

Aprendizagem por reforço robusta baseada em visão para navegação de UAVs em parques fotovoltaicos

JOÃO FERREIRA CAMPANHÃ
novembro de 2025

POLITÉCNICO DO PORTO
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Robust Vision-Based Deep Reinforcement Learning for UAV Inspection of Floating Photovoltaic Farms

João Ferreira Campanhã

Master in Electrical and Computer Engineering
Specialization Area of Autonomous Systems

ISEP INSTITUTO SUPERIOR
DE ENGENHARIA DO PORTO

DEPARTAMENTO DE ENGENHARIA ELETROTÉCNICA
Instituto Superior de Engenharia do Porto

November, 2025

*This dissertation partially satisfies the requirements of the
Thesis/Dissertation course of the program Master in Electrical and Computer
Engineering, Specialization Area of Autonomous Systems.*

Candidate: João Ferreira Campanhã, No. 1201033, 1201033@isep.ipp.pt

Scientific Guidance: Maria Benedita Malheiro, mbm@isep.ipp.pt

Company: INESC TEC

Advisor: Andry Maykol Pinto, andry.m.pinto@inesctec.pt



DEPARTAMENTO DE ENGENHARIA ELETROTÉCNICA
Instituto Superior de Engenharia do Porto
Rua Dr. António Bernardino de Almeida, 431, 4200-072 Porto

November, 2025

"If a machine is expected to be infallible, it cannot also be intelligent."

— Alan Turing

Acknowledgements

Nesta secção, gostaria de agradecer, em primeiro lugar, a todas as pessoas que me apoiaram direta e indiretamente na realização desta tese, mesmo aquelas que não foram mencionadas aqui.

Começando especificamente pela minha orientadora científica, Maria Benedita Malheiro, por todo o apoio incrível prestado e pela disponibilidade constante para esclarecer qualquer dúvida. Gostaria também de agradecer ao meu orientador, Andry Maykol Pinto, não só pela oportunidade de trabalhar neste tema de tese, mas também por todo o apoio prestado nos últimos anos.

Obrigado ao Francisco Neves, que me apresentou ao mundo da RL e sempre se esforçou para entender as minhas teorias e explicações, por mais absurdas que parecessem, alimentando constantemente a minha curiosidade. Também preciso agradecer à equipa do MOTUS, que sempre me orientou, apoiou e demonstrou-se sempre prestável para qualquer problema que tivesse.

Aos meus colegas do curso que continuamente me acompanharam nesta complicada jornada que foi estes últimos dois anos, especialmente ao meu parceiro de grupo e amigo Marco Almeida. Não deixarei de lembrar os meus amigos de longa data que me acompanharam mais uma nova etapa da minha vida.

À minha família, que me fez ser quem sou hoje e sempre esteve ao meu lado, mesmo nos momentos mais difíceis, deixando-me sem palavras para expressar tamanha gratidão. É também impossível não mencionar a minha namorada Raquel Mendonça, que sempre tentou apoiar-me e estar ao meu lado, figurativamente e literalmente, com amor e paciência.

Abstract

This dissertation proposes and validates a robust *Reinforcement Learning* (RL) method for visual navigation of *Unmanned Aerial Vehicles* (UAVs) tasked to inspect floating photovoltaic panel arrays in the Alqueva reservoir, using simulation-based development and testing. Panel inspection requires low-altitude flights, and the dynamic nature of the floating environment renders waypoint-based planning ineffective, requiring the method to operate under varied conditions and resist visual disturbances. To address these challenges, the study compares two feature extraction architectures: a vision-based model and a multimodal data model that combines visual data with numerical inputs, including actions and velocities.

The *Soft Actor-Critic* (SAC) policy was selected to process the latent state produced by the feature extractors. Following training with *domain randomization*, results showed that the multimodal model that combines visual and action inputs outperforms other variants in accuracy, control, and task completion. However, its robustness to visual perturbations remained somewhat limited. To address this shortcoming, the *domain randomization* was refined, the model retrained with appropriate regularization, and the hyperparameters tuned, significantly improving robustness at the cost of a slight reduction in overall performance.

This work contributes with a modular simulation pipeline for training and validation, a comparative analysis between models exploring unimodal and multimodal data, and practical insights into the accuracy–robustness trade-off in *Reinforcement Learning* (RL). Domain randomization and data multimodality were fundamental to improving model performance and generalization.

Keywords: Reinforcement Learning, Visual navigation, Unmanned Aerial Vehicles (UAVs), Domain Randomization, Photovoltaic (PV) array inspection, Multimodal learning, Simulation.

Resumo

Esta dissertação propõe e valida um método robusto de *Reinforcement Learning* (RL) para navegação visual de UAVs encarregues de inspecionar painéis fotovoltaicos flutuantes na albufeira de Alqueva, utilizando desenvolvimento e testes baseados em simulação. A inspeção dos painéis requer voos a baixa altitude, e a natureza dinâmica do ambiente flutuante torna ineficaz o planeamento baseado em pontos de referência, exigindo que o método opere em condições variadas e resista a perturbações visuais. Para enfrentar estes desafios, o estudo compara duas arquiteturas de extração de características: um modelo unimodal baseado em visão e um modelo multimodal que combina dados visuais com entradas numéricas, nomeadamente ações e velocidades.

A política *Soft Actor-Critic* (SAC) foi selecionada para processar o estado latente produzido pelos extratores de características. Após o treino com *randomização de domínio*, os resultados mostraram que o modelo multimodal que combina entradas visuais e ações supera as restantes variantes em precisão, controlo e conclusão da tarefa. No entanto, a sua robustez a perturbações visuais permaneceu algo limitada. Para superar esta deficiência, a *randomização de domínio* foi melhorada, o modelo foi retreinado com regularização apropriada e os hiperparâmetros foram ajustados, melhorando significativamente a robustez à custa de uma ligeira redução no desempenho geral.

Este trabalho contribui com um *pipeline* modular de simulação para treino e validação, uma análise comparativa entre modelos que exploram dados unimodais e multimodais, e perspetivas práticas sobre o compromisso entre precisão e robustez em *Reinforcement Learning* (RL). A randomização do domínio e a multimodalidade dos dados foram fundamentais para melhorar o desempenho e a generalização do modelo.

Palavras-Chave: Aprendizagem por Reforço (RL), Navegação visual, Veículos Aéreos Não Tripulados (UAVs), *Domain Randomization*, Inspeção de painéis Fotovoltaicos (PV), Aprendizagem multimodal, Simulação.

Contents

Contents	iv
List of Figures	vii
List of Tables	xi
List of Equations	xii
Listings	xv
List of Algorithms	xvii
List of Acronyms	xix
List of Symbols	xxiii
1 Introduction	1
1.1 Dissertation Scope	3
1.2 Background and Motivation	4
1.3 Objectives	4
1.4 Dissertation Structure	5
2 Reinforcement Learning	7
2.1 Definition and History	7
2.2 Core Concepts	9
2.2.1 Model-Based and Model-Free	12
2.2.2 Value-Based Methods	13
2.2.3 Policy-Based Methods	15
2.2.4 Actor Critic Methods	17
2.2.5 On-Policy and Off-Policy	18
2.2.6 Exploration and Exploitation	20
2.3 Deep Reinforcement Learning	21
2.4 Summary	27
3 Related Work	29

3.1	PV Inspection Procedures	29
3.2	Vision-Based RL	32
3.2.1	Decoupled Techniques	32
3.2.2	End-to-End Learning	33
3.3	Robust RL	35
3.3.1	Domain Randomization	36
3.3.2	Noise Injection	37
3.3.3	Adversarial Framework	37
3.4	Summary	38
4	Design of the Solution	39
4.1	Problem Formulation	39
4.2	Training, Validation and Evaluation	42
4.3	RL Architectures	44
4.4	Summary	47
5	Development of the Solution	49
5.1	Simulation Environment	49
5.2	Gazebo Configuration	50
5.3	RL Agent Integration	54
5.4	Summary	62
6	Tests and Results	65
6.1	Experimental Configuration	65
6.2	Model Training and Comparison	67
6.2.1	Overall Data Analysis	67
6.2.2	Remarks	72
6.3	Best Model Evaluation	73
6.3.1	Default Test	74
6.3.2	Aligned Test	77
6.3.3	Altitude Test	79
6.3.4	Image Perturbation Test	83
6.3.5	Image Perturbation and Changing Sunlight Test	86
6.3.6	Remarks	89
6.4	RL Agent Retraining	90
6.4.1	First Retraining	90
6.4.2	Regularization	92
6.4.3	Hyperparameter Tuning	95
6.4.4	Final Model Analysis	96
6.4.5	Remarks	101
6.5	Final Evaluation	101

6.5.1	Default Test	102
6.5.2	Aligned Test	105
6.5.3	Altitude Test	108
6.5.4	Image Perturbation Test	112
6.6	Summary	115
7	Conclusions	117
7.1	Contributions	117
7.2	Findings	118
7.3	Future Work	118
	References	119
	Appendix A Appendix	129
A.1	Feature Extractor Architectures in Python/Pytorch	129
A.2	YAML Configuration Examples	133
A.3	Reward Values Computation	136
A.4	Training Loop Step and Reset Functions	139

List of Figures

1.1	The floating PV solar plant in the Alqueva Dam, the largest on a reservoir in Europe.	3
2.1	RL framework.	8
2.2	Model-free and model-based methods [10, Chapter 8].	13
2.3	Classical value-based methods [23].	14
2.4	Visual representation of the main RL methods.	17
2.5	Difference between linear and exponential decay.	21
3.1	Creation of orthoimages of a <i>Photovoltaic</i> (PV) solar plant, trough visual and thermal images [49].	30
3.2	PV panel close inspection [50].	31
3.3	Corrective and non-corrective methods based on the <i>Unmanned Aerial Vehicle</i> (UAV) <i>Field Of View</i> (FOV) during flight over PV grid [52].	32
3.4	<i>Data-Regularized Q-learning</i> (DrQ)-v2 architecture [65].	34
3.5	Dreamer-V3 architecture [70].	34
3.6	Illustration of frame stacking (and frame skipping) in an Atari game [72].	35
3.7	Domain randomization of visual inputs [78].	36
4.1	Camera and body frames of the UAV, shown with respect to the world frame.	40
4.2	Visualization of the true state errors e_d and e_ψ	41
4.3	Example of training environment for the agent.	43
4.4	Default Feature extractor architecture for the SAC algorithm.	46
4.5	Feature extractor architecture with normalized actions for the SAC algorithm.	47
5.1	Solar panel model in the online store (a) and in Blender after configuration (b).	50
5.2	Final panel model used in the simulation.	51
5.3	Different types of textures used in the simulation.	52
5.4	Different texture layers used in the simulation.	53

5.5	Final appearance of the simulation world with sandy background. . .	54
5.6	<i>Reinforcement Learning</i> (RL) training configuration and orchestration. .	55
5.7	Implementation of the domain randomization at each reset step. . .	57
5.8	UAV preparation before the learning process.	59
5.9	Image perturbation example using OpenCV and Albumentations function.	61
5.10	High-level overview of the training rollout pipeline.	62
6.1	Average position metrics in (a) and yaw in (b) of the three models. .	68
6.2	Smoothed episode length in (a) and smoothed episode reward in (b). .	69
6.3	Evolution of the actor loss, critic loss, entropy loss, and entropy coefficient during training, shown in (a), (b), (c), and (d), respectively, for each model training.	71
6.4	Reward obtained in each evaluation phase over the training.	72
6.5	Rocky background <i>Physically Based Rendering</i> (PBR) texture used in all evaluation phases.	73
6.6	Overall reach-end behavior (a) and trajectory example (b) in the default test.	74
6.7	Reach-end behavior for each evaluated background in the default test. .	75
6.8	Reach-end behavior for each individual array in the default test. . .	76
6.9	Overall reach-end behavior (a) and trajectory example (b) in the aligned test.	77
6.10	Reach-end behavior for each evaluated background in the aligned test. .	78
6.11	Reach-end behavior for each individual array in the aligned test. . .	79
6.12	Overall reach-end behavior (a) and trajectory example (b) in the altitude test.	80
6.13	Reach-end behavior for each evaluated background in the altitude test. .	81
6.14	Reach-end behavior for each individual array in the altitude test. . .	82
6.15	Reach-end behavior at 2.5 m, 2.7 m and 2.9 m in the altitude test. . .	82
6.16	Overall reach-end behavior (a) and trajectory example (b) in the image perturbation test.	84
6.17	Reach-end behavior for each evaluated background in the image perturbation test.	85
6.18	Reach-end behavior for each individual array in the image perturbation test.	86
6.19	Overall reach-end behavior (a) and trajectory example (b) in the image perturbation and changing sunlight test.	87
6.20	Reach-end behavior for each evaluated background in the image perturbation and changing sunlight test.	88

6.21	Reach-end behavior for each individual array in the image perturbation and changing sunlight test.	88
6.22	Average position metrics (a) and average yaw metrics (b) of Model 3 and the retrained model.	91
6.23	Smoothed episode length (a) and smoothed episode reward (b) of Model 3 and the retrained model.	92
6.24	Average position (a) and average yaw metrics (b) for the retraining of the first and second (with regularization) retrained models.	93
6.25	Smoothed episode length (a) and smoothed episode reward (b) for the retraining of the first and second (with regularization) retrained models.	94
6.26	Average position metrics are represented in a) and average yaw metrics are represented in b), comparing the performance of the new retrained model.	97
6.27	Smoothed episode length in a) and smoothed episode reward in b) comparing the final retrained model, first retrained model, and the original Model 3 model.	98
6.28	Training evolution of actor, critic, and entropy losses and the entropy coefficient, shown in (a)–(d), for the final retrained, first retrained, and original Model 3.	100
6.29	Reward obtained in each evaluation phase over the training.	101
6.30	Overall reach-end behavior (a) and trajectory example (b) in the default test with the retrained model.	102
6.31	Reach-end behavior for each evaluated background in the default test with the retrained model.	103
6.32	Reach-end behavior for each individual array in the default test with the retrained model.	104
6.33	Overall reach-end behavior (a) and trajectory example (b) in the aligned test with the retrained model.	105
6.34	Reach-end behavior for each evaluated background in the aligned test with the retrained model.	106
6.35	Reach-end behavior for each individual array in the aligned test with the retrained model.	107
6.36	Overall reach-end behavior (a) and trajectory example (b) in the altitude test with the retrained model.	108
6.37	Reach-end behavior for each evaluated background in the altitude test with the retrained model.	109
6.38	Reach-end behavior for each individual array in the altitude test with the retrained model.	110

