begin{aligned}
&\text{(P1)}: \max_{\mathbf{V}} P_1 \\
&\text{where} \quad P_1 = \sum_{n=1}^{N} R_{\text{sum},n}, \\
&\text{subject to} \qquad \mathbf{q}_1 = \mathbf{q}_I,\ \mathbf{q}_{N+1} = \mathbf{q}_F, \\
&\qquad\qquad\qquad\ \mathbf{q}_{n+1} = \mathbf{q}_n + \Delta t \mathbf{v}_n, \quad \forall n, \\
&\qquad\qquad\qquad\ \mathbf{v}_n \in \{(1,0)^T, (-1,0)^T, (0,1)^T, (0,-1)^T, (0,0)^T\}, \quad \forall n,
\end{aligned}
\tag{2.9}
$$

where $\Delta t = V\delta_t$ represents the displacement of the UAV in each time slot. Because problem (P1) is non-convex and all optimization variables are linked, solving it with typical optimization techniques presents substantial challenges[17]. Furthermore, because the UAV can only move in five discrete directions, (P1) is a mixed-integer non-convex optimization problem that is difficult to solve. The computational demands of a brute-force solution become unfeasible as the number of time slots increases exponentially, even if perfect knowledge of the channel state is assumed. We use a model-free reinforcement learning (RL) strategy to overcome these constraints, allowing the UAV to intelligently plan its fly path based on real-time input on signal strengths received.

It is important to convert a given problem into a Markov Decision Process (MDP), where an agent learns the optimal actions by interacting directly with the environment, in order to solve a real-world problem such as (P1) using an RL method. The position and flying direction of the UAV at time slot n determine its position at time slot $n+1$. As a result, (P1) can be viewed as a discrete-time MDP in which the UAV assumes the role of an agent that gains the ability to maximize the total rate by setting its trajectory. This is accomplished by transforming problem (P1) into a MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ in the manner described below:

- $\mathcal{S}$ **state space:** We use UAV's current position on the horizontal plane, represented by $\mathbf{q}_n$, and the current time step, n. These elements form the state space $\mathcal{S}$, ensuring the UAV remains close to its destination as the flight time concludes. The state at any time n is thus defined as $s_n = \{\mathbf{q}_n, n\} \in \mathcal{S}$ with a dimension of 3, combining spatial and temporal data to guide the UAV's trajectory planning.

- $\mathcal{A}$ **action space:** At each time step n, the UAV must select an optimal flight path to maximize the sum rate. Its possible flying directions—left, right, up, down, and hover—constitute the action space $\mathcal{A}$, represented by $\mathbf{v}_n$.

- $\mathcal{P}$ **state transition:** The flying direction chosen in the present state determines the deterministic state transition, which leads to the next state.

- $\mathcal{R}$ **Reward:** In order to maximize the total rate and arrive at the destination by the end of the flight period, we should create a suitable reward. The reward obtained after carrying out the action a_n in state s_n and changing to state s_{n+1} is therefore defined as

$$\mathcal{R}(s_n, a_n) = \frac{R_{\text{sum},n}}{10} \tag{2.10}$$

  Initially, our evaluation of rewards was based on the reduction in the distance between the UAV's current position s_n, and its subsequent position s_{n+1}, relative to the final destination. If the UAV moved closer to the destination, it received a positive reward. However, the implementation of action elimination, which considers the distance to the destination and the required time using the Manhattan distance[18] between $\mathbf{q}_n$ and $\mathbf{q}_F$, calculated as $\text{manh}(\mathbf{q}_n, \mathbf{q}_F) = |x_n - x_F| + |y_n - y_F|$, determined the number of steps needed for the UAV to reach the final location by $\left\lceil \frac{\text{manh}(\mathbf{q}_n, \mathbf{q}_F)}{V\delta_t} \right\rceil$. This measure ensures that actions are selected to guarantee timely arrival at the destination, thereby removing the need for specific rewards for on-time arrival and distance reduction.

  The decision to normalize $R_{\text{sum},n}$ by dividing by 10 aims to smooth the learning process by scaling the reward magnitude. Furthermore, the MDP formulation corresponds to episodic tasks since each episode ends when the UAV reaches the terminal state, i.e., $\mathbf{q}_F$, or when the maximum number of time steps is reached.

Chapter 3

Adopted Algorithms and Proposed Solutions

We have turned to a model-free RL strategy, deploying algorithms, i.e., DQN, DDQN, and PPO. These algorithms enable the UAV to adjust its trajectory dynamically based on immediate feedback on signal strengths. This real-time adaptation occurs without the need for a predefined model, allowing for greater flexibility in response to changing conditions.

We have developed an approach that refines the learning mechanism. By reducing the action space and adjusting the reward signals, we can boost the efficiency and effectiveness of our RL algorithms. These modifications not only make the decision-making process more straightforward but also speed up the UAV's ability to adapt and optimize its route through complex environments.

3.1 Adopted Algorithms

3.1.1 Value-based Reinforcement Learning: DQN

DQN is a robust method in reinforcement learning that blends deep learning techniques with traditional Q-learning[19]. This combination enables agents to navigate and adapt strategies within complex environments effectively. At its core, DQN uses state representation, where it transforms the current environment state into a numerical format, such as processed features. It employs a

convolutional neural network (CNN), designed to take these state inputs and output Q values for each possible action.

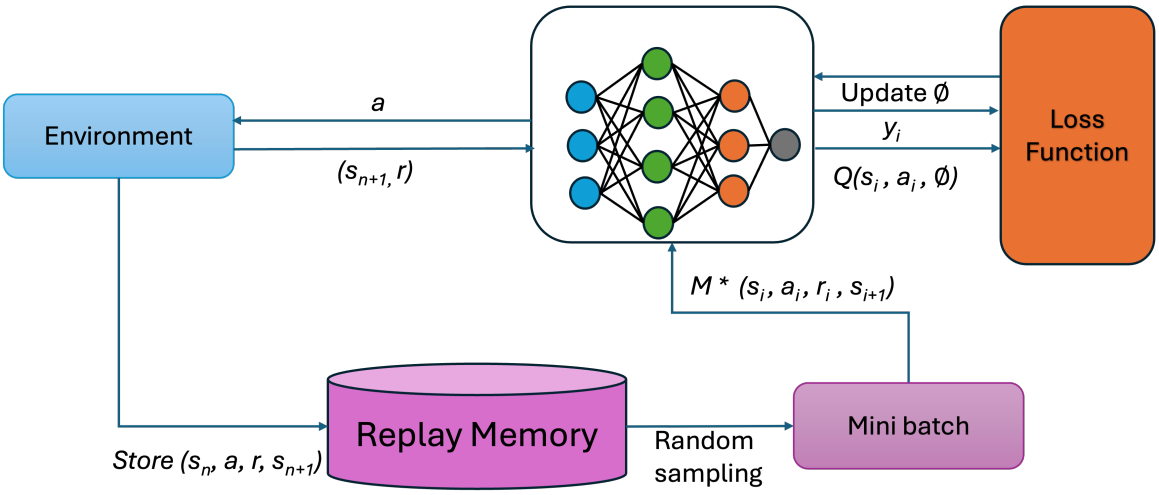


Figure 3.1: DQN algorithm with experience replay operational flow chart

Figure 3.1 illustrates the operational flow of the DQN algorithm incorporating an experience replay mechanism, which is pivotal in modern reinforcement learning frameworks. At the beginning of learning, the critic Q-function is initialized as $Q(s, a; \phi)$, with random parameters ϕ.

For each training timestep:

1. For the current state s_n, select an action a using an ϵ-greedy policy:

$$a = \begin{cases} \text{random action with probability } \epsilon, \\ \arg\max_a Q(s_n, a; \phi) \text{ with probability } 1 - \epsilon. \end{cases} \tag{3.1}$$

2. Execute action a, observe the reward r and the next state s_{n+1}.
3. Store the transition (s_n, a, r, s_{n+1}) in the replay memory buffer.
4. Sample a random mini-batch of M transitions (s_i, a_i, r_i, s_{i+1}) from the experience buffer.
5. For each sampled transition:
   - If s_{i+1} is a terminal state, set the value function target:

$$y_i = r_i \tag{3.2}$$

– Otherwise, set the target using:

$$y_i = r_i + \gamma \max_{a_{i+1}} Q(s_{i+1}, a_{i+1}; \phi) \tag{3.3}$$

6. Update the critic parameters ϕ by minimizing the loss L over all sampled experiences:

$$L = \frac{1}{2M} \sum_{i=1}^{M} (y_i - Q(s_i, a_i; \phi))^2 \tag{3.4}$$

7. Periodically update the critic network parameters ϕ.
8. Adjust ϵ according to its decay rate to reduce the exploration over time.

The algorithm efficiently balances exploration (via ϵ-greedy action selection) and exploitation (by updating the Q-network based on sampled experiences).The use of an experience replay buffer mitigates the correlations between consecutive learning updates.

3.1.2 Value-based Reinforcement Learning: DDQN

As highlighted in [20], DQN is typically structured for offline batch learning and is less adept at handling continuous online learning scenarios. The Q-learning update within DQN tends to overestimate action values, which can skew the accuracy of the policies it learns[21]. This overestimation stems from using the max operator to both choose and evaluate actions, resulting in potentially inflated value estimates, thereby introducing noise and inconsistency into the learning process.

The DDQN improves upon the standard DQN by using two sets of weights: one for the current policy (main network) and one for the target policy (target network). This division helps to mitigate the risk of overestimating action values. While the main network may still produce higher Q-value estimates, the target network offers a more conservative appraisal for updates.

Figure 3.2 illustrates the operational flow of the DDQN algorithm incorporating an experience replay mechanism. Both the main network and the target network start with identical parameters, ϕ, ensuring $\phi_t = \phi$. This setup helps avoid initial bias due to divergent behaviors between the networks.