6.39	Reach-end behavior for each specific altitude (2.5 m, 2.7 m and 2.9 m) in the altitude test with the retrained model.	111
6.40	Overall reach-end behavior (a) and trajectory example (b) in the per- turbation test with the retrained model.	113
6.41	Reach-end behavior for each evaluated background in the perturba- tion test with the retrained model.	113
6.42	Reach-end behavior for each individual array in the perturbation test with the retrained model.	114
A.1	RQT graph representing the nodes and the topics, focusing on the RL interface node.	142

List of Tables

1.1	Top 10 countries for annual and cumulative installed capacity in 2024 [1].	2
2.1	Comparison of Reinforcement Learning Algorithms	27
6.1	Hardware and software specification.	66
6.2	Default SAC hyperparameters used.	66
6.3	Average Yaw and Position Errors with Standard Deviations for all the planes and arrays in the default test.	76
6.4	Average Yaw and Position Errors with Standard Deviations for all the planes and arrays in the aligned test.	78
6.5	Average Yaw and Position Errors with Standard Deviations for all planes, arrays and altitudes, in the altitude test.	83
6.6	Average Yaw and Position Errors with Standard Deviations for all the planes and arrays in the image perturbation test.	85
6.7	Average Yaw and Position Errors with Standard Deviations for all the planes and arrays in the image perturbation and changing sunlight test.	89
6.8	Final SAC hyperparameters used in the retraining.	96
6.9	Average Yaw and Position Errors with Standard Deviations for all the planes and arrays in the default test with the retrained model.	104
6.10	Average Yaw and Position Errors with Standard Deviations for all the planes and arrays in the aligned test with the retrained model.	107
6.11	Average Yaw and Position Errors with Standard Deviations for all the planes, arrays and altitudes in the altitude test with the retrained model.	112
6.12	Average Yaw and Position Errors with Standard Deviations for all the planes and arrays in the perturbation test with the retrained model.	115

List of Equations

2.1	Markov property and Transition Probability Function.	9
2.2	Belief equation in a POMDP.	9
2.3	Reward function definition.	10
2.4	Cumulative return reward definition.	10
2.5	Deterministic policy.	10
2.6	Stochastic policy.	10
2.7	State-value function.	11
2.8	Action state-value function.	11
2.9	Advantage function.	11
2.10	Bellman expectation equation for state-value functions.	12
2.11	Bellman expectation equation for action state-value functions.	12
2.12	Bellman optimality equation for state-value functions.	12
2.13	Bellman optimality equation for action state-value functions.	12
2.14	Monte Carlo update equation.	14
2.15	<i>Temporal-Difference</i> (TD) learning update equation.	14
2.16	Eligibility traces definition.	15
2.17	TD(λ) update equation.	15
2.18	Stochastic policy parameterized by θ	15
2.19	Policy Gradient cost function.	16
2.20	Policy Gradient Theorem.	16
2.21	Policy Gradient theta update.	16
2.22	Update rule of the REINFORCE algorithm.	17
2.23	Update of the SARSA algorithm.	18
2.24	Importance weighted return.	19
2.25	Update of the Q-Learning algorithm.	19
2.26	ϵ policy.	20
2.27	Exponential ϵ decay.	21
2.28	Linear ϵ decay.	21
2.29	Update of the DQN algorithm.	22
2.30	Soft target update, using Polyak smoothing.	23
2.31	Loss function of the DDPG algorithm.	23
2.32	Target-critic update of the DDPG algorithm.	23

2.33	DDPG policy gradient.	23
2.34	Target-critic update of the TD3 algorithm.	24
2.35	Loss functions of the TD3 algorithm.	24
2.37	Loss function and KDL constraint of the TRPO algorithm.	25
2.38	Loss function of the PPO-Clip algorithm.	26
2.39	Entropy definition in the RL problem.	26
2.40	Target critic update of the SAC algorithm.	27
2.41	Policy loss of the SAC algorithm.	27
3.1	Minmax formulation of the RL problem.	37
4.1	Reward computation for the position and orientation errors.	41
4.2	Reward computation for the smoothness component.	42
4.3	Designed overall reward function.	42
5.1	Real-world camera intrinsic parameters.	53

Listings

A.1	Default CNN feature extractor.	129
A.2	Costum feature extractor with stacked actions/velocities.	130
A.3	Costum feature extractor with stacked actions/velocities and improved regularization.	131
A.4	YAML configuration for the final SAC training.	133
A.5	YAML file example used for loading the Gazebo simulation.	135
A.6	Python functions to compute yaw and position errors.	136
A.7	Compute reward based on the GT errors.	138
A.8	Simplified implementation of the step and reset function within the Gymnasium environment.	139

List of Algorithms

2.1	Simple Actor-Critic Algorithm	18
-----	---	----

List of Acronyms

AI	<i>Artificial Intelligence</i>
BVLOS	<i>Beyond Visual Line-Of-Sight</i>
CNN	<i>Convolutional Neural Network</i>
CURL	<i>Contrastive Unsupervised Representations for Reinforcement Learning</i>
DAE	<i>Digital Asset Exchange</i>
DDPG	<i>Deep Deterministic Policy Gradient</i>
DMC	<i>DeepMind Control</i>
DNN	<i>Deep Neural Network</i>
DP	<i>Dynamic Programming</i>
DQN	<i>Deep Q-Network</i>
DrQ	<i>Data-Regularized Q-learning</i>
EKF	<i>Extended Kalman Filter</i>
EL	<i>Electroluminescence</i>
EU	<i>European Union</i>
FCU	<i>Flight Control Unit</i>
FIFO	<i>First In, First Out</i>
FOV	<i>Field Of View</i>
FPS	<i>Frames per second</i>
GAE	<i>Generalized Advantage Estimation</i>
GNSS	<i>Global Navigation Satellite Systems</i>
GPS	<i>Global Positioning System</i>

GT	<i>Ground Truth</i>
ICM	<i>Intrinsic Curiosity Module</i>
IL	<i>Imitation Learning</i>
IMU	<i>Inertial Measurement Unit</i>
INESC TEC	<i>Institute for Systems and Computer Engineering, Technology and Science</i>
IR	<i>Infrared</i>
IRL	<i>Inverse Reinforcement Learning</i>
ISEP	<i>Instituto Superior de Engenharia do Porto</i>
JSON	<i>JavaScript Object Notation</i>
KL	<i>Kullback–Leibler</i>
LiDAR	<i>Light Detection and Ranging</i>
MBSE	<i>Mean Squared Bellman Error</i>
MC	<i>Monte Carlo</i>
MDP	<i>Markov Decision Process</i>
ML	<i>Machine Learning</i>
MLP	<i>Multi-Layer Perceptron</i>
NN	<i>Neural Network</i>
O&M	<i>Operation and Maintenance</i>
PBR	<i>Physically Based Rendering</i>
PDF	<i>Probability Density Function</i>
PID	<i>Proportional Integral Derivative</i>
POMDP	<i>Partially Observable Markov Decision Process</i>
PPO	<i>Proximal Policy Optimization</i>
PV	<i>Photovoltaic</i>
RAM	<i>Random-Access Memory</i>
RL	<i>Reinforcement Learning</i>

ROI	<i>Region of Interest</i>
ROS	<i>Robot Operating System</i>
RTF	<i>Real-Time Factor</i>
SAC	<i>Soft Actor-Critic</i>
SARSA	<i>State-Action-Reward-State-Action</i>
SB3	<i>Stable-Baselines3</i>
SDF	<i>Simulation Description Format</i>
SITL	<i>Software in the Loop</i>
SOTA	<i>State-of-the-art</i>
TALOS	<i>roboTics and Artificial intelligence Living labs improving Operations in PV Scenarios</i>
TD	<i>Temporal-Difference</i>
TD3	<i>Twin Delayed DDPG</i>
TEDI	<i>Thesis/Dissertation</i>
TRPO	<i>Trust Region Policy Optimization</i>
UAV	<i>Unmanned Aerial Vehicle</i>
USD	<i>Universal Scene Description</i>
VLOS	<i>Visual Line-Of-Sight</i>
YAML	<i>YAML Ain't Markup Language</i>
YOLO	<i>You Only Look Once</i>

List of Symbols

Symbol	Description	Units
$A()$	Advantage function	-
A	Random action variable	-
$\mathcal{A}$	Set of actions	-
a	Single action	-
α	Step size or Temperature	-
ω	Angular velocity	rad s^{-1}
$\mathcal{F}_b$	Body frame	-
$\mathcal{F}_c$	Camera frame	-
δ_t	TD error	-
γ	Discount factor	-
e_d	Distance state error	m
E	Eligibility traces	-
G_t	Accumulative reward	-
γ	Discount factor	-
G	Gradient steps at learning	-
lr	Learning rate	-
N_{starts}	Learning starting value	-
v	Linear velocity	m s^{-1}
N	Mini-batch size	-
O	Random observation variable	-
$\mathcal{O}$	Set of observations	-
$P()$	Transition probability function	-
$Z()$	Observation function	-
o	An observation	-
$\pi()$	Policy function	-
$Q()$	Action-value function	-
R	Random reward variable	-
$R()$	Reward function	-
r	Reward signal / value	-

Symbol	Description	Units
B	Replay Buffer	-
S	Random state variable	-
$\mathcal{S}$	Set of states	-
s	Single state	-
H_{target}	Target entropy	-
T_{target}	Target update interval	-
τ	Soft update coefficient	-
θ	Policy function parameters	-
f_{train}	Training steps frequency	-
$V()$	Value function	-
$\mathcal{F}_w$	World frame	-
e_ψ	Angular/yaw state error	$^\circ$

Chapter 1

Introduction

Photovoltaic (PV) power is growing rapidly worldwide, and large-scale PV farms are becoming a central element of the decarbonized electricity society. Global cumulative PV capacity has surpassed the 2 TW scale for the first time [1], and it is expected to continue growing strongly in the coming years, with the *European Union* (EU) being the second-largest global pioneer, as can be observed in the Table 1.1.

At utility scale, *Operation and Maintenance* (O&M), it is essential to maintain energy yield and long-term asset value. Traditional manual inspections are time-consuming, labor-intensive, and can require shutdowns, creating a bottleneck as PV capacity and plant sizes continue to increase. Aerial inspection using UAVs has emerged as a practical alternative: industry reports and case studies show that UAV-based thermal and visual surveys can cover large areas orders of magnitude faster than ground-based manual inspections and at significantly lower ongoing cost [2].

More precisely, unmanned drone teams are reported to inspect on the order of ~ 20 MW per UAV operator each day for thermal spot checks at utility scale [2]. Thermal drone surveys have been field tested to scan ~ 1 MW in approximately 10 minutes for some deployments [3]. Case studies report efficiency gains of up to $\sim 97\%$ compared with manual methods [3]. Typical operational savings are reported between 1000 USD/MW to 1700 USD/MW inspected, depending on the method and site [3].

Table 1.1: Top 10 countries for annual and cumulative installed capacity in 2024 [1].

ANNUAL INSTALLED CAPACITY			CUMULATIVE CAPACITY		
	China	357.3 GW		China	1048.5 GW
	EU	62.6 GW		EU	339.4 GW
	USA	47.1 GW		USA	224.1 GW
	India	31.9 GW		India	124.6 GW
	Pakistan	17.0 GW		Germany	99.8 GW
	Germany	16.7 GW		Japan	96.9 GW
	Brazil	14.3 GW		Brazil	52.1 GW
	Spain	7.5 GW		Spain	47.2 GW
	Italy	6.6 GW		Australia	38.6 GW
	France	5.9 GW		Italy	37.0 GW
	Japan	5.5 GW		South Korea	31.7 GW

Although UAV-based inspections substantially reduce time and operational cost compared to manual on-foot surveys, relying on human pilots for routine inspections is not the most optimal to scale [4]. Regulation in Europe currently requires *Visual Line-Of-Sight* (VLOS) for most operations, unless specific *Beyond Visual Line-Of-Sight* (BVLOS) approvals are granted [5], which restricts the continuous large-scale deployment of piloted missions.

In terms of covering large areas, the use of mission planning software has been a key component in scaling these operations [6]. However, planning is conducted mainly at a higher altitude, and it depends on correct *Global Navigation Satellite Systems* (GNSS) inputs to succeed.

This approach can be effective in certain environments. However, exceptional cases such as the Alqueva dam, which is displayed in the Figure 1.1, that hosts the largest floating solar plant in Europe on a reservoir, pose challenges for precise GNSS way pointing due to small oscillations and drift of the PV plant. Using traditional mission planning software, these areas can only be inspected if the UAV operates at a higher altitude to cover the entire site, which in turn results in images with reduced detail.

Even though the UAV flies at low altitude to capture these images in great detail, allowing for more effective fault detection, its FOV must be aligned with the panels in question to ensure better data quality. One way to achieve this is through autonomous navigation using a visual camera.



Figure 1.1: The floating PV solar plant in the Alqueva Dam, the largest on a reservoir in Europe.

1.1 Dissertation Scope

This dissertation was developed in partial fulfillment of the requirements for the Master’s degree in Electrical and Computer Engineering, specialization in Autonomous Systems, at *Instituto Superior de Engenharia do Porto* (ISEP), and corresponds to the *Thesis/Dissertation* (TEDI) subject.

This work was developed in collaboration with *Institute for Systems and Computer Engineering, Technology and Science* (INESC TEC), specifically the MOTUS Robotics¹ research group, as part of the EU-funded *roboTics and Artificial intelligence Living labs improving Operations in PV Scenarios* (TALOS) project (Grant Agreement No. 101119744, HORIZON Innovation Actions). The TALOS project aims to develop innovative autonomous robotic solutions to improve the safety, efficiency and sustainability of land-based, agricultural and floating PV installations.

This dissertation addresses two main challenges: perception and control. The proposed solution must be capable of interpreting the environment, primarily through visual information, and subsequently using this information to control the UAV in the most advantageous way, ensuring proper alignment with the PV panel under inspection.