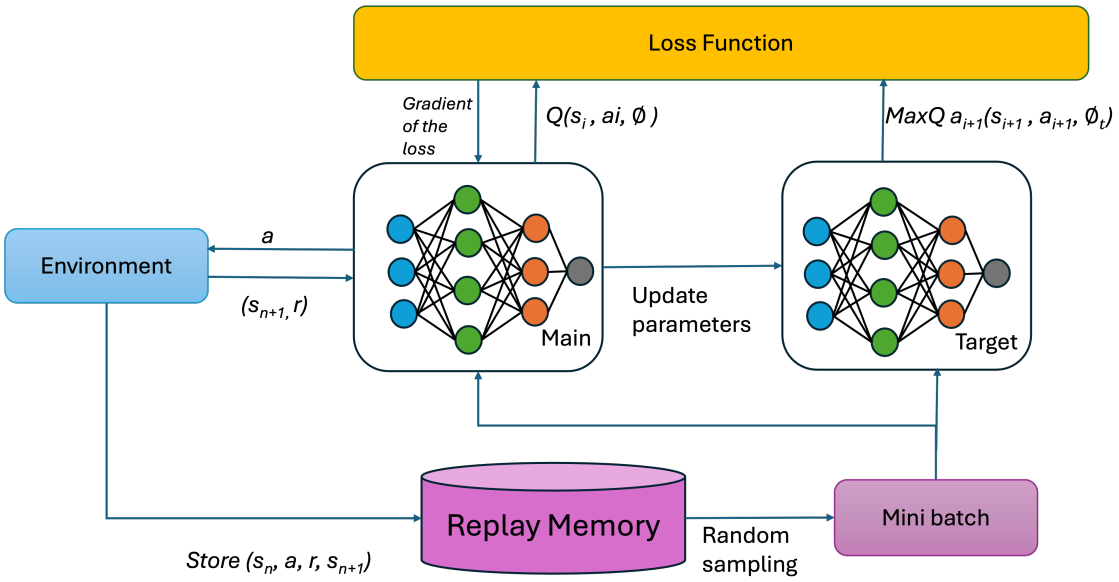


Figure 3.2: DDQN algorithm with experience replay operational flow chart

For each training timestep:

1. Steps 1-4 are similar to DQN, actions are selected using an epsilon-greedy policy based on the Q-values predicted by the main network.
2. When the next state s_{i+1} is non-terminal, the value function target for the DDQN update is given by:

$$y_i = r_i + \gamma Q(s_{i+1}, \arg\max_a Q(s_{i+1}, a; \phi); \phi_t) \tag{3.5}$$

   Here, the action a that maximizes the Q-value at state s_{i+1} is chosen using the current parameters ϕ, but the value is estimated using the target network parameters ϕ_t.
3. Critic Parameters: The loss L is minimized across all sampled experiences, where:

$$L = \frac{1}{2M} \sum_{i=1}^{M} (y_i - Q(s_i, a_i; \phi))^2 \tag{3.6}$$

4. Target Network Update: Periodically update the target network weights ϕ_t to align with the main network ϕ.
5. Adjust the exploration probability ϵ based on a decay schedule to enhance learning efficiency.

3.1.3 Policy-Based Reinforcement Learning: PPO

PPO is a policy gradient method and can be used for environments with either discrete or continuous action spaces[22]. It trains a stochastic policy in an on-policy way. Also, it utilizes the actor critic method. The theoretical category of PPO is illustrated in Figure 3.3, which demonstrates the classification of RL.

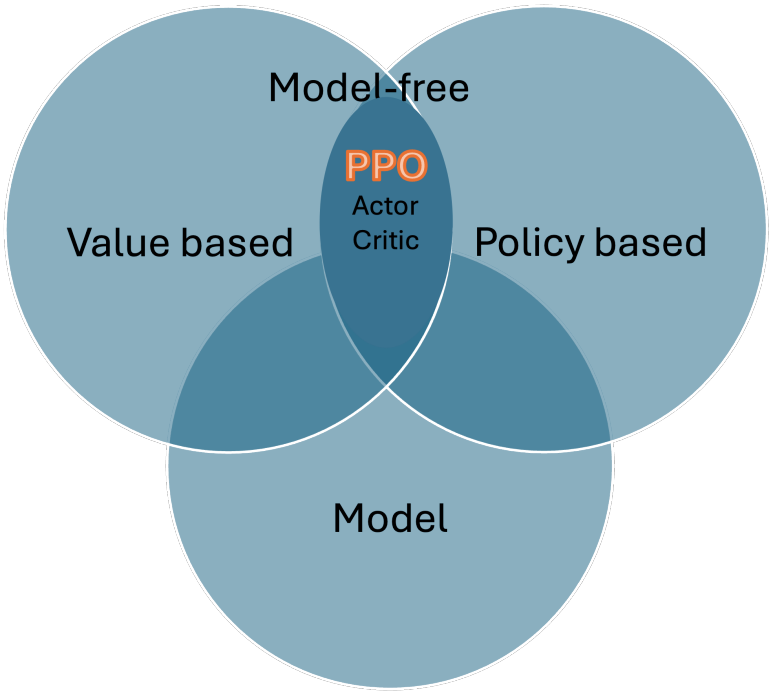


Figure 3.3: The classification of RL

PPO employs two principal networks:

1. **Actor Network:** With parameters θ, the actor outputs the probability distribution of actions given the current state. This network decides on actions by analyzing the state, producing probabilities for potential actions, and facilitating decision-making that accounts for environmental conditions.
2. **Critic Network:** With parameters ϕ, the critic evaluates the input state s and returns the corresponding expectation of the discounted long-term reward. This estimation helps predict the benefits of actions from given states, providing a standard for minimizing variance in predictions.

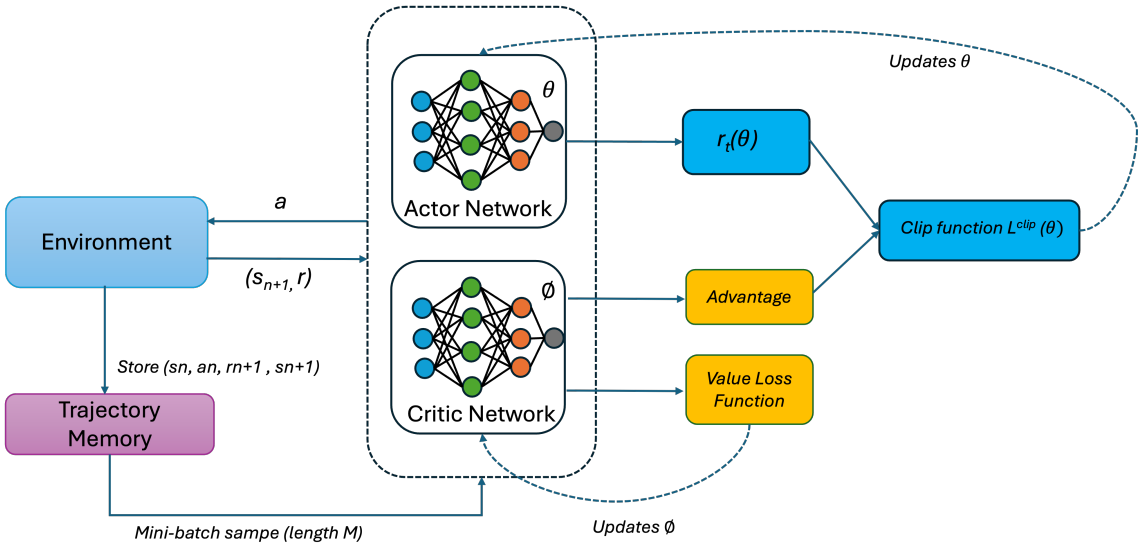


Figure 3.4: PPO algorithm training process flow

During the training process shown in Figure 3.4, PPO interacts with the environment through the current policy to collect data. The agent computes the probability of taking each action in the action space using $\pi(a|s;\theta)$ and selects an action a based on this probability distribution. Actions are executed in the environment to observe the resulting next states, rewards, and terminal flags.

The key steps in the PPO training loop are as follows:

1. **Data Collection:** For each episode, data is collected for a batch of steps from $t = t_s$ to $t_s + N - 1$. The return G_t and the value estimates are computed using:

$$G_t = R_t + \gamma G_{t+1} \tag{3.7}$$

   If s_{t_s+N} is not a terminal state, include the value of the next state, estimated by the critic network V_ϕ, in the discounted future reward:

$$G_t = R_t + \gamma V_\phi(s_{t_s+N}). \tag{3.8}$$

2. **Advantage Calculation:** The advantage function D_t is calculated as:

$$D_t = G_t - V_\phi(s_t). \tag{3.9}$$

3. **Mini-batch Sampling:** A mini-batch of size M is sampled from the

collected data. Each sample contains an experience tuple (s_i, a_i, R_t, s_{i+1}) along with the computed return and advantage.

4. **Critic Update:** The parameters of the critic network are updated by minimizing the loss across the sampled mini-batch:

$$L_{\text{critic}} = \frac{1}{2M} \sum_{i=1}^{M} (y_i - V_\phi(s_i))^2, \tag{3.10}$$

where y_i is the sampled return.

5. **Actor Update:** The actor parameters are updated by minimizing the clipped surrogate objective:

$$L_{\text{actor}}(\theta) = \frac{1}{M} \sum_{i=1}^{M} \min\left(r_i(\theta) \cdot D_i, \text{clip}(r_i(\theta), 1-\epsilon, 1+\epsilon) \cdot D_i\right), \tag{3.11}$$

where:

- $r_i(\theta) = \frac{\pi(A_i|S_i;\theta)}{\pi(A_i|S_i;\theta_{\text{old}})}$ is the ratio of the new policy probability to the old policy probability for taking action A_i in state S_i.
- D_i is the advantage estimate at the i-th step.
- ϵ is a hyperparameter defining the clipping range to limit the ratio $r_i(\theta)$, keeping the policy updates within reasonable bounds to ensure stable training.

Repeat these steps until the environment returns a terminal state or a predefined number of steps are completed. This structured approach helps in stabilizing the learning updates and improving the efficiency of the policy optimization.

3.2 Action Elimination

In our framework, we optimize UAV navigation by incorporating action elimination strategies into the reinforcement learning models DQN and PPO. This technique is crucial for ensuring that the UAV does not select actions that would lead it to unreachable destinations within the allotted time frame.