It is important to note that this dissertation does not focus on the detection or classification of PV solar panels, but rather on what is essential to ensure robust autonomous data collection. Here, *robust* refers to the ability to perform the task reliably in different environments or in the presence of changes/perturbations in the visual input.

The scope of this dissertation is limited to the development and validation of a Reinforcement Learning (RL)-based solution, with the objective of evaluating its

¹<https://motusrobotics.eu/>

suitability for autonomous inspection tasks. The floating PV plant at the Alqueva reservoir is considered the primary case study for evaluation. Performance is assessed exclusively in terms of perception and control within a single PV array. Higher-level aspects of mission planning, such as deciding when and where the RL agent should be deployed, are explicitly excluded from the scope of this work.

1.2 Background and Motivation

While previous sections have highlighted the rapid growth of PV installations and the associated inspection challenges, an additional gap remains in the area of autonomous navigation. Current UAV-based approaches rely heavily on predefined waypoints and high-level mission planning, which can become unreliable in dynamic environments such as floating PV farms. This highlights the need for more adaptive solutions that robustly integrate visual perception and control, an area in which RL agents can be particularly well suited.

The MOTUS Robotics group already had extensive experience with *Artificial Intelligence* (AI) and O&M solutions [7, 8, 9], particularly in the use of UAVs for inspection tasks. From a personal perspective, from bachelor’s studies to the completion of this dissertation, I have been deeply curious about how *Machine Learning* (ML) systems can be applied to solve demanding real-world problems. The opportunity to work on a dissertation within the scope of an EU-funded project, in collaboration with MOTUS Robotics, provided the ideal setting to channel this curiosity into practical research. It allowed me to contribute to an emerging field while also developing as both a researcher and an engineer.

1.3 Objectives

More specifically, the main objective is to develop and evaluate a model-free RL agent capable of navigating over PV solar panel arrays, primarily through visual observations, and controlling the UAV using high-level velocity actions. This solution will be developed and tested in a simulation-based environment.

In summary, the following sub-objectives are also defined:

- Ensure that the agent can be robust to environment differences and perturbations, observed by the visual input
- Ensure that it is compatible with the default tech stack used in autonomous UAV operations, such as the use of an Autopilot.
- Compare different approaches and select the best based on the results.
- Use a realistic simulation platform for the task in question.

- Develop a modular training pipeline that can be versatile and used for similar problems.
- Do an exhaustive evaluation of the agent performance for different valid conditions.

1.4 Dissertation Structure

The dissertation is organized into seven chapters that together provide the foundation for understanding and reproducing the final outcome.

Chapter 2 presents the foundational knowledge. It introduces the RL problem, the key concepts, and the main model-free RL algorithms, highlighting their specific characteristics and applications.

Chapter 3 reviews related work relevant to the problem at hand. It covers the main PV inspection procedures, techniques to achieve robust visual-based algorithms, and a discussion of *State-of-the-art* (SOTA) RL methods.

Chapter 4 describes the design of the proposed solution. The problem formulation is detailed, the steps to solve it are outlined, and the architectures and configurations employed are specified.

Chapter 5 focuses on the implementation of the solution, including the configuration of the simulation environment, the development of the training and evaluation pipelines, and the libraries used.

Chapter 6 presents and analyzes the results of training and evaluations. The performance of the model is assessed and any necessary adjustments or improvements are discussed and reimplemented if needed.

Finally, Chapter 7 provides the conclusions drawn from this work and offers suggestions for future research directions.

Chapter 2

Reinforcement Learning

This chapter addresses the core concepts, challenges and methods of *Reinforcement Learning* (RL). This review traces the evolution of RL algorithms, from early naive to modern deep approaches, highlighting their limitations and ultimately providing a theoretical foundation for this thesis.

2.1 Definition and History

As described by Sutton and Barto [10], “*Reinforcement Learning* (RL) is the process of learning what to do”. The concept behind is that any goal can be achieved by developing a mapping from situations to actions that maximizes overall rewards. This concept, intuitive to human learning, is also known as the Reward Hypothesis.

One method of achieving this reward maximization is through trial and error, a process where an agent learns via direct interaction with the environment and accumulated experiences. This interaction, distinct from Supervised and Unsupervised Learning, is visually represented in Figure 2.1. The core concepts of RL are further elaborated in Section 2.2.

From an engineering perspective, these principles are particularly captivating due to their versatility and adaptability in addressing real-world challenges. These include applications in fields such as autonomous robotics [11], self-driving vehicles, resource management, or financial systems [12].

The adopted iterative learning process aims to find optimal solutions to complex optimization problems. This characteristic, combined with its basic principles,

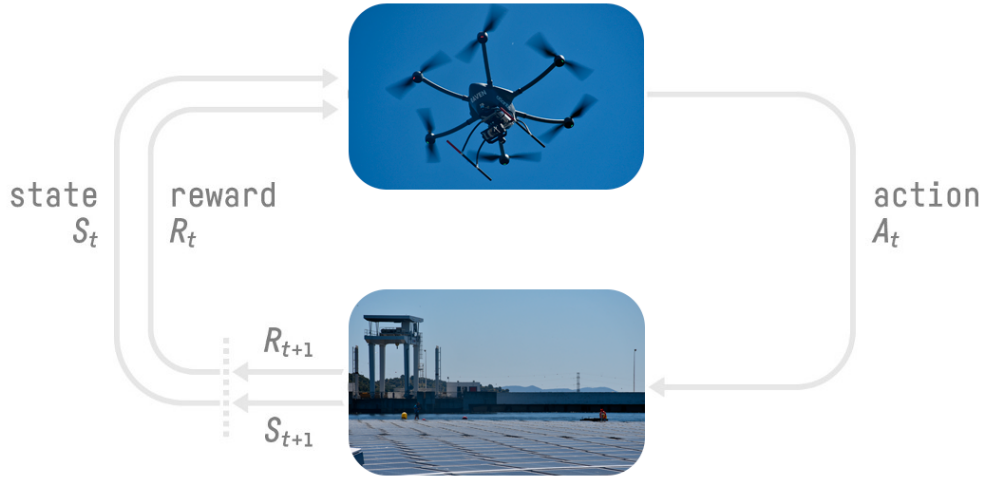


Figure 2.1: RL framework.

makes RL an interdisciplinary field that bridges machine learning, optimal control and even psychology.

The early days of RL can be traced back to the foundational ideas of psychology, control theory, and AI. Although distinct, all of these areas contributed to the evolution of RL.

In psychology, the idea of learning through trial and error was first articulated by Edward Thorndike in the early 20th century [13]. His “Law of Effect”, formulated in 1905, suggested that behaviors followed by satisfying outcomes are more likely to be repeated. Another important psychological theory was Pavlov’s classical conditioning [14], which, while not directly related to RL, influenced previous research and demonstrated how associations could be formed between stimuli and responses.

In the 1950s, control theory began to significantly influence RL, especially through the work of Richard Bellman [15], who introduced the Bellman equation. This equation became a key component of *Dynamic Programming* (DP), providing a mathematical framework to solve optimization problems in *Markov Decision Processes* (MDPs). This period marked the merging of control theory and RL, as many RL problems can be described as MDPs. Even today, the Bellman equation remains a fundamental tool in the definition and solution of RL algorithms.

Meanwhile, AI was just beginning and soon started to use RL. In 1959, Arthur Samuel created a checkers-playing program [16] that could learn from past experience by combining DP and trial-and-error. Further developments in the late 1980s, particularly by Richard Sutton, introduced TD learning [17], integrating the strengths of Monte Carlo and DP. TD learning enabled agents to learn from incomplete information and laid the foundation for many of the RL algorithms used today.

2.2 Core Concepts

The RL process can be defined as a *Markov Decision Process* (MDP) [10, Chapter 3]. This mathematical model is used to take sequential decisions in situations where the results are uncertain. It can be defined as a five-tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ where:

- States $\mathcal{S}$: A finite set of all possible states.
- Actions $\mathcal{A}$: A finite set of all possible actions.
- Transition Function $P(s' | s, a)$: The probability of moving to the next state s' after taking an action a in the actual state s . This is unknown and not explicitly used in the model-free RL).
- Reward Function $R(s, a, s')$: The expected immediate reward for transitioning from state s to next state s' after taking an action a .
- Discount Factor $\gamma \in [0, 1]$: A value that considers how much future rewards contribute to the expected return. At the extremes, with 0 only immediate rewards matter, and with 1 future rewards matter equally.

An MDP exhibits the Markov property [10, Chapter 3], which states that the distribution of the next state s' and reward depends only on the current state and action, not on the full history of past states and actions. In other words, the random variables S_{t+1} and R_{t+1} are conditionally independent of the past given (S_t, A_t) . This property is reflected in the transition probability function, defined in Equation 2.1.

$$P(s'|s, a) \doteq \mathbb{P}\{S_{t+1} = s' | S_t = s, A_t = a\} = \sum_{r \in \mathcal{R}} P(s', r | s, a). \quad (2.1)$$

In most complex problems, such as real-world robotic tasks, a MDP is not fully observable, which means that the agent does not have complete information about the state of the environment. This situation corresponds to a *Partially Observable Markov Decision Process* (POMDP) [10, Chapter 17], where the tuple is expanded to include an observation $o_t \in \mathcal{O}$ and an observation function Z that defines how the agent perceives the environment, given its true (but hidden) state. In this way, a POMDP is represented by the tuple $(\mathcal{S}, \mathcal{A}, P, R, \mathcal{O}, Z)$ [18]. To allow the agent to reason under state uncertainty, POMDP incorporates a belief state $b(s)$ that models uncertainty by assigning probabilities to all possible states. Equation 2.2 defines the belief state for the state s by $b_t(s)$, where it is a probability based on the history made of past observations $o_{1:t}$ and the past actions $a_{1:t-1}$.

$$b_t(s) = \mathbb{P}(S_t = s | o_{1:t}, a_{1:t-1}) \quad (2.2)$$

Since tracking all past observations and actions becomes infeasible, the belief is often approximated, e.g., through a *Neural Network* (NN). By using a Markovian belief state, a POMDP can be reformulated as an MDP, making it more tractable and solvable [19]. The reward function $R(s, a, s')$, which allows the agent to choose the most desirable actions, corresponds to the expected immediate reward $R_t \in \mathbb{R}$ that the agent will receive after transitioning from state $s_t = s$ to state $s_{t+1} = s'$ through action $a_t = a$ as demonstrated in Equation 2.3 [10, Chapter 3].

$$R(s, a, s') \doteq \mathbb{E}[R_t \mid S_t = s, A_t = a, S_{t+1} = s'] \quad (2.3)$$

It is also possible to assume that the reward depends solely on the current state and the action taken, in which case it can be written as $R(s, a)$. In some cases, the reward may even depend only on the action, denoted as $R(a)$.

The cumulative rewards over an episode that an agent collects over time, starting from time step t , are called the return G_t [10, Chapter 3]. This relationship is demonstrated in Equation 2.4 and it is calculated by summing all subsequent rewards $R_{t+1}, R_{t+2}, R_{t+3}, \dots$, where each reward is progressively discounted by a factor γ which prevents infinite returns in cyclic processes and emphasizes recent over distant rewards.

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (2.4)$$

These basic MDP concepts enable the definition of other RL principles, such as policy π , value function V , and optionally a model for the RL problem. A policy π specifies how the learning agent should act in a given state. Broadly speaking, a policy defines the mapping of perceived environmental states to the corresponding actions [10, Chapter 3]. The deterministic policy represented in Equation 2.5 applies to problems that conform to a basic MDP framework.

$$\pi(s) = a \quad (2.5)$$

For POMDPs, where the true state of the environment is either uncertain or affected by noise, a stochastic policy can be more appropriate. This type of policy can be defined as a probability distribution over actions given the current state [10, Chapter 3], as shown in Equation 2.6.

$$\pi(a \mid s) = \mathbb{P}(A_t = a \mid S_t = s) \quad (2.6)$$

The choice between deterministic and stochastic policies implies a trade-off. Deterministic policies can offer simplicity, safety, and computational efficiency, while stochastic policies provide robustness, exploration capabilities, and the ability to handle uncertainty in partially observable environments [20]. Therefore, selecting the appropriate policy type depends on the specific characteristics and requirements of the given problem.

For any policy π , deterministic or stochastic, two value functions are defined: a state-value function $V_\pi(s)$ and an action-value function $Q_\pi(s, a)$. A third function, the advantage function $A_\pi(s, a)$, is derived from these. The state-value function $V_\pi(s)$ evaluates how good a state is in terms of future expected rewards. As expressed by Equation 2.7, it provides immediate feedback by looking ahead to estimate the total expected return based on current state [10, Chapter 3].

$$V_\pi(s) = \mathbb{E}_\pi [G_t \mid S_t = s] \quad (2.7)$$

The action-value function $Q_\pi(s, a)$ measures the expected return when taking action a in state s while following policy π [10, Chapter 3] – see Equation 2.8. This makes the action-value function suitable for decision-making, rather than just state evaluation.

$$Q_\pi(s, a) = \mathbb{E}_\pi [G_t \mid S_t = s, A_t = a] \quad (2.8)$$

The advantage function $A_\pi(s, a)$ is obtained by subtracting the action-value function $Q_\pi(s, a)$ from the state-value function $V_\pi(s)$ [21], as expressed by Equation 2.9. This function reduces the variance by using $V_\pi(s)$ as a baseline. A positive value indicates that taking action a in state s is better than average, while a negative value suggests the opposite.

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s) \quad (2.9)$$

The Bellman Equation is used to recursively evaluate or compute the state value function $V_\pi(s)$ and the state-action value function $Q_\pi(s, a)$ [10, Chapter 3]. This equation takes advantage of the dynamic programming concept, which breaks down the problem of computing the value of a state or an action into smaller sub-problems [10, Chapter 4]. There are two classical variations of the Bellman Equation: the Bellman Expectation Equation [22], represented by Equations 2.10 and 2.11, and the Bellman Optimality Equation [22], defined through Equations 2.12 and 2.13. The Bellman Expectation, as the name suggests, computes the expected return by averaging all actions dictated by policy π , then considering transitions and rewards.

In this way, it evaluates the value of states and actions under the policy π .

$$V_\pi(s) = \mathbb{E}_\pi [R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s] \quad (2.10)$$

$$Q_\pi(s, a) = \mathbb{E}_\pi [R_{t+1} + \gamma Q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \quad (2.11)$$

The structure of a MDP ensures the existence of an optimal policy π^* , which, by definition, performs at least as well as any other policy π . This is significant because it enables the computation of the optimal value functions $V_{\pi^*}(s)$ or $Q_{\pi^*}(s, a)$, which represent the maximum expected return achievable by any policy. This can be achieved by using the Bellman Optimality Equation, which works by maximizing over actions instead of averaging, directly identifying the best possible action at each step [22].

$$V_{\pi^*}(s) = \max_a \mathbb{E} [R_{t+1} + \gamma V_{\pi^*}(S_{t+1}) \mid S_t = s, A_t = a] \quad (2.12)$$

$$Q_{\pi^*}(s, a) = \mathbb{E} \left[R_{t+1} + \gamma \max_{a'} Q_{\pi^*}(S_{t+1}, a') \mid S_t = s, A_t = a \right] \quad (2.13)$$

The Bellman Expectation Equation is linear and can be solved directly for small-scale problems. In contrast, the Bellman Optimality Equation is non-linear and, in general, does not admit a closed-form solution [22]. The optimality equation is typically used directly only in model-based RL, where the environment dynamics are assumed to be known. In model-free RL, approximations are necessary because, although the Bellman Optimality Equation does not explicitly include transition probabilities, their values are required for computation but are generally unknown.

2.2.1 Model-Based and Model-Free

As mentioned, RL algorithms can be categorized as model-based or model-free, each with its own advantages and trade-offs. The fundamental difference lies in how they handle the transition dynamics.

In model-based RL, the agent either learns or receives a transition model $P(s' \mid s, a)$ and a reward function, such as $R(s, a)$ [10, Chapter 8]. This allows planning ahead by simulating future states and rewards before taking actions, improving sample efficiency by reducing reliance on trial-and-error learning. However, inaccuracies in the learned model can lead to suboptimal policies.

In contrast, model-free RL has no explicit transition model. Instead, it learns directly from experience by updating value functions, either $Q(s, a)$ or $V(s)$, or by optimizing a policy [10, Chapter 8]. Although more flexible in complex environments where modeling transitions is impractical, model-free methods are often sample-inefficient, requiring extensive interaction with the environment to learn optimal behavior.

Figure 2.2 illustrates this distinction, highlighting how model-based approaches take advantage of the learned or given model for planning, while model-free methods rely solely on direct experience.

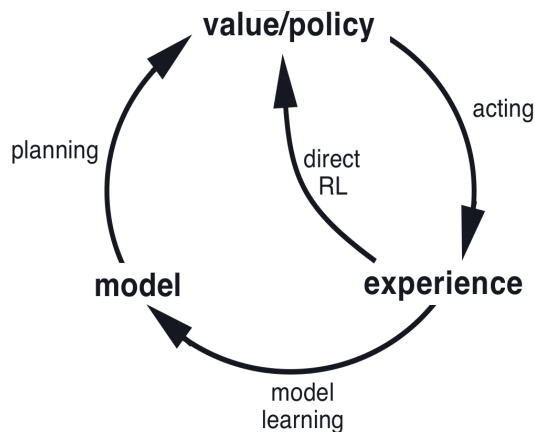


Figure 2.2: Model-free and model-based methods [10, Chapter 8].

This graphical representation provides a simplified overview, as there are numerous ways to approach the RL problem. The following sections introduce various methods, with a special focus on those applicable to model-free learning.

2.2.2 Value-Based Methods

Methods that estimate and improve $V(s)$ or $Q(s)$ value functions are value-based methods. For a well-defined MDP, these equations can be solved using: (i) tabular methods such as value iteration with *Dynamic Programming* (DP) [10, Chapter 4] in the model-based setting; and (ii) algorithms like *Monte Carlo* (MC) [10, Chapter 5] and *Temporal-Difference* (TD) learning [10, Chapter 6] in the model-free setting, where transition probabilities are unknown.

MC learns from complete episodes by observing the total reward G_t an agent receives, from the start to the end of the episode, and uses this information to update the value of states [10, Chapter 5]. Since MC does not have access to the transition probabilities, it updates the value function, using an approximation of the Bellman Expectation Equation, as demonstrated in Equation 2.14.

$$V(S) \leftarrow V(S) + \alpha(G_t - V(S)) \quad (2.14)$$

TD learning estimates value functions from experience. TD(0) updates value estimates solely based on the immediate next step, whereas TD(λ) blends updates from multiple future steps. In TD(0), the value function is updated one step at a time, meaning it continuously bootstraps¹ at each step [10, Chapter 6]. This process introduces significant bias and may not capture the true value of a state, particularly in environments where future rewards are highly variable. Equation 2.15 shows the update of the estimated value function $V(s_t)$, where R_{t+1} is the reward received after taking an action in state S_t , γ is the discount factor, $V(s_t)$ is the estimated value of the current state, and $V(s_{t+1})$ is the estimated value of the next state.

It incorporates the concept of TD error, described by δ_t , which quantifies the discrepancy between the predicted value and the observed reward plus the estimated value of the subsequent state [10, Chapter 6]. The only similarity with MC is that the updating of TD(0) value function also relies on an approximation of the Bellman Expectation Equation, as shown in Equation 2.15.

$$V(S_t) \leftarrow V(S_t) + \underbrace{\alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]}_{\delta_t} \quad (2.15)$$

Figure 2.3 shows how these different methods behave with the same state-transition tree. The tree represents the structure of possible state transitions, where each white node corresponds to a state, and the branches indicate the potential paths the agent can take from that state, along with the associated rewards, as represented as black nodes.

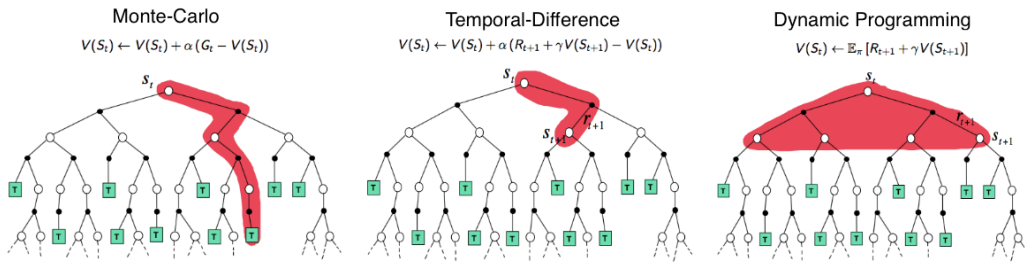


Figure 2.3: Classical value-based methods [23].

In TD(λ), the parameter λ controls how much past states are updated based on future rewards. It balances the bias-variance trade-off by combining information from one-step TD updates and full-episode returns. This differs from TD(0), which

¹In the context of RL, bootstrapping refers to updating the value of a state based on other value estimates rather than waiting for actual outcomes or returns.

only uses information from the next step, and from MC, which uses the full return from all steps. It introduces the concept of eligibility traces, or temporary memory, to track how much credit a state should receive for a future reward [10, Chapter 12].

State credits decay over time, allowing past states to contribute to the value update with weights that decrease as they “become older”. This behavior is defined in Equation 2.16, where γ is the discount factor that determines how much future rewards are valued relative to immediate ones, and $\lambda \in [0, 1]$ is the trace decay parameter controlling how quickly the influence of previous states fades. Specifically, higher values of λ make the algorithm consider longer temporal dependencies (reducing bias but increasing variance), while lower values emphasize more recent states (reducing variance but increasing bias).

$$E_t(s) = \gamma\lambda E_{t-1}(s) + 1(S_t = s), \quad \lambda \in [0, 1] \quad (2.16)$$

The eligibility traces $E_t(s)$ are then incorporated into the TD learning update (Equation 2.17), which introduces the learning rate α and adjusts the update magnitude for each state accordingly.

$$V(s) \leftarrow V(s) + \alpha\delta_t E_t(s) \quad (2.17)$$

TD learning is essential to many RL algorithms since it enables agents to learn efficiently in online, incremental, and partially observable environments. This is the case of incremental methods, such as *State-Action-Reward-State-Action* (SARSA) algorithm and Q-learning, and Actor Critic methods.

2.2.3 Policy-Based Methods

Policy-based, or policy gradient methods, represent a category of RL that directly learn a parameterized policy to maximize expected cumulative rewards. Unlike value-based methods, these approaches adjust policy parameters directly without explicitly solving the Bellman equations. The policy is often defined stochastically, which means that it represents the probability of taking action a given state s , parameterized by a vector θ , as shown in Equation 2.18. θ represents the parameters of the policy function that determine how the agent acts in different states [10, Chapter 13].

$$\pi_\theta(a|s) = \mathbb{P}\{A_t = a \mid S_t = s, \theta_t = \theta\} \quad (2.18)$$

This parameterized policy can be defined differently depending on whether the actions are discrete or continuous [10, Chapter 13]. For discrete actions, the softmax function is often used. Such actions are common in games, simple navigation tasks, and control problems. The policy methods can handle high-dimensional and continuous action spaces naturally. In this case, a Gaussian distribution is typically employed, representing a *Probability Density Function* (PDF) over an infinite set of possible actions. This approach is more common in physics-based environments and robotics control tasks.

The goal, in these cases, can be set to optimize the agent's performance under a given policy. This can be achieved by defining a performance measure $J(\theta)$. Where the optimal θ^* can maximize the function, adjusting the policy to favor actions that lead to higher rewards [10, Chapter 13]. In the case of episodic tasks, this can be defined as in the Equation 2.19.

$$J(\theta) = V_{\pi_\theta}(s_0) = \mathbb{E}_{\pi_\theta} \left[\sum_{t \in 0:T} \gamma^t R_t \middle| S_0 = s_0 \right] \quad (2.19)$$

However, directly computing the expected return $J(\theta)$ is infeasible, as it would require evaluating rewards over all possible state-action trajectories generated by policy $\pi_\theta(a|s)$. Thankfully, the Policy Gradient Theorem provides a solution by showing that the gradient of the expected return with respect to the policy parameters θ can be estimated without needing to compute $J(\theta)$ explicitly [10, Chapter 13].

This is represented in Equation 2.20, which allows us to approximate how changes in θ affect the expected reward, enabling gradient-based optimization of the policy.

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi [Q^\pi(s, a) \nabla_\theta \log \pi_\theta(a|s)] \quad (2.20)$$

The parameter θ can then be updated according to the Equation 2.21. This update consists of adding the current parameter set θ_t to a stochastic estimate of the true gradient ascent of the performance measure $J(\theta)$ with respect to θ_t [10, Chapter 13].

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)}, \quad (2.21)$$

One of the most well-known purely policy-based techniques is the REINFORCE algorithm. This algorithm updates the policy parameters θ by estimating the total expected return G_t using Monte Carlo sampling [10, Chapter 13]. This algorithm relies on the previously mentioned theorem, as expressed in Equation 2.20.

In this manner, at the end of an episode, REINFORCE updates its parameters by following Equation 2.22. The update rule adjusts the parameters to increase the probability of actions that led to higher returns in the observed episode.

$$\theta \leftarrow \theta + \alpha G_t \nabla_{\theta} \log \pi_{\theta}(a|s) \quad (2.22)$$

This algorithm can also incorporate a baseline value function [10, Chapter 13], which improves convergence by reducing variance. As with other RL algorithms, numerous variations of this method exist.

Nevertheless, one major drawback of policy gradient methods like REINFORCE is their susceptibility to converging to local optima rather than the global optimum. Additionally, these methods alone, even despite a few tricks, often exhibit high variance in their parameter updates and can be sample inefficient, requiring many episodes to learn effectively.

2.2.4 Actor Critic Methods

One of the ways to reduce the high variance and noise due to random events is to introduce a critic. This is done by combining the policy gradient methods with the previous value-based methods, where the value function can assist the policy update [10, Chapter 13].

These methods are called Actor-Critic, since the Actor will still be updating the policy parameters θ of the policy $\pi_{\theta}(a|s)$, but in the direction suggested by the Critic, which updates the value function parameters w of the approximated Value-Function, that can be either state-value Function $V_w(s)$ or action-value Function $Q_w(s, a)$.

In a nutshell, the different methods can be described in a spectrum as represented in Figure 2.4, where the actor critic simply combines the value-based and policy-based [10, Chapter 13].

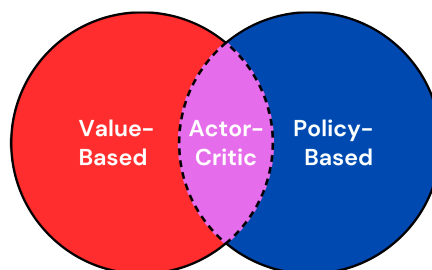


Figure 2.4: Visual representation of the main RL methods.

The steps are described in the Algorithm 2.1, where the δ , which represents the TD error, provides a lower-variance estimate, that measures how different the

current estimate is from the expected future value. The learning rate of both policy and value function is described by α_θ and α_w respectively. The rest of the code uses the previously mentioned concepts.

Algorithm 2.1: Simple Actor-Critic Algorithm

```

1 Initialize policy parameters  $\theta$ , value function parameters  $w$ 
2 loop(for each episode)
3   Initialize state  $s$ 
4    $I \leftarrow 1$ 
5   while  $s$  is not terminal do
6     Sample action  $a \sim \pi_\theta(a|s)$ 
7     Take action  $a$ , observe  $s'$ , reward  $r$ 
8      $\delta \leftarrow r + \gamma V_w(s') - V_w(s)$ 
9     if  $s'$  is terminal then
10        $V_w(s') \leftarrow 0$ 
11     end if
12      $w \leftarrow w + \alpha_w \delta \nabla V_w(s)$ 
13      $\theta \leftarrow \theta + \alpha_\theta I \delta \nabla \log \pi_\theta(a|s)$ 
14      $I \leftarrow \gamma I$ 
15      $s \leftarrow s'$ 
16   end while
17 end loop

```

Actor-Critic methods are currently the most widely used and generally more effective. However, they come with certain drawbacks. Their increased complexity and larger number of parameters lead to higher computational costs. Additionally, tuning them can be challenging, and the interaction between the actor and critic can cause convergence issues.