During the action elimination process, we evaluate the validity of each proposed action by comparing the time required to execute the action against the time

remaining for the operation. Specifically:

$$\text{if } t_{\text{required}} \leq t_{\text{left}}, \text{ then append the action to the state's valid actions.} \quad (3.12)$$

- DQN and DDQN Implementation: The Q-values are updated based on the Bellman equation during the training process, where we incorporate checks to ensure that only valid actions, those that the UAV can execute within the remaining time, are considered. This reduction in the action space simplifies the decision-making process, enabling more efficient learning and faster convergence to optimal policies.

- PPO Implementation: In PPO, policy optimization is more directly influenced by the action elimination strategy. During each epoch of training, the policy network suggests actions based on the current state, and the feasibility of each action is evaluated via (3.12). Actions that are deemed unfeasible—those that do not allow the UAV to reach its destination within the available time—are filtered out, and their probabilities are adjusted to zero in the action distribution. This selective consideration of actions ensures that the policy gradient updates focus only on viable actions. Furthermore, the advantage calculation in PPO, which helps in estimating how much better an action is compared to the average, is crucial for emphasizing efficient routes and maneuvers.

  This structure encourages the UAV to prioritize routes and actions that are not just locally optimal but also globally beneficial, ensuring it remains on track to reach its destination efficiently. The integration of action elimination thus plays a pivotal role in enhancing the operational efficiency of UAVs in dynamic environments, allowing these models to adapt more effectively to the constraints imposed by the mission timeline.

Chapter 4

Experiments

4.1 Experimental Validation

To validate the efficacy of our proposed solution, we have designed two sets of comparisons:

4.1.1 Comparison Set 1: Effectiveness of Action Elimination in UAV Navigation

In our experimental setup, we addressed the challenge of redundant or invalid actions by implementing an 'Action Elimination' strategy. This method was designed to exclude actions that would prevent the UAV from reaching its destination within the available time, thus refining the decision-making process and enhancing the efficiency of learning. By eliminating these suboptimal actions, we expected the UAV to focus on more viable paths, improving both its performance and learning speed.

4.1.2 Comparison Set 2: Evaluating Reward Function Efficacy in DQN, DDQN, and PPO

The first reward function, focused on cumulative gains, rewards agents based on the aggregated success across all actions, promoting strategies that optimize long-term benefits. The second function, based on incremental improvements,

rewards agents for positive changes between consecutive actions, encouraging responsiveness and immediate adaptations to the environment.

By applying these reward functions across different reinforcement learning architectures, we aim to uncover nuanced insights into how each algorithm adapts to and is potentially optimized by different motivational frameworks. The findings from these experiments are expected to shed light on the suitability of each RL approach for various operational contexts, guided by the effectiveness of the reward strategies employed.

4.2 DQN and DDQN Implementation Details

Step 1: Environment Construction A 300x300 grid environment is constructed where the agent's task is to navigate efficiently while maximizing the total sum rate. The users' locations are generated randomly. Environment and UAV Settings are displayed in Table 4.1. We made the total flying duration larger than the time required (60 seconds), which encouraged agent to explore more in the learning phase.

Parameter	Value
Path Loss Exponent	2
UAV Height (m)	100
Number of Users	10
Transmit Power (dBm)	45
System Bandwidth (Hz)	10^6
Noise Spectral Density (dBm/Hz)	-135
Reference Channel Gain (dB)	-60
Maximum Speed (m/s)	10
Time Slot Length (s)	1
Total Flying Duration (s)	80

Table 4.1: UAV and Environment Settings

Step 2: Memory Buffer Construction Experience Replay is a technique in reinforcement learning that allows an agent to learn from a batch of past experiences. Each experience, encapsulating the agent's current state, action, reward, and subsequent state, is stored in a memory buffer. This technique

```python
def memorize(self, state, action, reward, done, new_state, error=None):
    """ Save an experience to memory, optionally with its TD-Error
    """

    experience = (state, action, reward, done, new_state)

    # Check if buffer is already full
    if self.count < self.buffer_size:
        self.buffer.append(experience)
        self.count += 1
    else:
        self.buffer.popleft()
        self.buffer.append(experience)
```

Figure 4.1: Memory buffer construction in DDQN python codes

helps the agent to recall and learn from these experiences, improving learning efficiency and stability.

Step 3: Target Network Construction The neural network is fundamental in learning the action-value function in DQN and DDQN. The structure of the network varies slightly as showed in Figure 4.2 depending on whether a standard or dueling architecture is employed.

```python
def network(self, dueling):
    """ Build Deep Q-Network
    """
    inp = Input((self.state_dim[0],))

    # D
    x = Dense(64, activation='relu')(inp)
    x = Dense(64, activation='relu')(x)

    if (dueling):
        # Have the network estimate the Advantage function as an intermediate layer
        x = Dense(self.action_dim + 1, activation='linear')(x)
        x = Lambda(lambda i: K.expand_dims(i[:, 0], -1) + i[:, 1:] - K.mean(
            i[:, 1:], keepdims=True), output_shape=(self.action_dim,))(x)
    else:
        x = Dense(self.action_dim, activation='linear')(x)
    return Model(inp, x)
```

Figure 4.2: DQN and DDQN network construction

The standard DQN consists of the following layers:

- **Input Layer:** This creates an input layer that accepts states of dimension 3.
- **Hidden Layers:** Two dense layers with 64 units each and ReLU activation functions. These layers learn to extract features from the input state.
- **Output Layer:** This is a standard DQN output layer, directly estimating Q-values for each action.

In the dueling architecture variant, the network has an additional branch that separately estimates the state value and the advantages for each action:

- **Input and Hidden Layers:** Same as in the standard architecture.
- **Dense Layer:** The first dense layer outputs action values.
- **Lambda Layer:** The Lambda layer separates these into two parts:
  * The state value $V(s)$, which is the first output.
  * The advantage values $A(s, a)$ for each action, which are the remaining outputs.
  * Combination Formula: The Q-value for each action is computed using the formula:

$$Q(s, a) = V(s) + (A(s, a) - \text{mean}(A(s, a))) \tag{4.1}$$

    This formula adjusts the advantage values by subtracting their mean, effectively normalizing them around zero and stabilizing the training process by reducing the variance of the advantage estimates.

Step 4: Main Network Construction In DDQN, the main network determines the optimal action with the highest Q-value for a given state. The architecture of the main network mirrors that of the target network.

Step 5: Agent Construction and Training

The training hyperparameters are displayed in Table 4.2.

In the DDQN framework, we instantiated a single agent with two neural networks: the main network, which is used for the ongoing learning and action selection, and the target network, which helps stabilize the updates by providing a more constant approximation of the Q-values. The agent trains using

Parameter	Value
Replay Memory Size	20000
Batch Size	256
Target Update Frequency	80
Learning Rate	1e-3
Maximum Episodes	2000
Epsilon Start Value	1.0
Minimum Epsilon	0.01
Epsilon Decay Rate	0.995

Table 4.2: DQN and DDQN training hyperparameters

batches of experiences stored in the replay buffer shown in Figure 4.3. But in DQN, there is only one network, which is used for action selection and evaluation shown in Figure 4.3 second condition.

The target network's weights are updated periodically to match those of the main network every few episodes shown in Figure 4.4, rather than continuously after every single step. This setup mitigates the risk of large updates that could destabilize the learning process.

4.3 PPO Implementation Details

PPO is implemented using a structured approach, involving key components such as memory for storing experiences, actor and critic networks, and training protocols.

Step 1: Environment Construction This step is same to the DQN and DDQN implementation, where the agent interacts with an environment designed for training reinforcement learning algorithms.

Step 2: Memory Construction In PPO, memory plays a critical role by storing trajectories, which encompass sequences of states, actions, rewards, values, and log probabilities. This mechanism enables the agent to learn from extensive past experiences. Unlike in DQN or DDQNs, where experience replay randomly samples disjointed experiences from a buffer, PPO's memory is specifically designed for on-policy learning. This means that PPO continuously updates its policy using data collected from the policy currently in use, thus requiring the storage of complete trajectories to effectively compute advantage estimates and

```python
def train_agent(self, batch_size):
    """ Train Q-network on batch sampled from the buffer
    """
    # Sample experience from memory buffer (optionally with PER)
    s, a, r, d, new_s, idx = self.buffer.sample_batch(batch_size)

    # Apply Bellman Equation on batch samples to train our DDQN
    q = self.agent.predict(s)
    next_q = self.agent.predict(new_s)
    q_targ = self.agent.target_predict(new_s)

    for i in range(s.shape[0]):
        old_q = q[i, a[i]]
        if d[i]:
            q[i, a[i]] = r[i]
        if not self.dueling:
            next_best_action = np.argmax(next_q[i, :])
            q[i, a[i]] = r[i] + self.gamma * next_q[i, next_best_action]
        else:
            next_best_action = np.argmax(next_q[i, :])
            q[i, a[i]] = r[i] + self.gamma * q_targ[i, next_best_action]
        if (self.with_per):
            # Update PER Sum Tree
            self.buffer.update(idx[i], abs(old_q - q[i, a[i]]))
    # Train on batch
    self.agent.fit(s, q)
    # Decay epsilon
    self.epsilon *= self.epsilon_decay
```

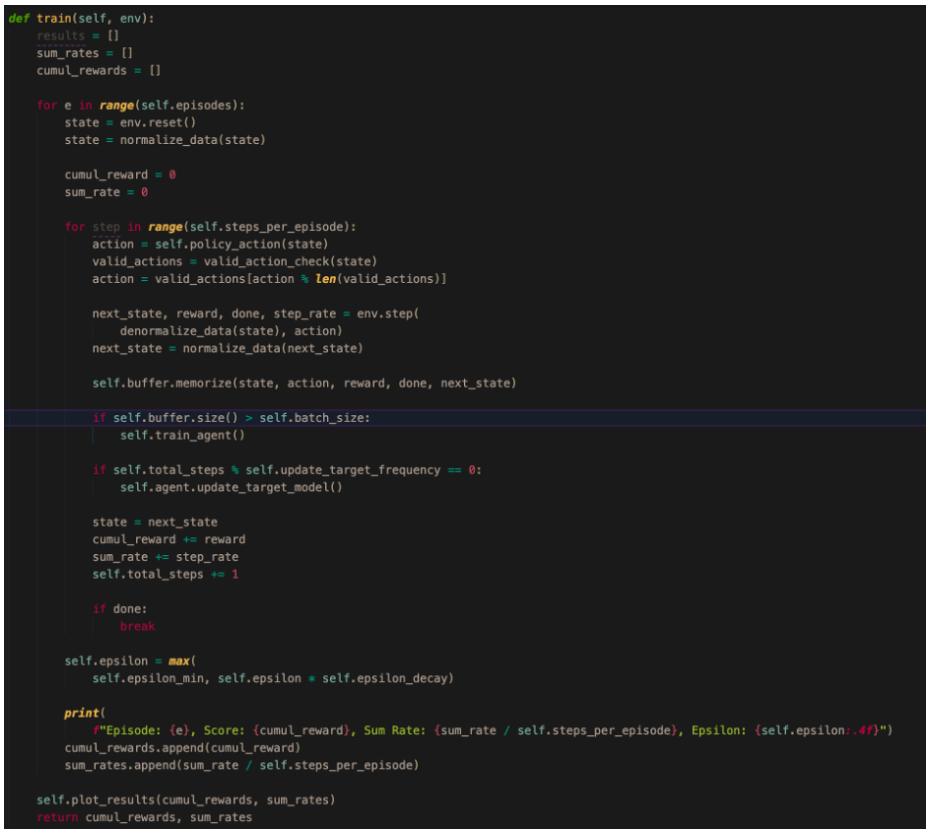
Figure 4.3: DQN and DDQN Q value calculation

update the policy in a coherent manner. These trajectories provide the context needed for calculating the rewards-to-go and the advantages, which are essential for the temporal credit assignment in complex environments where actions have long-term consequences. This structured approach to memory usage helps align the learning process directly with the current policy's performance, promoting more stable and consistent policy improvement.

Step 3: Build Actor and Critic Network

The Actor Network determines the actions to take given the current state. Additionally, we implemented it using an optimizer Adam, known for its effectiveness across various scenarios by adjusting the learning rate dynamically for each weight. This network outputs a probability distribution over possible actions.

Actor Network Construction: As illustrated in Figure 4.5, the architecture



```
def train(self, env):
    results = []
    sum_rates = []
    cumul_rewards = []

    for e in range(self.episodes):
        state = env.reset()
        state = normalize_data(state)

        cumul_reward = 0
        sum_rate = 0

        for step in range(self.steps_per_episode):
            action = self.policy_action(state)
            valid_actions = valid_action_check(state)
            action = valid_actions[action % len(valid_actions)]

            next_state, reward, done, step_rate = env.step(
                denormalize_data(state), action)
            next_state = normalize_data(next_state)

            self.buffer.memorize(state, action, reward, done, next_state)

            if self.buffer.size() > self.batch_size:
                self.train_agent()

            if self.total_steps % self.update_target_frequency == 0:
                self.agent.update_target_model()

            state = next_state
            cumul_reward += reward
            sum_rate += step_rate
            self.total_steps += 1

            if done:
                break

        self.epsilon = max(
            self.epsilon_min, self.epsilon * self.epsilon_decay)

        print(
            f"Episode: {e}, Score: {cumul_reward}, Sum Rate: {sum_rate / self.steps_per_episode}, Epsilon: {self.epsilon:.4f}")
        cumul_rewards.append(cumul_reward)
        sum_rates.append(sum_rate / self.steps_per_episode)

    self.plot_results(cumul_rewards, sum_rates)
    return cumul_rewards, sum_rates
```

Figure 4.4: DQN and DDQN training process

begins with the construction of an input layer. This input layer serves as the foundation for building two subsequent hidden layers, with the final output size corresponding to the number of possible actions. The outputs are logits for each action. Logits, representing the natural logarithm of the probabilities output by the actor network, enhance numerical stability by transforming the multiplication of probabilities into a summation of log probabilities. This transformation simplifies computations and increases stability.

Critic Network Architecture The critic network shares the same input structure as the actor, processing inputs through a similarly constructed hidden layer configuration. However, it diverges in its final layer, which consists of a single output unit designed to provide a scalar estimate of the state's value. This setup enables the critic to assess the potential long-term rewards from each state, guiding the training process of the actor network.

Optimizers

```
# Input layer
observation_input = keras.Input(
    shape=(observation_dimensions,), dtype="float32")

# Actor Network
logits = mlp(observation_input, list(hidden_sizes) + [num_actions])
actor = keras.Model(inputs=observation_input, outputs=logits)

# Critic Network
value = tf.squeeze(
    mlp(observation_input, list(hidden_sizes) + [1]), axis=1)
critic = keras.Model(inputs=observation_input, outputs=value)

# Initialize the policy and the value function optimizers
policy_optimizer = keras.optimizers.Adam(learning_rate=policy_learning_rate)
value_optimizer = keras.optimizers.Adam(
    learning_rate=value_function_learning_rate)
```

Figure 4.5: Actor-Critic network construction

- **Policy Optimizer:** Utilizes the Adam optimizer with a specified policy learning rate, facilitating efficient learning updates for the actor network.
- **Value Function Optimizer:** Also employs the Adam optimizer, but tailored with a value function learning rate, ensuring the critic network effectively converges to accurate value estimations.

Step 4: Train Network and Update The PPO algorithm employs a training procedure that includes both the actor and critic networks. This method is structured to refine the agent's strategy by utilizing the advantages calculated based on the difference between the estimated value functions and the actual returns.

- Terminal State Evaluation: At the end of each episode or batch of interactions, check if the final state is a terminal state. If it is, the expected future reward is zero. If not, use the critic network to estimate the future reward from the current state. This evaluation influences the computation of the returns for each state-action pair in the trajectory.
- Reward Backpropagation: Process the rewards stored in the buffer in reverse order to compute the discounted returns efficiently. This backpropagation adjusts each reward with the knowledge of subsequent rewards,

thereby calculating the total expected return from each state onwards till the end of the episode.

– Critic Network Training: Train the critic using the states and their corresponding computed discounted returns shown in Figure 4.6. The objective is to minimize the mean squared error between the predicted value from the critic network and the actual computed returns. This step refines the critic's accuracy in valuing different states under the current policy.

```
@tf.function
def train_value_function(observation_buffer, return_buffer):
    with tf.GradientTape() as tape:  # Record operations for automatic differentiation.
        value_loss = tf.reduce_mean(
            (return_buffer - critic(observation_buffer)) ** 2)
    value_grads = tape.gradient(value_loss, critic.trainable_variables)
    value_optimizer.apply_gradients(
        zip(value_grads, critic.trainable_variables))
    return value_loss
```

Figure 4.6: PPO critic network value function training

– Actor Network Adjustment: Update the actor network weights to optimize the policy. This involves maximizing the expected return by adjusting the policy towards actions that have historically led to higher advantages. The advantage function, computed as the difference between the discounted rewards and the value estimates from the critic, guides this update. Actions leading to rewards higher than what the critic predicted are encouraged. The frequency of training the policy network often differs from that of the value function.This is because each update to the policy network can significantly impact the behavior of the agent, and frequent, drastic changes can lead to instability in learning. But more frequent updates in value function can help it quickly adapt to changes in the environment dynamics or in the policy itself.

```
# Update the policy and implement early stopping using KL divergence
for _ in range(train_policy_iterations):
    kl, policy_loss = train_policy(

        observation_buffer, action_buffer, logprobability_buffer, advantage_buffer
    )

    if kl > 1.5 * target_kl:
        # Early Stopping
        break
epo_policy_loss.append(policy_loss)

# Update the value function
for _ in range(train_value_iterations):
    value_loss = train_value_function(
        observation_buffer, return_buffer)
epo_value_loss.append(value_loss)
```

Figure 4.7: PPO actor and critic network updates

Chapter 5

Evaluation, Analysis and Comparisons

5.1 Comparison Set 1 Results: Effectiveness of Action Elimination

Without Action Elimination: Initially, the UAV often failed to reach the destination, frequently opting for less effective paths. This was observed through detailed trajectory and action probability analyses (Figures 5.1 and 5.2). The inefficacy in navigation underscored the need for a strategy to restrict non-viable choices.

With Action Elimination: Upon integrating the Action Elimination strategy, there was a noticeable improvement. The UAV could consistently reaching its destination within the time constraint and converge to a higher reward (Figure 5.3 and 5.4), marking a clear enhancement over its prior performance.

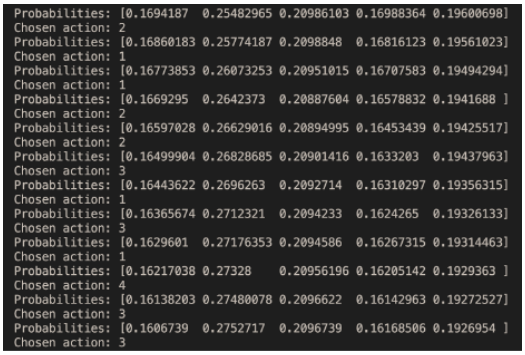


Figure 5.1: PPO agent action probabilities distribution before AE

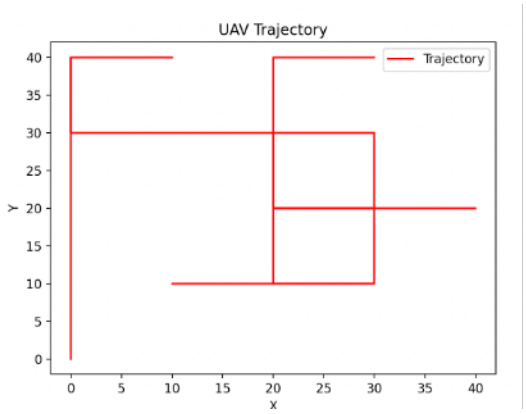


Figure 5.2: PPO agent trajectory before AE

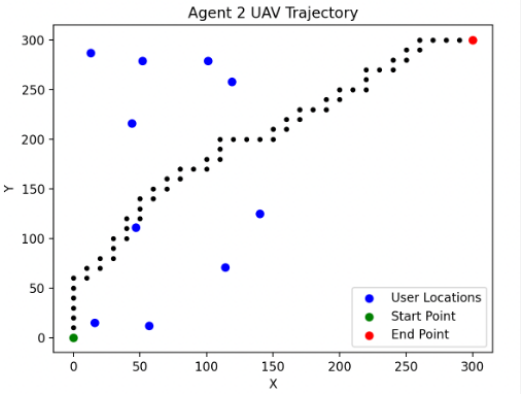


Figure 5.3: DDQN agent trajectory after AE

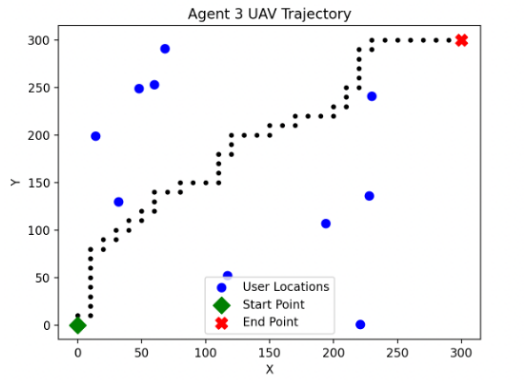


Figure 5.4: PPO agent trajectory after AE

5.2 Comparison Set 2 Results: Evaluating Reward Function Efficacy in DQN, DDQN, and PPO

This study explores the impact of two distinct reward functions—cumulative sum rate and difference of sum rate — on the performance of DQN, DDQN, and PPO algorithms. The objective is to discern how these reward structures influence the learning curves, policy stability, and convergence rates of these models.