2.2.5 On-Policy and Off-Policy

So far, the discussion has focused on increasing the accumulated reward based on the estimation of a value function or by iterating the policy itself, sampling experience from the same policy. However, this approach assumes that the policy used to generate the data and the policy being learned are the same. This is the essence of on-policy learning, where the policy π is directly or indirectly updated using the information gathered from an agent's experience with the current policy.

One classical example of an on-policy algorithm is the SARSA. Where the action-state-value function $Q(s, a)$ is updated using the Bellman Equation approximation, like demonstrated in Equation 2.23. Where the $[r + \gamma Q(s', a')]$ represents the Q_{target} , which reflects the ideal future reward expected by the agent, given its future action a' taken in the next state s' , dictated by the current policy [24].

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)] \quad (2.23)$$

In contrast, off-policy methods provide greater flexibility. In these methods, the agent learns from experiences generated by a behavior policy μ , while simultaneously

optimizing a target policy π [24]. This enables the reuse of past experiences, whether produced by the agent itself, other agents, or even humans, to improve the current policy. As a result, such algorithms are generally more sample-efficient, which can be highly advantageous in many tasks.

However, this flexibility introduces a challenge: the collected data may not be representative of the target policy, leading to biased updates. To mitigate this issue, importance sampling can be applied to reweight experiences according to the relative probabilities of actions under both policies [24]. Following this idea, an importance-weighted return $G_t^{\pi/\mu}$ can be computed as described in Equation 2.24.

$$G_t^{\pi/\mu} = \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)} \frac{\pi(a_{t+1}|s_{t+1})}{\mu(a_{t+1}|s_{t+1})} \cdots \frac{\pi(A_T|S_T)}{\mu(A_T|S_T)} G_t \quad (2.24)$$

Besides the previously mentioned benefits, off-policy tends to create a high-variance estimate, especially if there exists a discrepancy between the data used for learning and the current policy. However, not all off-policy algorithms use this same concept. Some use bootstrapping, function approximation, or experience replay² [25].

One example of a simple off-policy algorithm that does not use importance sampling is the classical Q-learning. In this algorithm, the agent may explore using any behavior policy. However, since it is an off-policy algorithm, it will use the action generated by the behavior policy, to compute a Q-value estimation based on an optimal/greedy target policy at each step [24]. This can be demonstrated explicitly in the update Equation 2.25.

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \quad (2.25)$$

What distinguishes Q-Learning from the classical SARSA update in Equation 2.23 is that the Q_{target} is computed as $[r + \gamma \max_{a'} Q(s', a')]$ instead. In this case, the evaluation is based on the highest-value action rather than the current policy's behavior. This is why Q-Learning is also known as SarsMAX, as this is the only key difference between the two algorithms [24].

It is important to remember that choosing between on-policy and off-policy methods always involves trade-offs. On-policy methods are typically more stable and straightforward to train but require more samples, while off-policy methods are more sample-efficient due to experience reuse, though they may require careful tuning to remain stable, especially in highly dynamic or non-stationary environments, even when using techniques such as importance sampling or experience replay [26].

²Experience replay takes advantage of a replay buffer which stores transitions (s, a, r, s') in a buffer, enabling the RL agent to learn from past experiences. Instead of the states (s) and the next states (s') , this can also be exchanged for observations (o) and the next observations (o') .

Although SARSA and Q-Learning are value-based methods, there also exist off-policy and on-policy algorithms that belong to the class of policy gradient methods. These are better explored in Chapter 2.3.

2.2.6 Exploration and Exploitation

One of the most commonly used behavior policies with the Q-Learning algorithm is the famous ϵ -greedy, which is a simple way of managing the exploration and exploitation trade-off. This describes the dilemma an agent faces when deciding between trying out new actions to gain more knowledge about its environment (exploration) and using the actions it already knows to maximize immediate rewards (exploitation).

Correctly balancing this trade-off is important to ensure that the agent does not get stuck in suboptimal actions, while also preventing it from continuously exploring new options without ever committing to the best one discovered so far in the learning process [27].

Knowing this, the ϵ -greedy, as described in the Equation 2.26 choses a random action with a probability dictated by the exploration parameter ϵ , in the case of the random action is not chosen the policy will select an action a^* which is mapped to the highest estimated value at state s .

$$a_t = \begin{cases} \text{a random action } a \in \mathcal{A}, & \text{with probability } \epsilon, \\ \arg \max_{a \in \mathcal{A}} Q(s_t, a), & \text{with probability } 1 - \epsilon. \end{cases} \quad (2.26)$$

One also could think of a naive-approach, where the agent explores more actions at the beginning of the learning process, and reduce this exploration over time as it gain more knowledge of the environment. This is the principle of the epsilon decay, where the underlying principle is to allow extensive exploration when the agent is still unfamiliar with the environment, thereby enabling it to gather a broad range of experiences [27]. As learning progresses, the agent leverages this accumulated knowledge to make more informed decisions by exploiting the best-known actions.

The epsilon decay is generally expressed as in Equation 2.27, where the term $\epsilon_{\max}$ represents the initial exploration rate, ensuring that the agent starts with a high degree of randomness in its actions. Conversely, $\epsilon_{\min}$ sets a lower bound, guaranteeing that a minimal level of exploration continues even after significant learning has taken place. The exponential decay factor $e^{-\lambda t}$ is crucial, as it regulates how quickly the exploration rate diminishes over time, with λ serving as the decay rate.

$$\epsilon_t = \epsilon_{\min} + (\epsilon_{\max} - \epsilon_{\min}) \cdot e^{-\lambda t} \quad (2.27)$$

This same idea can also be applied to a linear decay, which is often simpler and commonly used. It reaches the end at t step, and then stays constant until the end of the episode at T , as can be denoted in Equation 2.28.

$$\epsilon_t = \max \left(\epsilon_{\min}, \epsilon_{\max} - (\epsilon_{\max} - \epsilon_{\min}) \cdot \frac{t}{T} \right) \quad (2.28)$$

The difference between them will mostly depend on the use case and performance analysis. Figure 2.5 illustrates the visual difference between them.

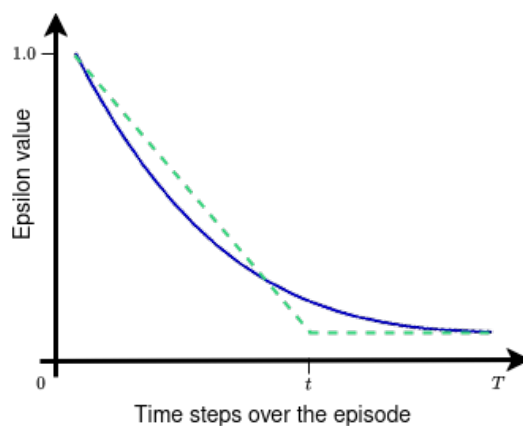


Figure 2.5: Difference between linear and exponential decay.

Other techniques, such as curiosity-driven exploration³ or entropy-based exploration⁴, have also been developed to handle this trade-off. These methods encourage the agent to explore based on novelty or uncertainty, which can complement strategies like decay in balancing exploration and exploitation.

2.3 Deep Reinforcement Learning

In 2013, Google DeepMind unveiled a breakthrough with the *Deep Q-Network* (DQN) algorithm [29], which learned to play the Atari 2600 games directly from raw pixels. However, only with the 2015 Nature publication [30], this brought widespread attention and validation to the approach in the scientific community.

Directly playing Atari games can be seen as a simple problem for us humans, however, the same can't be said for a computer. Because the learning process, requires it to understand the data provided by a high-dimensional input space from

³This is accomplished using an *Intrinsic Curiosity Module* (ICM), which provides an intrinsic reward based on the agent's error in predicting the next state from the current state and the action taken [28].

⁴This is better explored in Chapter 2.3, with the *Soft Actor-Critic* (SAC) algorithm.

the raw pixels [29], and the learning of complex dynamics and long-term Dependencies from scratch. This approach is usually termed as *end-to-end learning*, and it represents the ideal example of full learning example without heuristics or human intervention.

In this way, this time period marked the beginning of a new era in artificial intelligence research, when deep learning and classical algorithms came together.

Deep Q-Network

Looking at more technical detail, the *Deep Neural Network* (DNN) allows for better function approximation, leading to better generalization across similar states and scalability. In the case of the DQN, due to the use of neural networks, it defines a *Mean Squared Bellman Error* (MBSE) loss function to minimize the difference between the current Q-value estimate [30], as can be formulated in Equation 2.29.

It works by trying to minimize the difference between the Q-value predicted by the current network $Q(s, a; \theta_i)$ and the target Q-value $r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$, where the θ represents the weights/parameters of the current neural network, while the θ^- represents the parameters of the target network, which is a fixed copy of the Q-network. The target network does not get updated during training, and it is periodically replaced with the current network's weights. This strategy helps to stabilize learning by introducing bias and reducing variance.

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right] \quad (2.29)$$

The DQN, do it's off policy nature, allows for the use of experience replay, which is stored in the dataset/replay buffer D , with a uniform sampling U . Some modifications can be applied to the base DQN algorithm [31], in the way that it processes and uses these experiences. Due to the maximization over possible actions in the target network, the DQN only allows for discrete action-spaces, which can be seen a great drawback, specially in real-world robotics applications.

The DQN is a well-known example of deep RL, but it is somewhat of an exception. Most modern algorithms combine value-based and policy gradient methods, leading to actor-critic frameworks.

While there's no fixed categorization, these modern actor-critic model-free algorithms can be broadly classified by their policy type (deterministic or stochastic) and whether they are on-policy or off-policy.

Deep Deterministic Policy Gradient

A representative example of a deterministic off-policy algorithm that extends reinforcement learning to continuous action spaces is the *Deep Deterministic Policy Gradient* (DDPG) [32]. DDPG adopts the Q-learning framework for critic evaluation, while enabling continuous control through a deterministic policy-gradient actor. To improve training stability, it introduces the use of Polyak (soft) updates, which reduce target variance by slowly updating both the target-actor network θ_{targ} and the target-critic network ϕ_{targ} , as expressed in Equation 2.30. The Polyak parameter ρ controls the smoothness of the update, where the closer it is to 1, the slower and smoother the update becomes. This soft update has become standard for other off-policy algorithms, which will be mentioned later.

$$\begin{aligned}\theta_{\text{targ}} &\leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta \\ \phi_{\text{targ}} &\leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi\end{aligned}\tag{2.30}$$

In its standard formulation, DDPG maintains four neural networks: the actor and critic, together with their corresponding target networks [32]. The critic, in this way, is updated by minimizing the loss function described in the Equation 2.31, where the ϕ and ϕ_{target} are the parameters of the critic and target-critic networks, respectively, while the θ_{target} represent the parameters of the target-actor.

$$L(\phi, \mathcal{D}) = \mathbb{E}_{(s,a,r,s',d) \sim \mathcal{D}} \left[(Q_{\phi}(s, a) - y(r, s', d))^2 \right]\tag{2.31}$$

This is a type of MBSE loss function, which is closely related to the Bellman approximation equation used in Q-learning, as previously mentioned in the Equation 2.29. In this loss function, the $y(r, s', d)$ represents the update of the target-critic network, as it can be seen in Equation 2.32. However, instead of iterating over all possible actions, the target actor network $\mu_{\theta_{\text{target}}}(s')$ determines the best action a' .

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))\tag{2.32}$$

As mentioned, due to being deterministic and continuous, the policy is learned by the actor network, as denoted in the Equation 2.33. Where the θ represents the parameters of the network.

$$\max_{\theta} \mathbb{E}_{s \sim \mathcal{D}} [Q_{\phi}(s, \mu_{\theta}(s))]\tag{2.33}$$

Finally, to explore better, especially at the beginning where it is more crucial, noise such as Gaussian noise is added to actions at training time [33], allowing to find useful learning signals. The DDPG framework can be used as the baseline for other algorithms built on top of it.

Twin Delayed DDPG

One well-known algorithm that addresses the limitations of DDPG is *Twin Delayed DDPG* (TD3), specifically targeting issues such as overestimation bias and high variance in policy updates. To mitigate these problems, TD3 [34] introduces several key improvements.

The first modification involves the use of two twin networks for the critic, Q_{ϕ_1} and Q_{ϕ_2} , as well as for the target critic, $Q_{\phi_{1,\text{targ}}}$ and $Q_{\phi_{2,\text{targ}}}$, hence the name “Twin” in the algorithm. This process is called double-Q learning, where only the network that produces the minimum Q-value for the target is used for the update of the loss function, as can be described by the Equation 2.34.

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{i,\text{targ}}}(s', a'(s')) \quad (2.34)$$

Then, each critic is updated separately by regressing its Q value toward this same target, allowing an independent approximation of the Q-values as demonstrated in Equation 2.35.

$$\begin{aligned} L(\phi_1, \mathcal{D}) &= \mathbb{E}_{(s,a,r,s',d) \sim \mathcal{D}} \left[(Q_{\phi_1}(s, a) - y(r, s', d))^2 \right], \\ L(\phi_2, \mathcal{D}) &= \mathbb{E}_{(s,a,r,s',d) \sim \mathcal{D}} \left[(Q_{\phi_2}(s, a) - y(r, s', d))^2 \right]. \end{aligned} \quad (2.35)$$

Just like the DDPG, to increase exploration stability, the TD3 also adds a small amount of Gaussian noise to the action selected by the target actor, when computing the target value [34]. This smoothing is then clipped to ensure that perturbation remains small and bounded, regularizing the policy by preventing the critic from exploiting sharp peaks in the Q-value estimates [34].

Other important feature, is the fact that actor network updates less frequently than the critic networks [34]. This “delay” in updating the actor reduces the variance in the policy gradient, leading to more stable learning.

These are some examples of deterministic off-policy actor-critics algorithms, however there also exists completely opposite approaches, like stochastic on policy-algorithms inside the actor-critic methods.

Trust Region Policy Optimization

One of the best examples is the *Trust Region Policy Optimization* (TRPO) algorithm [35], which, in contrast to vanilla policy gradient methods, that can suffer from catastrophic policy updates due to overly large steps, enforces a trust region constraint to limit the divergence between the new and old policies. This ensures stable and monotonic⁵ improvements with theoretical guarantees.