5.2.1 Results with Cumulative Sum Rate

DQN and DDQN: Utilizing the cumulative sum rate, DQN and DDQN demonstrated an average sum rate of 8.50 and 8.51 (Figure 5.6 - 5.8), respectively, showcasing effective long-term reward accumulation. The convergence for episode rewards was observed at approximately 68, although with notable oscillations in the learning curve (Figure 5.5).

PPO: The PPO algorithm aligned well with the cumulative sum rate, achieving a stable learning trajectory shown in policy loss and value loss iteration(Figure 5.11 and 5.12), with a sum rate of 8.5 and converging episode rewards around 67 (Figure 5.9 - 5.10). This reward function supports PPO's focus on maximizing expected returns through stable policy updates.

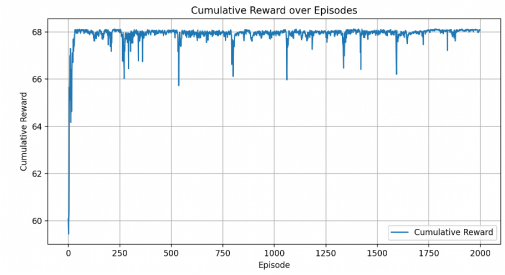


Figure 5.5: DQN agent cumulative reward over episodes with the first reward function

```
Episode: 1985, Score: 68.07581176162691, Sum Rate: 8.509476470203358
Episode: 1986, Score: 68.0704811531448, Sum Rate: 8.508810144143094
Episode: 1987, Score: 68.04452038459998, Sum Rate: 8.505565048074994
Episode: 1988, Score: 68.0306186571802, Sum Rate: 8.503827332147518
Episode: 1989, Score: 67.9736574905697, Sum Rate: 8.496707186321212
Episode: 1990, Score: 67.99120068364635, Sum Rate: 8.498900085455785
Episode: 1991, Score: 68.01255602127931, Sum Rate: 8.501569502659915
Episode: 1992, Score: 68.0343524432674, Sum Rate: 8.50429405540842
Episode: 1993, Score: 68.01278236039136, Sum Rate: 8.501597795048912
Episode: 1994, Score: 68.09278736309047, Sum Rate: 8.511598420386306
Episode: 1995, Score: 68.05846281813388, Sum Rate: 8.507307852266734
Episode: 1996, Score: 68.08691693185091, Sum Rate: 8.51086461648136
Episode: 1997, Score: 68.08206484799224, Sum Rate: 8.510258105999032
Episode: 1998, Score: 68.07620196759216, Sum Rate: 8.509525245949014
Episode: 1999, Score: 68.08585794236481, Sum Rate: 8.510732242795608
```

Figure 5.6: DQN agent output screenshot with the first reward function

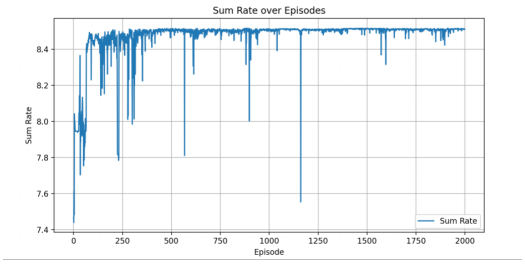


Figure 5.7: DDQN agent sum rate over episodes with the first reward function

```
Episode: 1974, Score: 68.0909047535625, Sum Rate: 8.511363094195321
Episode: 1975, Score: 68.070596474569l9, Sum Rate: 8.508824559321148
Episode: 1976, Score: 68.08863584766792, Sum Rate: 8.511079480958493
Episode: 1977, Score: 68.07781246026063, Sum Rate: 8.509726557532582
Episode: 1978, Score: 68.07435888213436, Sum Rate: 8.509294860266802
Episode: 1979, Score: 68.07236127120903, Sum Rate: 8.509045158901133
Episode: 1980, Score: 68.03787934658362, Sum Rate: 8.504734918322942
Episode: 1981, Score: 68.10813418966737, Sum Rate: 8.513516773708423
Episode: 1982, Score: 68.11074941724358, Sum Rate: 8.513843677155451
Episode: 1983, Score: 68.1036147423277, Sum Rate: 8.512951842790951
Episode: 1984, Score: 68.01458428663196, Sum Rate: 8.501823035828991
Episode: 1985, Score: 68.10067929518983, Sum Rate: 8.512584911898728
Episode: 1986, Score: 68.10012410693169, Sum Rate: 8.512515513366457
Episode: 1987, Score: 68.11177566682693, Sum Rate: 8.5139719583533 7
Episode: 1988, Score: 68.10023801304555, Sum Rate: 8.512529751630698
Episode: 1989, Score: 68.10169758620614, Sum Rate: 8.51271219827577
Episode: 1990, Score: 68.10157070799464, Sum Rate: 8.512696338499328
Episode: 1991, Score: 68.10682548470228, Sum Rate: 8.513353185587786
Episode: 1992, Score: 68.09296482853644, Sum Rate: 8.511620603567057
Episode: 1993, Score: 68.07283349348491, Sum Rate: 8.509104186685613
Episode: 1994, Score: 68.07262249415061, Sum Rate: 8.509077811768822
Episode: 1995, Score: 68.07900773339063, Sum Rate: 8.509875966673828
Episode: 1996, Score: 68.06246149096937, Sum Rate: 8.507807686371168
Episode: 1997, Score: 68.09093049407875, Sum Rate: 8.511366311759843
Episode: 1998, Score: 68.10316360608172, Sum Rate: 8.512895450760215
Episode: 1999, Score: 68.09246673849165, Sum Rate: 8.511558342311442
```

Figure 5.8: DDQN agent output screenshot with the first reward function

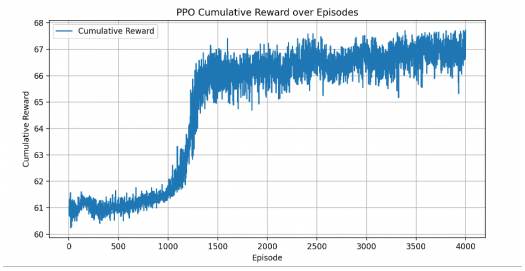


Figure 5.9: PPO agent cumulative reward over episodes with the first reward function

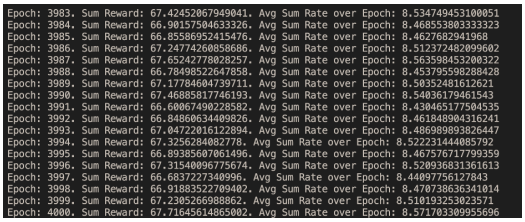


Figure 5.10: PPO agent output screenshot with the first reward function

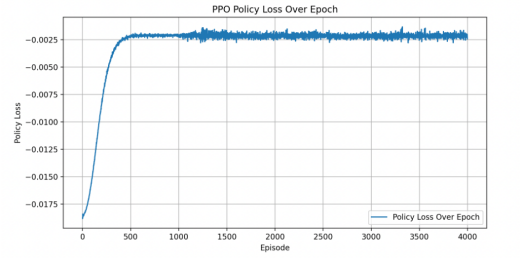


Figure 5.11: PPO agent policy loss over episodes with the first reward function

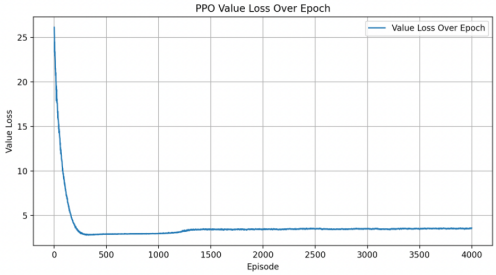


Figure 5.12: PPO agent value loss over episodes with the first reward function

5.2.2 Results with Difference of Sum Rate

DQN and DDQN: When implementing the difference of sum rate, both algorithms adjusted better to incremental reward changes, maintaining a more consistent sum rate of approximately 8.516 with less fluctuation. This setup fostered a more responsive approach to environmental dynamics, albeit with a lower episode reward.

PPO: Contrarily, PPO experienced challenges with this reward structure. The emphasis on immediate rewards led to higher variance in policy updates and a less stable learning curve, culminating in a lower sum rate of 8.2. The frequent minor adjustments introduced significant noise, complicating the policy and value loss landscapes.

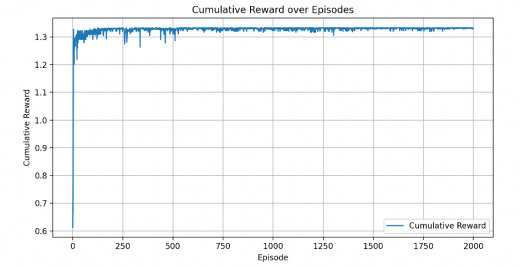


Figure 5.13: DQN agent cumulative reward over episodes with the second reward function

```
Episode: 1985, Score: 1.3325590954038313, Sum Rate: 8.51648133767891
Episode: 1986, Score: 1.3313759278903854, Sum Rate: 8.516751922162879
Episode: 1987, Score: 1.3313759278903854, Sum Rate: 8.517331383537728
Episode: 1988, Score: 1.3325590954038313, Sum Rate: 8.51698000752763
Episode: 1989, Score: 1.3308911255320064, Sum Rate: 8.516783202397402
Episode: 1990, Score: 1.3325590954038313, Sum Rate: 8.517041953306329
Episode: 1991, Score: 1.3325590954038313, Sum Rate: 8.516690623071757
Episode: 1992, Score: 1.3308911255320064, Sum Rate: 8.51697534822992
Episode: 1993, Score: 1.3325590954038313, Sum Rate: 8.51698708828598
Episode: 1994, Score: 1.3293030383829931, Sum Rate: 8.517177753567035
Episode: 1995, Score: 1.3313759278903854, Sum Rate: 8.516911981833914
Episode: 1996, Score: 1.3293030383829931, Sum Rate: 8.516443776830037
Episode: 1997, Score: 1.3293030383829931, Sum Rate: 8.515472185468894
Episode: 1998, Score: 1.329772174483268, Sum Rate: 8.516503487817916
Episode: 1999, Score: 1.3325590954038313, Sum Rate: 8.51651029607127
```

Figure 5.14: DQN agent output screenshot with the second reward function

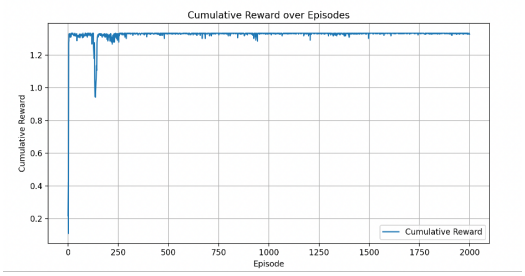


Figure 5.15: DDQN agent cumulative reward over episodes with the second reward function

```
Episode: 1981, Score: 1.329772174483268, Sum Rate: 8.51449112210134
Episode: 1982, Score: 1.3325590954038313, Sum Rate: 8.51424102341895
Episode: 1983, Score: 1.3313759278903854, Sum Rate: 8.51702709904767
Episode: 1984, Score: 1.3313759278903854, Sum Rate: 8.515617906652711
Episode: 1985, Score: 1.3325590954038313, Sum Rate: 8.513524931152494
Episode: 1986, Score: 1.3325590954038313, Sum Rate: 8.51644218883232
Episode: 1987, Score: 1.3313759278903854, Sum Rate: 8.516136243196577
Episode: 1988, Score: 1.3277512593304825, Sum Rate: 8.516426053581778
Episode: 1989, Score: 1.3325590954038313, Sum Rate: 8.516883355850743
Episode: 1990, Score: 1.3313759278903854, Sum Rate: 8.51709203212022
Episode: 1991, Score: 1.3325590954038313, Sum Rate: 8.516213120131592
Episode: 1992, Score: 1.3270128847831915, Sum Rate: 8.516588039073103
Episode: 1993, Score: 1.3313759278903854, Sum Rate: 8.516972485993346
Episode: 1994, Score: 1.3308911255320064, Sum Rate: 8.51602719247633
Episode: 1995, Score: 1.3308911255320064, Sum Rate: 8.516288353560622
Episode: 1996, Score: 1.329772174483268, Sum Rate: 8.516708335741589
Episode: 1997, Score: 1.3325590954038313, Sum Rate: 8.516411946949077
Episode: 1998, Score: 1.3325590954038313, Sum Rate: 8.516394768002117
Episode: 1999, Score: 1.3270128847831915, Sum Rate: 8.516617185905492
```

Figure 5.16: DDQN agent output screenshot with the second reward function

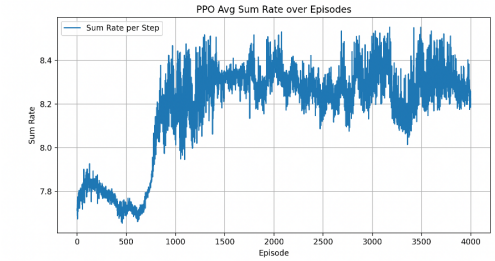


Figure 5.17: PPO agent sum rate over episodes with the second reward function

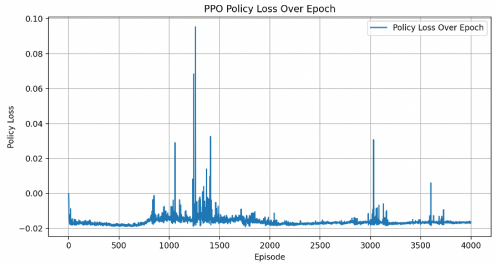


Figure 5.18: PPO agent policy loss over episodes with the second reward function

5.2.3 Discussion and Conclusions

Our experiments highlight the importance of choosing the right reward function for different types of reinforcement learning algorithms. We found that DQN and DDQN perform well with both types of reward functions we tested, showing

a slight preference depending on whether the algorithm needs to prioritize stability or responsiveness. On the other hand, PPO showed a clear preference for rewards that accumulate over time, which fits well with its focus on long-term improvements.

Analysis of DQN and DDQN: DQN and DDQN are designed to find the best action values, or Q-values, for each state-action pair. These values help the algorithm decide which action to take. The networks are trained to predict the total future rewards for each action, considering the current state.

1. **Cumulative Sum Rate Reward ($R_{\mathbf{sum}}$):** This reward structure encourages the agent to maximize the total rewards collected during an episode, aligning with the goal of DQN and DDQN to learn reliable long-term reward predictions. However, it might not always consider necessary short-term sacrifices for gaining better positions in the future.
2. **Differential Sum Rate Reward ($R_{\mathbf{diff\_ksum}}$):** This approach focuses on the changes in rewards from one state to the next, encouraging the agent to constantly seek improvements. This can lead to a more dynamic policy but might also make the agent's performance more variable, especially in unpredictable environments.

Analysis of PPO: Unlike DQN and DDQN, PPO updates its policy based on direct calculations of expected returns rather than Q-values. It adjusts the policy to favor actions that seem to offer better returns, based on calculated advantages.

1. **Cumulative Sum Rate Reward ($R_{\mathbf{ksum}}$):** Using a total sum rate as the reward gives PPO a strong and consistent signal that matches its strategy of maximizing returns over the long term. This helps PPO evaluate actions more effectively based on their potential future benefits.
2. **Differential Sum Rate Reward ($R_{\mathbf{diff\_sum}}$):** Focusing on immediate rewards from one state to the next can help make PPO's policy more responsive. However, the frequent minor changes in reward can introduce noise, making policy updates less stable and harder to manage.

Analysis of DQN Versus DDQN:

1. **Convergence Speed:** DQN can sometimes converge faster than DDQN in simpler environments where the overestimation of Q-values does not severely mislead the learning process. DDQN often converges slower than DQN because it addresses the overestimation bias by decoupling the selection of the action from the evaluation of its value. This additional step can lead to more cautious updates, thereby requiring more iterations to converge to an optimal policy.
2. **Stability:** DDQN tends to be more stable than DQN. It reduces the overestimation bias by using two networks: one to choose the best action and another to evaluate the action's value. This separation helps in providing a more accurate estimate of the Q-values, which generally leads to more stable and reliable learning outcomes. The learning updates in DDQN are less prone to the fluctuations caused by overestimated Q-values, which are common in DQN.

In conclusion, our results show that the choice of reward function significantly affects how well different reinforcement learning algorithms perform. Matching the reward function with the algorithm's design and learning objectives is crucial for achieving the best results.

5.3 Optimization of PPO Hyperparameters

PPO's performance is highly sensitive to its hyperparameter settings, such as the clipping range, learning rate, and methods for estimating advantages. Incorrect configurations can destabilize the policy, leading to erratic behavior. Conversely, while DQN and DDQN are also sensitive to their respective settings, such as learning rate and target network update frequency, they generally offer more forgiveness due to their more gradual update mechanisms. This section delves into the fine-tuning of PPO's hyperparameters to elucidate their impact on learning outcomes.

5.3.1 Training Policy Iterations

Increasing the number of policy training iterations generally enhances the sum rate but also introduces more fluctuation in learning outcomes. Starting from

a baseline configuration as detailed below, we modified the number of training policy iterations to observe its impact.

Table 5.1: Hyperparameters Before Optimization

Hyperparameter	**Value**
Steps per Epoch	80
Epochs	4000
Clip Ratio	0.2
Policy Learning Rate	3×10^{-4}
Value Function Learning Rate	1×10^{-3}
Train Policy Iterations	3
Train Value Iterations	3
Target KL	0.01
Buffer Size	80
Activation Function	tanh

Figures representing various training policy iterations are presented to illustrate the trade-offs between higher total rewards and increased fluctuations.

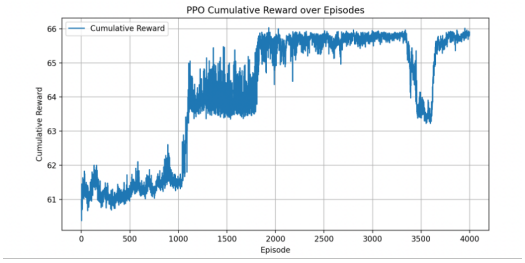


Figure 5.19: Train Policy Iterations = 1

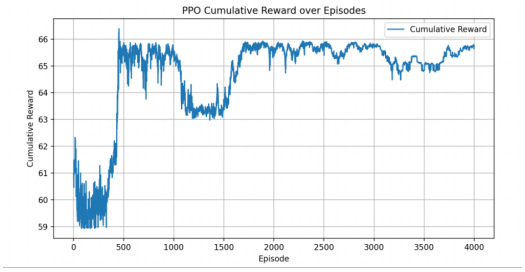


Figure 5.20: Train Policy Iterations = 3

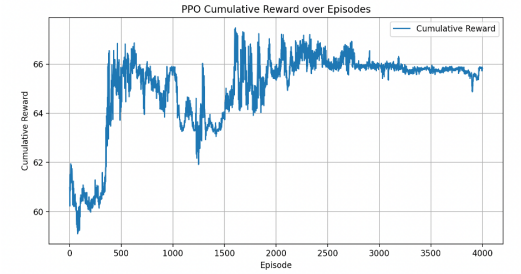


Figure 5.21: Train Policy Iterations = 5

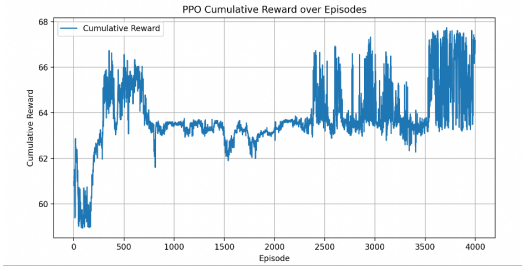


Figure 5.22: Train Policy Iterations = 10

Given the observed trends, a setting of five training policy iterations was selected for continued training due to its balance between performance improvement and stability.