This is achieved using methods like *Kullback–Leibler* (KL) divergence, denoted by D_{KL} in Equation 2.37. The maximization of the surrogate⁶ loss function $L(\theta)$ ensures that the new policy $\pi(\theta)$ assigns higher probability to good actions and lower probability to bad actions, ultimately leading to improved performance [35].

$$\max_{\theta} \quad L(\theta) = \mathbb{E}_{s,a \sim \pi_{\theta_{old}}} \left[\frac{\pi_{\theta}(a|s)}{\pi_{\theta_{old}}(a|s)} A(s, a) \right] \quad (2.36)$$

$$\text{s.t.} \quad \mathbb{E}_{s \sim d^{\pi_{\theta_{old}}}} [D_{KL}(\pi_{\theta_{old}}(\cdot|s) \parallel \pi_{\theta}(\cdot|s))] \leq \delta. \quad (2.37)$$

Solving directly with the KL constraint is difficult and computationally expensive because it turns the problem into a non-convex optimization problem. Therefore, to approximate the surrogate advantage function, the concept of Lagrangian duality is applied, with some mathematical modifications, to ensure that the approximation remains computationally feasible, besides it still remaining a second-order optimization technique [36].

Proximal Policy Optimization

The *Proximal Policy Optimization* (PPO) algorithm [37] aims to address the same issue that tries to TRPO solve: how to perform large updates to the policy while ensuring that the performance does not degrade? But unlike TRPO, which uses a second-order method to constrain the policy update, PPO simplifies this by introducing a first-order method with a trust region constraint to achieve stable policy updates [38].

PPO’s key idea is to strike a balance between making large improvements to the policy and preventing overly large updates that could destabilize learning. It achieves this through an adaptive objective function that includes a clipping mechanism (in PPO-Clip [37]) or a KL divergence penalty (in PPO-Penalty [37]), which ensures that the new policy doesn’t deviate too far from the old policy. Since the

⁵Monotonic refers to a mathematical term describing a function that is either always increasing or always decreasing on its domain. In the context of RL, it means the algorithm will always aim to increase the reward.

⁶A surrogate is a function that approximates another function.

PPO-Clip is the most common approach and a bit more efficient, this will be the one that will be explored.

The PPO-Clip variant, the objective function is modified as shown in Equation 2.38. Where $r_t(\theta)$ is the probability ratio between the new and old policies, A^t is the estimated advantage, which is commonly estimated using the *Generalized Advantage Estimation* (GAE) [21].

In this case, ϵ is a crucial hyperparameter that determines the clipping range. It is added or subtracted depending on whether the advantage function is positive or negative. The clipped objective constrains the policy update by limiting how much the probability ratio can deviate from 1, thereby preventing the new policy from making overly large changes.

$$L(\theta) = \mathbb{E}_{s, a \sim \pi_{\theta_{\text{old}}}} \left[\min \left(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t \right) \right]. \quad (2.38)$$

The PPO is clearly more efficient, but due to the first-order optimization, but it still can have trouble to operate in high-dimensional spaces compared to the TRPO. However, both of these approaches are still on-policy, besides having stochastic policies.

Soft Actor-Critic

The *Soft Actor-Critic* (SAC) algorithm [39] is designed to address the issue of exploration versus exploitation in reinforcement learning in a new way. While ensuring stable and efficient learning, with both a stochastic policy and off-policy approach.

It essentially changes the formulation of the RL problem, to include the concept of Entropy. In this way, the Entropy H is used to obtain a quantitative measure of how unpredictable or random the actions of the entire action space are, given the policy $\pi_{\theta}(\cdot|s)$ for a particular state s , as formally in the Equation 2.39. With this in mind, one of the goals is to regularize this entropy, with the purpose of promoting a sustained exploration, avoiding the policy to stay suboptimal [40].

$$H(\pi_{\theta}(\cdot|s)) = -\mathbb{E}_{a \sim \pi_{\theta}(\cdot|s)} [\log \pi_{\theta}(a|s)] \quad (2.39)$$

There are two SAC implementations, one that uses a fixed entropy regularization coefficient/temperature α , and another that enforces an entropy constraint by varying α over the course of training [40]. This same coefficient is often denoted as “temperature”, but for the sake of simplicity, this coefficient will be considered as fixed.

The target Q-value $y(r, s', d)$, which is also updated using Polyak averaging, is used in the same sense as the one in the TD3 and DDPG, as mentioned in Equation 2.32. However, it incorporates the concept of Entropy mentioned earlier, which changes the function $y(r, s', d)$ as it is possible to observe Equation 2.40. To emphasize that the subsequent actions must be sampled from the policy again, $\tilde{a}'$ is used in place of a' . On the other hand, r and s' , must originate from the replay buffer [40].

$$y(r, s', d) = r + \gamma(1 - d) \left(\min_{j=1,2} Q_{\phi_{\text{target},j}}(s', \tilde{a}') - \alpha \log \pi_{\theta}(\tilde{a}'|s') \right), \quad \tilde{a}' \sim \pi_{\theta}(\cdot|s'). \quad (2.40)$$

The policy loss function is then used, computed as demonstrated in Equation 2.41, which similar to the one demonstrated by the DDPG and TD3, however it includes the min-double-Q trick, the stochasticity, and the entropy term [40].

$$\max_{\theta} \mathbb{E}_{\substack{s \sim \mathcal{D} \\ \xi \sim \mathcal{N}}} \left[\min_{j=1,2} Q_{\phi_j}(s, \tilde{a}_{\theta}(s, \xi)) - \alpha \log \pi_{\theta}(\tilde{a}_{\theta}(s, \xi)|s) \right], \quad (2.41)$$

2.4 Summary

In this manner, it is possible to develop a high-level summarizing table, such as the one in Table 2.1, that captures the essential characteristics of the previously mentioned deep RL algorithms.

Table 2.1: Comparison of Reinforcement Learning Algorithms

Algorithm	Value/Policy-based/AC	On/Off-policy	Action Space	Policy Type
DQN	Value-based	Off-policy	Discrete	Deterministic
DDPG	Actor-critic	Off-policy	Continuous	Deterministic
TD3	Actor-critic	Off-policy	Continuous	Deterministic
TRPO	Actor-critic	On-policy	Continuous	Stochastic
PPO	Actor-critic	On-policy	Discrete/ Continuous	Stochastic
SAC	Actor-critic	Off-policy	Continuous	Stochastic

These are just a few examples within the realm of model-free methods. However, as new approaches continue to emerge. For example, researchers have also explored modern model-based RL for real-world problems that demand high efficiency and stability [41].

These models, incorporate recent advancements from control theory and system identification, alongside the concepts used in model-free RL. Model-based approaches also utilize the concept of a *World Model* [41], allowing the agent to internally simulate and predict potential outcomes in order to determine the best course of action before actually interacting with the environment.

New sub-fields of RL have also emerged that have a great significance to the world of robotics. One of the examples is the rise of the *Inverse Reinforcement Learning* (IRL) [42], where basically the goal is to infer the reward function of an agent based on its observed behavior, rather than learn a policy that maximizes expected cumulative reward. This is done by relying on expert demonstrations, but should not be confused with the concept of *Imitation Learning* (IL) [43], which is a supervised learning method and focuses on directly mimicking the expert's actions without seeking to uncover the underlying objectives that drive the behavior.

Another example, is the growth of the *Safe-RL* [44], where the problem focuses on ensuring that the agent's learning process doesn't cause harmful consequences or risks while exploring or interacting with the environment.

And finally, another relevant topic are the *Robust-RL* algorithms [45] where the main concern is on making RL agents more resilient to uncertainties in the environment. The goal is, in this way, to ensure the agent performs well even under conditions where the environment may be unpredictable or change unexpectedly. This is a necessary condition, for example, to a robot operate on a real-world environment.

Rather than existing in isolation, these RL subfields are deeply intertwined. This overlap enables the application of concepts across different areas, fueling new research and development.

Chapter 3

Related Work

This section explores how UAV navigation for *Photovoltaic* (PV) panel inspection has been approached and solved using traditional methods, highlighting key requirements, commonly used techniques, and existing challenges. Building on the foundational *Reinforcement Learning* (RL) concepts previously introduced, the current chapter examines how recent RL algorithms can be applied to vision-based tasks while ensuring that UAVs operate effectively under real-world conditions. Finally, a critical review is performed to serve as a starting point to address the thesis problem.

3.1 PV Inspection Procedures

Compared to manual PV panel inspections by technicians, UAVs offer a more effective approach due to their speed, ease of visualization, and enhanced analysis capabilities.

UAVs can be equipped with multiple camera sensors capable of capturing different light spectra, allowing the detection of faults that may be difficult to identify manually or with ground systems [46]. While visual cameras are still commonly used to detect surface issues, such as cracks, dirt, and bird droppings, thermal imaging supported by *Infrared* (IR) sensors enables the identification of damaged cells that may appear unaffected to the naked eye. Finally, *Electroluminescence* (EL) cameras enhance fault detection by revealing internal issues such as micro-cracks and cell degradation, which are not visible through visual or IR imaging. EL imaging requires the application of direct current to PV panels, temporarily halting energy

production during daylight hours. Effective UAV use of these cameras requires automation, which directly impacts inspection costs and image quality [47]. These techniques are well summarized by Meribout et al. [46].

Typically, flight planning is simplified by operating at higher altitudes using Boustrophedon paths¹, guided by predefined GNSS waypoints (set before the inspection). This approach creates an orthomosaic by overlaying images, which can be examined later. Figure 3.1 illustrates the application of this concept by Cardoso et al. [49]. Although it offers a comprehensive overview of the scene, high-altitude flight limits the resolution of individual images and does not work with empty zones [50].

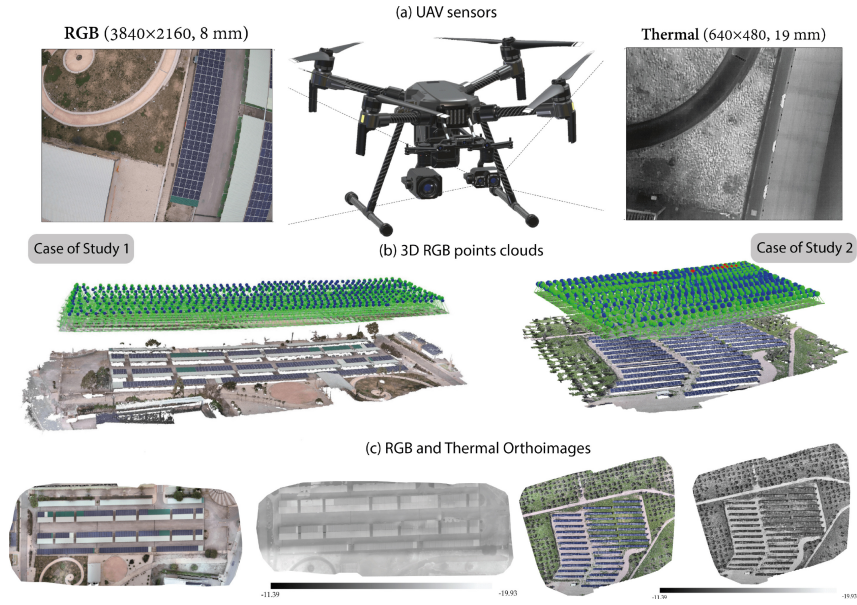


Figure 3.1: Creation of orthomosaics of a PV solar plant, through visual and thermal images [49].

Numerous studies confirm the value of high-resolution photos in PV plant inspection [47]. A flight campaign conducted at 30 m and 80 m above the ground by Gallardo-Saavedra et al. [51] only detected 37% and 24% of anomalies respectively, compared to the total close-up manual inspections, highlighting the impact of resolution on fault detection.

Consequently, this demand for high-resolution images requires UAVs to evolve from manual or automatic piloting to fully autonomous close-range operation and inspection [4].

Several research works have attempted to address these needs. Morando et al. [50] combines classical computer vision techniques to detect and segment solar panels, with a traditional *Proportional Integral Derivative* (PID) controller, to maximize

¹This is a movement pattern where each line of travel alternates direction. It is used in several robotics application domains, such as vacuuming, mapping or area scanning and inspection [48].

the time the UAV remains aligned with the center of the panel as it flies over the panel array, as illustrated in Figure 3.2. The proposed approach implements way-point navigation and real-time PV panel detection based on thermal and visual imaging.

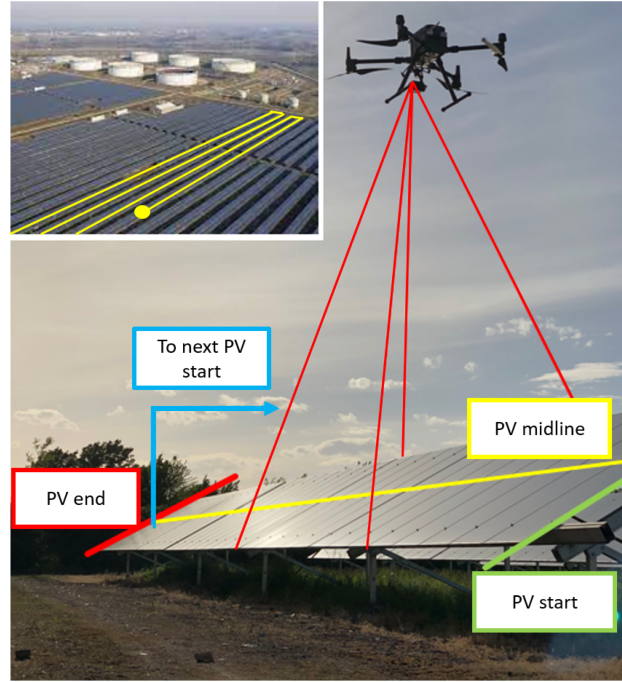


Figure 3.2: PV panel close inspection [50].

Roggi et al. [52] explores visual servoing to inspect PV panel arrays with GNSS position errors, using the camera FOV along with classical computer vision techniques such as Canny Edge and Hough Lines, as can be seen in the Figure 3.3. The results show that the proposed approach reduces the number of frames with incomplete coverage of the target PV panel arrays compared to the non-corrective approach.

However, these traditional approaches lack robustness to changes in lighting and environmental conditions, a common problem with traditional methods [53].

At the time of writing, no studies were found that applied non-traditional methods of perception and control of UAVs for the detailed inspection of PV panel arrays.

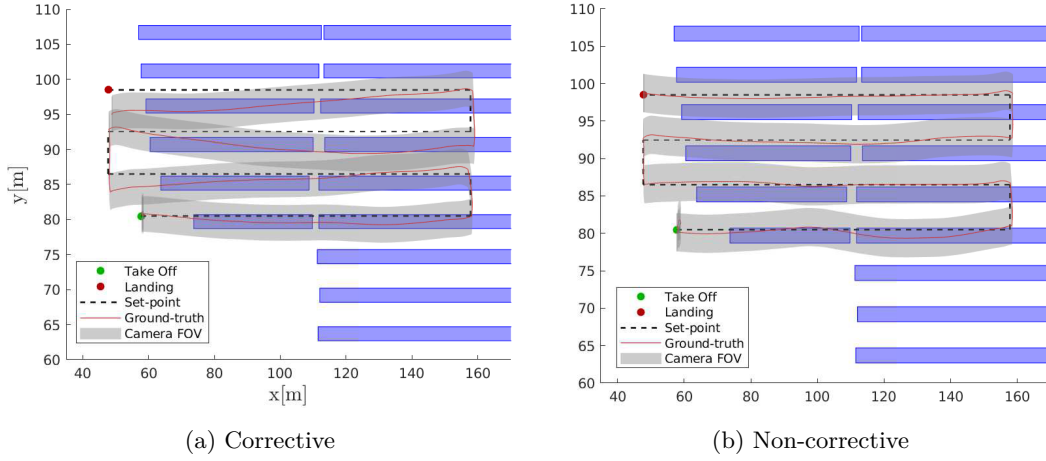


Figure 3.3: Corrective and non-corrective methods based on the UAV FOV during flight over PV grid [52].

3.2 Vision-Based RL

The use of RL for vision-based robotics tasks has become increasingly attractive, since it enables end-to-end mapping from raw image pixels to agent actions. This setup enables simultaneous learning of perception and control through gradient-based optimization of cumulative reward. However, compared to the more conventional state-based RL, vision-based RL must handle high-dimensional inputs, leading to greater complexity and sample inefficiency [54]. To mitigate this, there are methods that decouple perception from control.

3.2.1 Decoupled Techniques

The decoupling process involves separating the perception component from the RL problem. For visual tasks, this can be done using classical computer vision techniques or specialized neural networks, such as object detection models like the pre-trained *You Only Look Once* (YOLO), while keeping them independent from RL.

These detection models can output a bounding box that defines a *Region of Interest* (ROI), which is then provided to the RL agent as its state. Such approaches have been applied in UAV applications, as demonstrated in [55, 56, 57].

Research like [58, 59] explores pre-trained autoencoders using unlabeled datasets that are relevant to a particular vision task. The latent representations generated by these trained encoders are subsequently utilized by the RL algorithm. This approach enables the RL algorithm to operate on lower-dimensional, feature-rich data, often leading to enhanced efficiency and performance in learning. Another example of decoupling perception is the work of Wang et al. [60], where the focus is on learning a general visual representation rather than a task-specific representation. They use unsupervised keypoint detection to extract foreground regions, followed by an

unsupervised visual-attention mechanism to produce a refined foreground mask for each video frame. This fully unsupervised pipeline requires three separate training phases to reach the final control stage.

Despite the advantages of these techniques, particularly for specialized tasks, RL often emphasizes end-to-end learning.

3.2.2 End-to-End Learning

Model-free and model-based RL represent two core approaches to implement end-to-end vision-based tasks. Model-free RL learns policies directly from raw high-dimensional inputs at the cost of sample inefficiency, while model-based RL improves efficiency by learning a predictive model of the environment dynamics. End-to-end techniques have already been extensively used to solve real-world robotics problems involving UAVs, such as in the work of Neves et al. [61].

The *Deep Q-Network* (DQN) is a model-free method introduced by Mnih et al. [30] that integrates *Convolutional Neural Networks* (CNNs) and RL to implement end-to-end learning. The CNN, trained via backpropagation using the reward signal from the RL agent, extracts the visual features of the image. This concept has been applied to complex UAV control and navigation tasks, namely by [62, 63].

Laskin et al. [64] and Yarats et al. [65] proposed model-free algorithms to address sample inefficiency. The *Contrastive Unsupervised Representations for Reinforcement Learning* (CURL) algorithm of Laskin et al. [64] combines data augmentation and contrastive loss, which learns embeddings² that group similar and separate dissimilar samples. Xing et al. [67] applied the CURL model-free method to UAV visual flight control. The *Data-Regularized Q-learning* (DrQ)-v2 model-free algorithm by Yarats et al. [65] also performs data augmentation, achieving a sample efficiency similar to model-based methods. This updated version of the original DrQ algorithm, incorporates a lighter-weight augmentation, configures only three hyperparameters (replay buffer size, mini-batch size and learning rate) and uses DDPG at its core, as shown in Figure 3.4.

More recently, Fujimoto et al. [68] proposed the MR.Q algorithm. MR.Q combines model-free Q-learning with learned dynamics by using an ensemble of transition models to locally linearize the Q-function, enhancing generalization and sample efficiency without explicit planning. On several *DeepMind Control* (DMC) visual control tasks, MR.Q outperformed state-of-the-art algorithms such as DreamerV3 and DrQ-v2, demonstrating superior sample efficiency and generalization.[69]. However, this algorithm has not yet been tested in real-world conditions.

²Embeddings are learned vector representations that map high-dimensional or discrete data into a continuous, lower-dimensional space, preserving semantic or structural relationships among the original items [66].

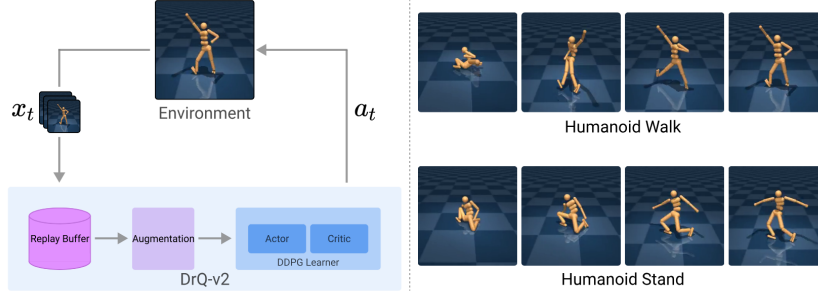


Figure 3.4: DrQ-v2 architecture [65].

The Dreamer-V3 model-based algorithm developed by Hafner et al. [70] is the latest version of the Dreamer series. This algorithm uses a world model that lets the agent simulate or “dream” future trajectories in a learned latent space, allowing planning without interacting with the real environment. In this way, Dreamer-V3 can learn a range of different control tasks through the actor-critic network and do planning, from the learned transition dynamics, with a set of fixed hyperparameters. This architecture is represented in Figure 3.5.

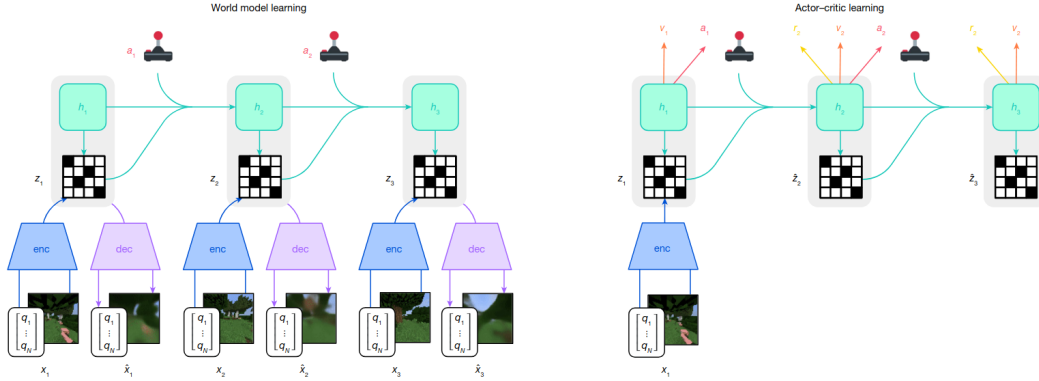


Figure 3.5: Dreamer-V3 architecture [70].

Dreamer-V3 performs well across tasks, particularly in sparse reward settings, due to its long-term planning capabilities. It has been successfully applied to real-world drone racing, as demonstrated by Romero et al. [71], achieving high performance despite requiring extensive training time (approximately 240 h).

Understanding the dynamics directly from frames can be quite challenging, as each frame contains only the information available at that moment, becoming partially observable. Establishing temporal relationships between frames is therefore essential. This principle underlies the frame-stacking technique, in which an agent’s observation is a buffer holding the last n frames. However, if the buffer simply stores the most recent n frames, the temporal window it covers is limited by the *Frames per second* (FPS). Consequently, only short-term dynamics can be represented with

a reasonably sized buffer. To capture slightly longer-term dynamics, frame skipping is typically combined with frame stacking. This combination became widely known after the DQN paper [30] and has since become a standard practice in most vision-based reinforcement-learning applications. Figure 3.6 illustrates the technique in detail for an Atari game.



Figure 3.6: Illustration of frame stacking (and frame skipping) in an Atari game [72].

In this way, frame stacking can be thought of as an approximation of the belief state, where the true state is implicitly estimated based on a fixed window of past observations [73].

Multimodality is another component that can improve end-to-end applications, such as in the case of UAVs applications, by providing complementary sensory information. Fusing data from cameras, *Light Detection and Ranging* (LiDAR), *Inertial Measurement Unit* (IMU), and *Global Positioning System* (GPS) enhances perception and state estimation and decision-making in complex real-world environments, as demonstrated in the works [74, 75].

3.3 Robust RL

Robust RL focuses on improving an agent’s performance under disturbances, uncertainties, and changes in the environment. Robustness is also fundamental to reducing the sim-to-real gap, one of the biggest challenges of RL, contributing to the deployment of reliable real-world applications.

An agent that is robust to physical disturbances, particularly in the case of UAVs, has to withstand sudden gusts of wind or unexpected forces [76]. In contrast, an agent that is robust to environmental changes is capable of maintaining high performance even when key aspects of its operating domain shift beyond the training distribution [76].

In the literature, there are three main, but not exclusive, methods to achieve this desired overall robustness: domain randomization, noise injection, and the use of a protagonist-adversary architecture.

3.3.1 Domain Randomization

Domain randomization is a widely used approach for increasing an agent’s robustness against different environment changes and effectively bridging the sim-to-real gap [77], largely because it is conceptually simple. In practice, domain randomization begins by selecting the simulator’s most critical parameters to vary. These parameters are typically grouped into visual, dynamic, and kinematic or structural categories, each dealing with different parts of the simulation [77].

Visual randomizations, as can be observed in Figure 3.7, include variations in light intensity or colours [78]. It is also possible to apply random textures to objects and backgrounds, changing lighting directions and intensities randomly, altering camera settings, such as the *Field Of View* (FOV) or focal length.

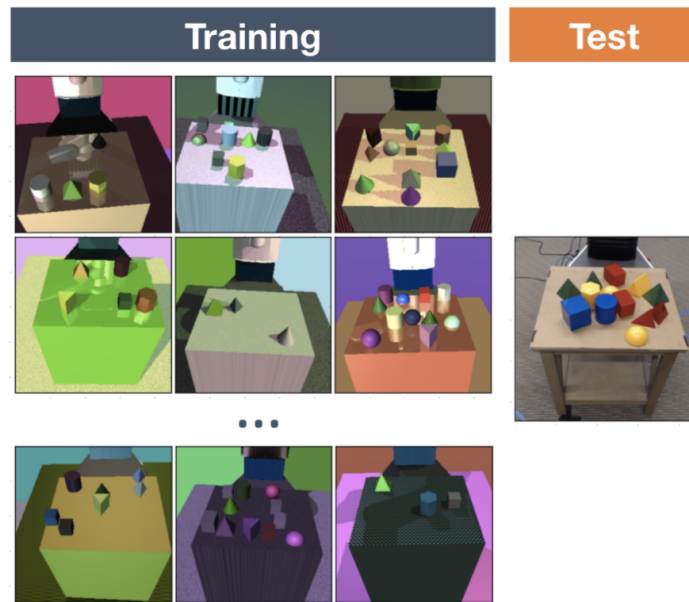


Figure 3.7: Domain randomization of visual inputs [78].

On the other hand, dynamic randomizations cover variations in physical properties, including link masses, joint friction, actuator gains or delays, and contact parameters such as restitution and friction [77]. This can be useful when sharing the same policy across different robots.

Crucially, it is necessary to balance the magnitude of randomization with training stability. If the parameter ranges are too large from the outset, the task may become unsolvable. Therefore, it is common to adopt a curriculum strategy [79], beginning with narrow variations and gradually expanding the range as the agent’s performance stabilizes. Domain randomization has been widely used in the scope of UAV control, namely by [80, 81, 82].

3.3.2 Noise Injection

Although closely related, noise injection is distinct from domain randomization. In contrast to modifying environment parameters, noise injection involves adding randomness directly to observations, actions, states, or even network parameters [83]. This is done to regularize the learning process or encourage exploration, resulting in policies that are more robust and less sensitive to unmodeled disturbances.

Noise injection has been applied to enhance learning performance of UAVs across various tasks. Villanueva and Fajardo [84] injected Gaussian noise into the network activations by adding a Gaussian Noise layer after the convolutional backbone, which processes visual inputs. The method provided a stable exploration, leading to a more robust policy. Duisterhof et al. [85] injected Gaussian noise directly into the simulated light sensor measurements at each time step to mirror the highly variable readings observed in real flights. By training under these per-step perturbations, the network learned to smooth and filter inputs, extracting a stable, low-pass-filtered gradient feature, and consequently achieved a 94 % success rate in cluttered rooms.

3.3.3 Adversarial Framework

However, the previously mentioned approaches struggle to capture rare events and lack true worst-case guarantees. To address this, researchers have developed various frameworks inspired by the research of [86], which sees the robustness problem as a protagonist and adversary challenge.