5.3.2 Activation Functions

The choice of activation function significantly impacts the ability of the network to model the dynamics of the environment. While ReLU offers benefits in certain contexts, its linear behavior for positive values might oversimplify the learning model. This oversimplification can lead to suboptimal generalization across different states of the environment.

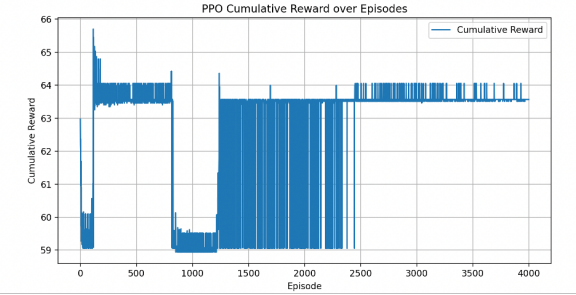


Figure 5.23: Performance Impact of Switching to ReLU Activation

Our experiments showed that tanh activation led to more consistent and generally superior performance, suggesting its better suitability for capturing the complexities involved in our specific environment.

5.3.3 Buffer Size

In our initial setup, we set the Train Policy Iterations to 10, aligned with the 80 steps per episode, and began with a buffer size designed to match this scale. As the experiments progressed, we incrementally expanded the buffer size to explore its impact on the agent's learning dynamics. Increasing the buffer size demonstrated a tangible benefit in terms of convergence speed and stability when training with consistent iterations.

A larger buffer size enriches the agent's experience by introducing a wider array of situations, thereby enhancing the diversity of the state-action pairs encoun-

tered. This diversity is crucial for comprehensive learning in reinforcement learning frameworks like PPO, which employ strategies such as experience replay. By storing a broader range of experiences, the agent gains access to a more varied set of environmental dynamics, which helps mitigate the risk of overfitting to recent patterns and reduces the correlation between consecutive training instances.

The expanded buffer ensures that each update to the model is informed by a broader context, enabling more effective generalization and smoother learning curves. This approach not only aligns with the principles of efficient data utilization in machine learning but also leverages the inherent strengths of PPO's learning mechanism.

Based on these observations, we selected a buffer size of 20,000, coupled with tanh activation and reduced the Train Policy Iterations to 5 for continued optimization. This configuration balances the need for efficient learning with the stability required to achieve consistent performance improvements.

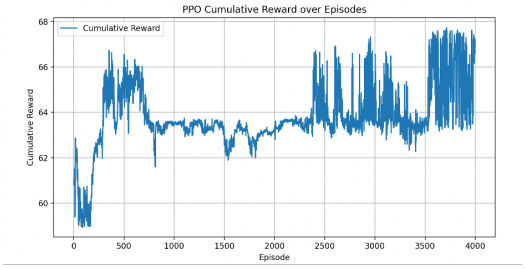


Figure 5.24: Performance with Buffer Sizes = 80

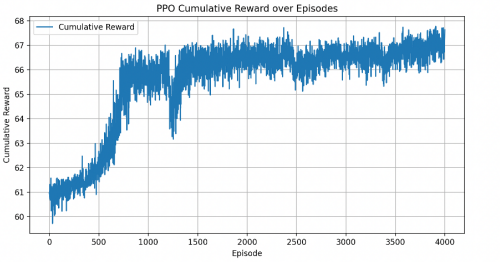


Figure 5.25: Performance with Buffer Sizes = 800

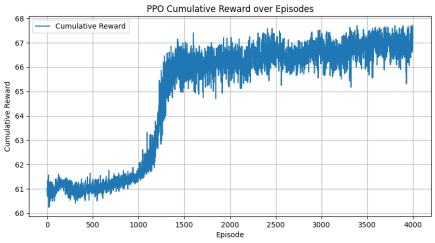


Figure 5.26: Performance with Buffer Sizes = 2000

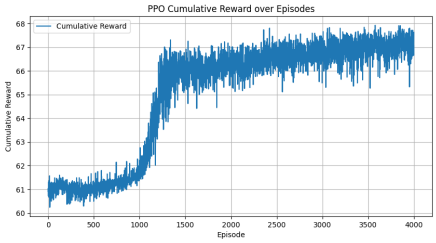


Figure 5.27: Performance with Buffer Sizes = 10000

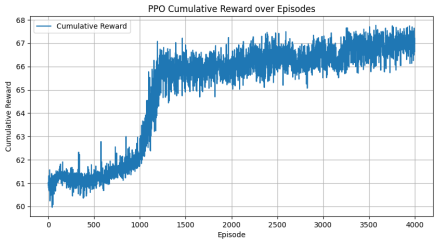


Figure 5.28: Performance with Buffer Sizes = 20000

5.3.4 Learning Rates

Selecting the right learning rates is vital for achieving effective training outcomes. Different learning rates for the policy and value networks can address their unique requirements. Adjustments in learning rates should be informed by the observed training performance and overall stability.

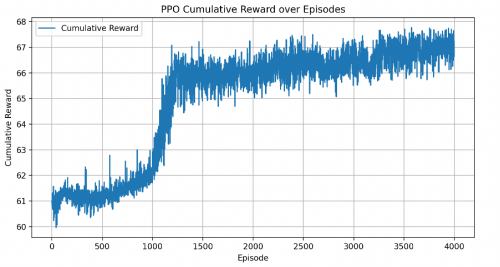


Figure 5.29: Learning rate = 3e-4

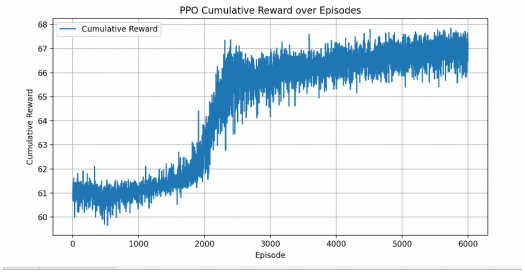


Figure 5.30: Learning rate = 1e-4

As observed, a lower learning rate stabilizes the learning curve but tends to delay convergence, while a higher learning rate accelerates convergence. The learning rate of 3e-4 begins to show convergence around epoch 1200, whereas the 1e-4 rate starts around epoch 2300. We opted for the 3e-4 rate, considering the balance between speed and stability.

5.3.5 KL Divergence Target

The KL divergence target serves as a safeguard against overly large policy updates, maintaining updates within a controlled range to prevent destabilizing the training process and deteriorating the policy.

Commonly, KL divergence targets range from 0.01 to 0.1, depending on the specifics of the environment and the sensitivity of the training process to policy modifications. Our experiments show minimal differences in learning outcomes between targets set at 0.01 and 0.1. Thus, we selected 0.1 to foster quicker learning without compromising stability.

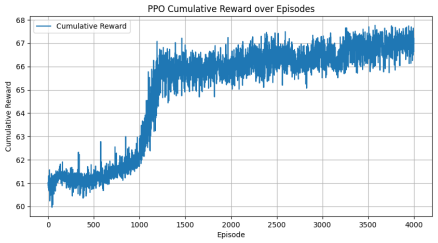


Figure 5.31: KL Divergence Target = 0.01 cumulative reward over episodes

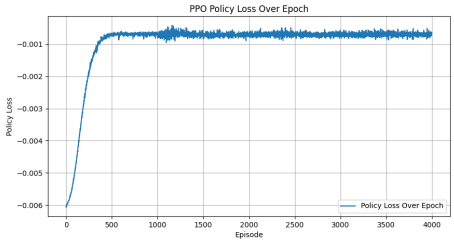


Figure 5.32: KL Divergence Target = 0.01 policy loss over episodes

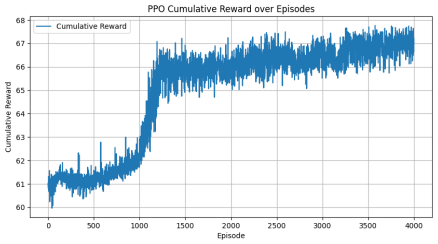


Figure 5.33: KL Divergence Target = 0.1 cumulative reward over episodes

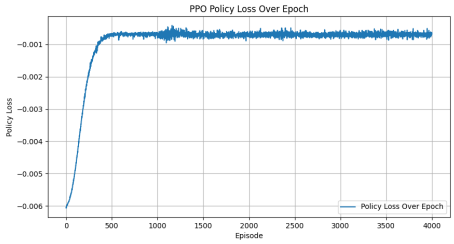


Figure 5.34: KL Divergence Target = 0.1 policy loss over episodes

5.3.6 Clip Ratio

The clip-ratio helps to control the size of the policy updates, ensuring that the changes are neither too large—to prevent destabilizing the training process—nor too small—to allow significant learning progress. The typical starting value for the clip-ratio in PPO is around 0.1 to 0.3. A common starting point is 0.2, which is often used in benchmark implementations and has been found empirically effective in many cases. If the agent's performance is improving steadily without large fluctuations in reward, the current clip-ratio is likely appropriate. If the training shows high variance in performance or sudden drops in effectiveness, consider lowering the clip-ratio to make policy updates more conservative. If the learning seems too slow, or the policy improvements have plateaued, a slight increase in the clip-ratio might be needed to allow for

A clip ratio within 0.1 - 0.3 appears to provide a balanced threshold that constrains the policy updates enough to maintain stability without stifling learning progress. The reduced noise and faster convergence suggest that the updates are being effectively regularized, promoting smoother adjustments to the policy.

A higher clip ratio such as 1.0 offers a looser constraint on policy updates, which can allow more significant changes to the policy during training. While this can accelerate learning in some scenarios, it also increases the risk of introducing volatility into the training process, as evidenced by the sporadic noise. The pulse noise observed may indicate episodes where the larger permissible updates led to substantial, yet potentially non-beneficial, shifts in policy.

The findings underscore the importance of carefully selecting the clip ratio in PPO to balance between policy update freedom and necessary constraints to ensure training stability. The optimal range of 0.1 to 0.3 for the clip ratio provides a prudent balance that enhances training effectiveness without compromising stability, making it a recommended setting for similar training environments. And we go with 0.2 in this paper.

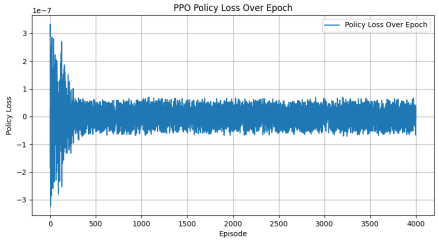


Figure 5.35: Clip ratio = 0.0 policy loss over episodes

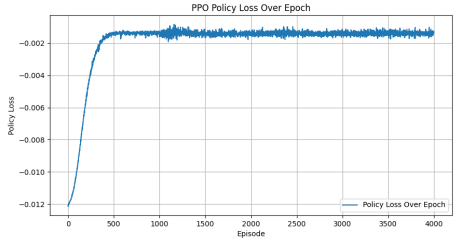


Figure 5.36: Clip ratio = 0.2 policy loss over episodes

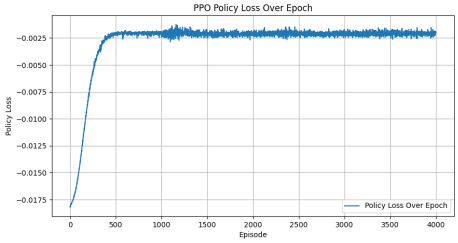


Figure 5.37: Clip ratio = 0.3 policy loss over episodes

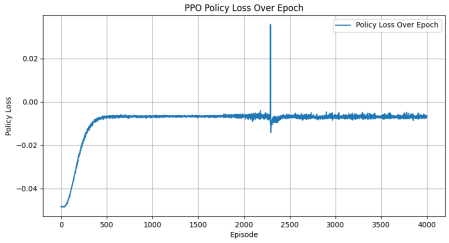


Figure 5.38: Clip ratio = 1 policy loss over episodes

5.3.7 Conclusion

Through systematic experimentation and adjustment of PPO's hyperparameters, this study has highlighted the nuanced interplay between these parameters

and the algorithm's performance. The insights gained underscore the necessity of tailored hyperparameter configurations to harness the full potential of advanced reinforcement learning algorithms like PPO. Besides, the hyperparameters after fine-tuning are shown as below table.

Table 5.2: Final Hyperparameters After Optimization

Hyperparameter	**Value**
Steps per Epoch	80
Epochs	4000
Clip Ratio	0.2
Policy Learning Rate	3×10^{-4}
Value Function Learning Rate	1×10^{-3}
Train Policy Iterations	5
Train Value Iterations	5
Target KL	0.1
Buffer Size	20000
Activation Function	tanh

Chapter 6

Conclusions

6.1 Conclusions

This chapter encapsulates the methodologies deployed, the findings uncovered, and the advantages of the algorithms proposed throughout this research. Additionally, it highlights our contributions and discusses the limitations inherent in our approaches.

6.1.1 Summary and Contributions

- We formalized the problem by maximizing the sum rate through the joint optimization of the UAV trajectory within the stipulated environment, modeling it as a MDP.
- Demonstrated the efficacy of action elimination, which significantly enhanced learning outcomes by ensuring timely destination reach, higher rewards, and avoiding suboptimal actions.
- Conducted comparative analyses of different reward functions within DQN, DDQN, and PPO frameworks. It was observed that DQN and DDQN exhibited robust performance across both reward structures, albeit with nuances in their preference towards stability or responsiveness. Conversely, PPO distinctly favored cumulative reward structures, aligning with its strategy of prioritizing sustained long-term benefits.

- Evaluated the stability and convergence speeds of DQN and DDQN under identical environmental and reward conditions, revealing that DQN might converge more swiftly in less complex settings where the overestimation of Q-values is less detrimental. Meanwhile, DDQN's updates, being less susceptible to fluctuations from overestimated Q-values, offer a steadier learning trajectory.
- Thoroughly fine-tuned hyperparameters within the PPO framework to determine their impact on learning effectiveness, providing insights into optimal settings for future reinforcement learning projects.

6.1.2 Limitations and Future Work

Despite the accomplishments, this study encounters several limitations which pave the way for future research:

- The absence of intricate environmental parameters led us to adopt a simplified model using a fixed channel gain of $\beta_0 = -60$ dB. Future studies should incorporate a more complex and realistic channel model to enhance the applicability of the findings.
- The trajectory optimization was confined to the x and y coordinates based on fixed flying height. Prospective research could integrate obstacle avoidance and adapt to changes in UAV altitude, expanding the scope of trajectory optimization.

This body of work lays foundational insights that could significantly benefit the evolution of UAV navigation systems in increasingly complex scenarios, underscoring the need for continuous advancements in algorithmic strategies and model sophistication.

Bibliography

[1] Yongs Zeng, Qingqing Wu, and Rui Zhang. Accessing from the sky: A tutorial on uav communications for 5g and beyond. *Proceedings of the IEEE*, 107(12):2327–2375, 2019. doi: 10.1109/JPROC.2019.2952892.

[2] Yong Zeng, Jiangbin Lyu, and Rui Zhang. Cellular-connected uav: Potential, challenges, and promising technologies. *IEEE Wireless Communications*, 26(1):120–127, 2019. doi: 10.1109/MWC.2018.1800023.

[3] Nan Zhao, Weidang Lu, Min Sheng, Yunfei Chen, Jie Tang, F Richard Yu, and Kai-Kit Wong. Uav-assisted emergency networks in disasters. *IEEE Wireless Communications*, 26(1):45–51, 2019.

[4] Meng Zhang, Shu Fu, and Qilin Fan. Joint 3d deployment and power allocation for uav-bs: A deep reinforcement learning approach. *IEEE Wireless Communications Letters*, 10(10):2309–2312, 2021. doi: 10.1109/LWC.2021.3100388.

[5] Yuanwei Liu, Zhijin Qin, Yunlong Cai, Yue Gao, Geoffrey Ye Li, and Arumugam Nallanathan. Uav communications based on non-orthogonal multiple access. *IEEE Wireless Communications*, 26(1):52–57, 2019.

[6] Irshad A Meer, Mustafa Ozger, Dominic Schupke, and Cicek Cavdar. Mobility management for cellular-connected uavs: Model based versus learning based approaches for service availability. *IEEE Transactions on Network and Service Management*, 2024.

[7] Hongguang Sun, Chao Ma, Linyi Zhang, Jiahui Li, Xijun Wang, Shuqin Li, and Tony QS Quek. Coverage analysis for cellular-connected random 3d mobile uavs with directional antennas. *IEEE Wireless Communications Letters*, 12(3):550–554, 2023.

[8] MAHMOUD Almasri, Xavier Marjou, and FANNY Parzysz. Reinforcement-learning based handover optimization for cellular uavs connectivity. *WSEAS Transactions on Computer Research*, 10:93–98, 2022.

[9] Akram Al-Hourani, Sithamparanathan Kandeepan, and Simon Lardner. Optimal lap altitude for maximum coverage. *IEEE Wireless Communications Letters*, 3(6):569–572, 2014.

[10] Mohammad Mozaffari, Walid Saad, Mehdi Bennis, and Mérouane Debbah. Unmanned aerial vehicle with underlaid device-to-device communications: Performance and tradeoffs. *IEEE Transactions on Wireless Communications*, 15(6):3949–3963, 2016.

[11] Fumie Ono, Hideki Ochiai, and Ryu Miura. A wireless relay network based on unmanned aircraft system with rate optimization. *IEEE Transactions on Wireless Communications*, 15(11):7699–7708, 2016.

[12] Yang Wu, Weiwei Yang, Xinrong Guan, and Qingqing Wu. Uav-enabled relay communication under malicious jamming: Joint trajectory and transmit power optimization. *IEEE Transactions on Vehicular Technology*, 70 (8):8275–8279, 2021. doi: 10.1109/TVT.2021.3089158.

[13] Yong Zeng, Xiaoli Xu, Shi Jin, and Rui Zhang. Simultaneous navigation and radio mapping for cellular-connected uav with deep reinforcement learning. *IEEE Transactions on Wireless Communications*, 20(7):4205–4220, 2021.

[14] Walter Debus and L Axonn. Rf path loss & transmission distance calculations. *Axonn, LLC*, pages 1–5, 2006.

[15] Hyung G Myung. Introduction to single carrier fdma. In *2007 15th European signal processing conference*, pages 2144–2148. IEEE, 2007.

[16] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(4):623–656, 1948. doi: 10.1002/j.1538-7305.1948.tb00917.x.

[17] Liheng Liu. Performance evaluation of direct air-to-ground communication using new radio (5g), 2017.

[18] Manhattandistance, 2007. URL `https://reference.wolfram.com/language/ref/Manh`

[19] Liangheng Lv, Sunjie Zhang, Derui Ding, and Yongxiong Wang. Path planning via an improved dqn-based learning policy. *IEEE Access*, 7:67319–67330, 2019.

[20] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. 30(1), 2016.

[21] Haijun Zhang, Miaolin Huang, Huan Zhou, Xianmei Wang, Ning Wang, and Keping Long. Capacity maximization in ris-uav networks: a ddqn-based trajectory and phase shift optimization approach. *IEEE Transactions on Wireless Communications*, 22(4):2583–2591, 2022.

[22] Anton Zakharenkov and Ilya Makarov. Deep reinforcement learning with dqn vs. ppo in vizdoom. In *2021 IEEE 21st international symposium on computational intelligence and informatics (CINTI)*, pages 000131–000136. IEEE, 2021.