In these formulations, the environment becomes a two-player zero-sum game, with the disturbance modelled as the adversary and the main agent as the protagonist [86]. The algorithm simultaneously learns two policies: one for the adversary and one for the protagonist, optimized under the minimax objective defined in Equation 3.1, where the R^{1*} represents the optimal equilibrium reward, R^1 the reward function of the protagonist agent (player 1), and the ν and μ represent the policy of the adversary and the protagonist respectively.

$$R^{1*} = \min_{\nu} \max_{\mu} R^1(\mu, \nu) = \max_{\mu} \min_{\nu} R^1(\mu, \nu) \quad (3.1)$$

This approach is inspired by the H_{∞} control framework, which minimizes the worst-case impact of disturbances. Similarly, it enables the protagonist to maximize reward under adversarial conditions by preparing for the worst-case scenarios [86]. The protagonist achieves its maximum guaranteed return at the saddle point, where neither the adversary can further decrease the return nor the protagonist can further increase it, creating a policy that is robust to environmental uncertainties and perturbations [86].

The adversarial framework has been validated in various UAV applications, as shown in the works of Wan et al. [87], Hsu et al. [88], Deshpande et al. [89].

3.4 Summary

Classical waypoint-based UAV inspections are reliable for broad surveys but often miss high-resolution details due to high-altitude operation. Vision-assisted navigation enables low-altitude flight, using either traditional feature-based methods, which usually lack robustness, or supervised learning-based detectors and trackers, which require task-specific training and tuning.

Reinforcement Learning (RL) approaches, such as classical end-to-end agents that rely on feature extractors like CNNs, have demonstrated strong and proven results. Recent algorithms that explore data augmentation (e.g., CURL, DrQ-v2) and model-based planning (e.g., DreamerV3) show promising performance in terms of sample efficiency and effective learning. However, the lack of mature and proven frameworks remains a major obstacle to developing a complete solution.

Robustness strategies, such as domain randomization and noise injection, constitute simple yet effective methods to reduce the sim-to-real gap. In contrast, adversarial training, although effective, should be considered a last resort due to its complexity. Few studies have explored these strategies for PV array inspection, and there are currently no open benchmarks for comparison.

Drawing from this analysis, this thesis aims to address this identified gap by developing and training a visual-based RL agent. The proposed methodology, which balances advanced capabilities with practical implementation, comprises:

- End-to-end model-free RL agent: This design avoids the complexities of decoupled perception systems by learning control-relevant features directly from raw visual data via a CNN. The resulting controller is inherently modular and can be easily integrated into a larger mission plan governed by high-level waypoints.
- Temporal awareness: To effectively navigate, the agent must perceive its motion. This can be achieved through frame stacking, a standard technique that provides the model with the necessary temporal context to understand its own state better.
- Pragmatic robustness: To ensure reliable performance and facilitate sim-to-real transfer, the agent is trained using domain randomization and noise injection. This approach is chosen for its proven effectiveness and simplicity of implementation over more complex methods like adversarial training.

In the next Chapter 4, a design solution will be proposed, taking into account this summarized information as well as the details presented in Chapter 2.4.

Chapter 4

Design of the Solution

This chapter presents a UAV navigation solution that jointly addresses the problem of perception and control for the task in question. It is supported by both the theoretical foundations and the *State-of-the-art* (SOTA), detailed in the previous chapters.

4.1 Problem Formulation

As mentioned, the problem is to teach a quadcopter UAV to track and navigate autonomously over a array of photovoltaic panels without relying on GNSS data for alignment.

The UAV, represented by the body frame $\mathcal{F}_b = \{x_b, y_b, z_b\}$, will be controlled using high-level velocity commands in the body frame, specifically ω_z and v_y , rather than directly controlling the rotor speeds. This simplifies the problem to some extent. The UAV will maintain a fixed altitude z and a constant forward velocity v_x . It will navigate relative to the world frame $\mathcal{F}_w = \{x_w, y_w, z_w\}$ and will be equipped with a gimbal, allowing to configure the position of the camera frame $\mathcal{F}_c = \{x_c, y_c, z_c\}$, as shown in Figure 4.1.

In this context, the best method to capture the position of the UAV relative to the PV solar panels is through the visual camera sensor. As such, the visual camera will serve as the primary source of observations for the state of the agent. Since sensor readings can only partially observe the full system state, the problem is formulated as a POMDP using a model-free RL agent. Since model-free methods

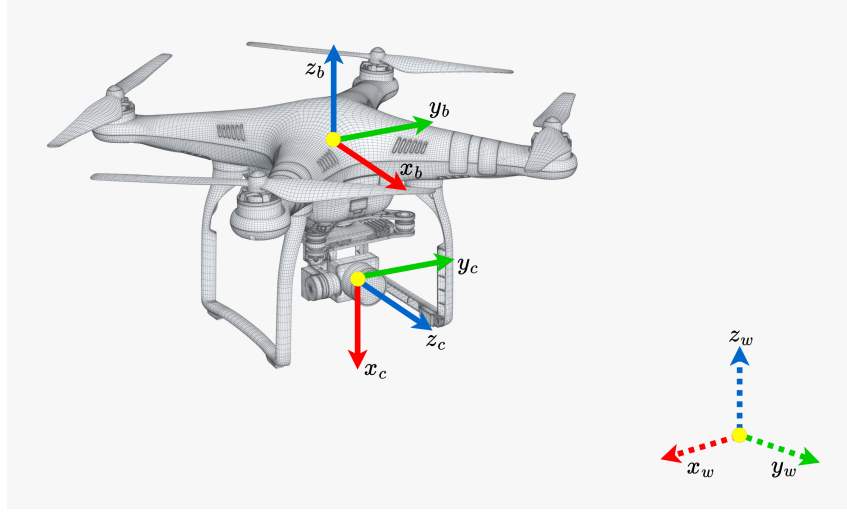
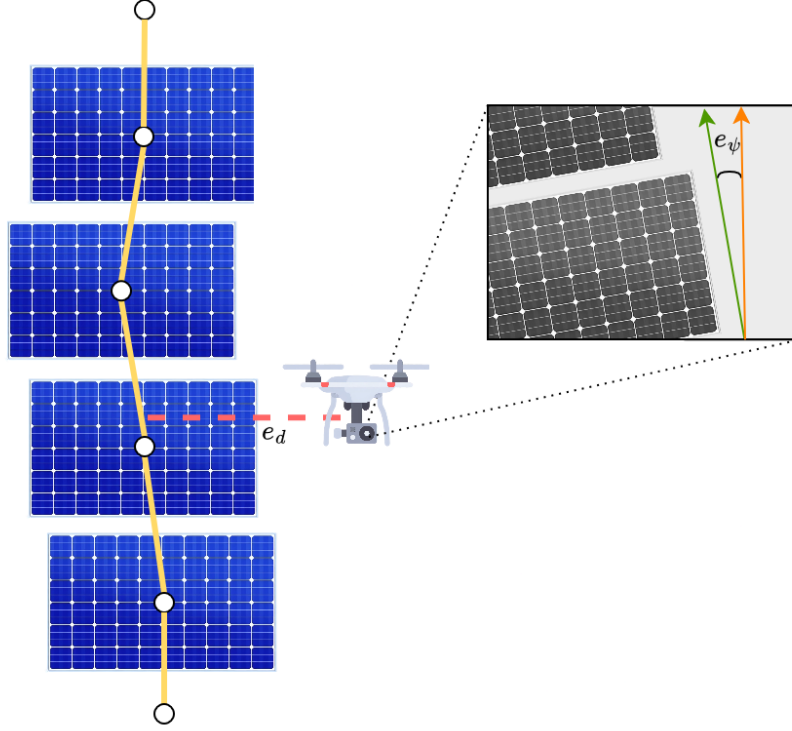


Figure 4.1: Camera and body frames of the UAV, shown with respect to the world frame.

do not require knowledge of the transition dynamics $P(s'|s, a)$, the problem can be represented by the tuple $(\mathcal{S}, \mathcal{A}, R, \mathcal{O}, Z)$ where:

- $\mathcal{S}$: Set of true state space, where $S_t = (e_d, e_\psi)$ is defined by the displacement magnitude error and the angular error, respectively. The agent will be trained to operate within an interval range $e_{d,t} \in [0, e_{d_{\max}}]$ and $e_{\psi,t} \in [0, e_{\psi_{\max}}]$. This is illustrated in Figure 4.2.
- $\mathcal{A}$: Set of action space of high-level body velocity commands, which is defined by $A_t = (v_{y,t}, \omega_{z,t})$, with $v_{y,t} \in [v_{y_{\min}}, v_{y_{\max}}]$ and $\omega_{z,t} \in [\omega_{\min}, \omega_{\max}]$. The action v_x , will not be included, since it will be fixed, with a value of 0.25 m s^{-1} , which is an adequate velocity for the inspection.
- R : The reward function $\mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, with $\mathcal{S}$ and $\mathcal{A}$ corresponding to the state and action spaces, depends primarily on the current state s_t and action a_t . This is better described in the Equation 4.3, which is explained below.
- $\mathcal{O}$: The observation space, where $O_t = [0, 1]^{H \times W \times C}$ is composed of normalized stacked grayscale pixel images, where C represents the stacked channels and H and W the height and width of the frame, respectively.
- Z : The observation function $Z(O_t) = f(O_t) \in \mathbb{R}^n$ encodes the input an random observation O_t into a n -dimensional latent feature space, approximating the true state S_t .

Figure 4.2: Visualization of the true state errors e_d and e_ψ .

The RL reward function was defined to encourage a reduction in the true state-space errors at instant t ($e_{d,t}$, $e_{\psi,t}$) while increasing each specific rewards $r_{\text{pos},t}$ and $r_{\text{yaw},t}$, respectively. As shown in Equation 4.1, these reward components are computed using a zero-mean Gaussian function with a task-specific standard deviation σ , chosen according to the scale of the corresponding error. The Gaussian output ranges between 0 (minimum) and 1 (maximum).

$$\begin{aligned} r_{\text{pos},t} &= \exp\left(-\frac{e_{d,t}^2}{2\sigma_d^2}\right), \\ r_{\text{yaw},t} &= \exp\left(-\frac{e_{\psi,t}^2}{2\sigma_\psi^2}\right), \end{aligned} \quad (4.1)$$

The smoothness reward component is computed differently. First, the quadratic difference between the current action a_t and the previous action a_{t-1} is calculated; this yields: $\Delta a_t = (a_t - a_{t-1})^2$.

Next, this difference is normalized and inverted over the full action range, producing an error vector $e_{\text{diff_norm}}$ whose elements lie in $[0, 1]$, where 0 indicates an abrupt change and 1 indicates no change in the action. These normalized values are then weighted by a diagonal importance matrix $A = \text{dig}(k_{\text{pos}}, k_{\text{yaw}})$, composed of constants k_{pos} and k_{yaw} satisfying $\sum_i k_i = 1$, and summed to yield the smoothness reward r_{smooth} , as shown in Equation 4.2.

$$\begin{aligned}
\Delta a_t &= (a_t - a_{t-1})^2 \\
e_{\text{diff_norm}} &= 1 - \frac{\Delta a_t}{m_{\text{diff}}} \\
A &= \text{diag}(k_{\text{pos}}, k_{\text{yaw}}) \\
r_{\text{smooth}} &= \sum_{i=1}^n (A_{ii} \cdot (e_{\text{diff_norm}})_i)
\end{aligned} \tag{4.2}$$

Another consideration for the reward function is that if the agent reaches the end of the array within the perfection bounds, it receives a large bonus β . However, if during an episode it reaches the limit of the state space or fails to reach the end within the perfection bounds, it incurs a large penalty λ . These sparse rewards, are triggered by the respective boolean reward signals $r_{\text{goal},t}$ and $r_{\text{boundary},t}$.

Finally, these dense and sparse rewards are combined into an overall reward at instant t : R_t . This is computed using nonnegative weights whose sum is unity to weight the dense rewards, as given in Equation 4.3.

$$\begin{aligned}
R_t &= w_{\text{yaw}} r_{\text{yaw},t} + w_{\text{pos}} r_{\text{pos},t} + w_{\text{smooth}} r_{\text{smooth},t} + \beta r_{\text{goal},t} - \lambda r_{\text{boundary},t} \\
\text{subject to } &\begin{cases} w_{\text{yaw}} + w_{\text{pos}} + w_{\text{smooth}} = 1, \\ w_i \geq 0, \quad \forall i \in \{\text{yaw}, \text{pos}, \text{smooth}\}. \end{cases}
\end{aligned} \tag{4.3}$$

4.2 Training, Validation and Evaluation

To ensure consistent learning across varying conditions, the agent will be trained with domain randomization, as previously mentioned. This makes each episode unique yet similar in a way that the agent can understand the task. Randomization will focus on the following principles:

- Background changes shall be randomly selected from a finite set of realistic planes $\text{rand}(\text{plane}_1, \dots, \text{plane}_n)$ on which the PV panels are mounted.
- Panel layouts will be arranged in three side-by-side arrays, i_0 , i_1 , and i_3 , representing the left, middle, and right arrays, corresponding to all the observation conditions. This provides the agent with three distinct aerial perspectives, depending on which array it navigates, allowing it to sometimes capture the neighboring panels within the camera's FOV.

- Offsets for all the deployed panels will be applied, in both x and y directions, in relation to the base panel positions:

$$([x_{\text{panel}_{\text{base}}}, y_{\text{panel}_{\text{base}}}] + \text{rand}([x_{\text{panel}_{\text{offset}}}, y_{\text{panel}_{\text{offset}}}])).$$

- UAV base positions are also going to be randomized at the beginning of each episode. First, it will be randomly chosen which array the UAV will need to follow, and there will be a base position associated with each array:

$$\text{rand}(i_0, i_1, i_3) \implies ([x_{\text{uav}_i}, y_{\text{uav}_i}, 0]).$$

- Finally, the randomly selected base UAV position will be added to an offset that can change in terms of x, y and ψ (yaw), in a way that:

$$([x_{\text{uav}_i}, y_{\text{uav}_i}, 0]) + \text{rand}([x_{\text{uav}_{\text{offset}}}, y_{\text{uav}_{\text{offset}}}, \psi_{\text{uav}_{\text{offset}}}])).$$

This set of conditions can be represented to an extent in the Figure 4.3. Where both the index that agent will follow, as the UAV pose and panel positions, suffer misalignments, with different possible backgrounds.

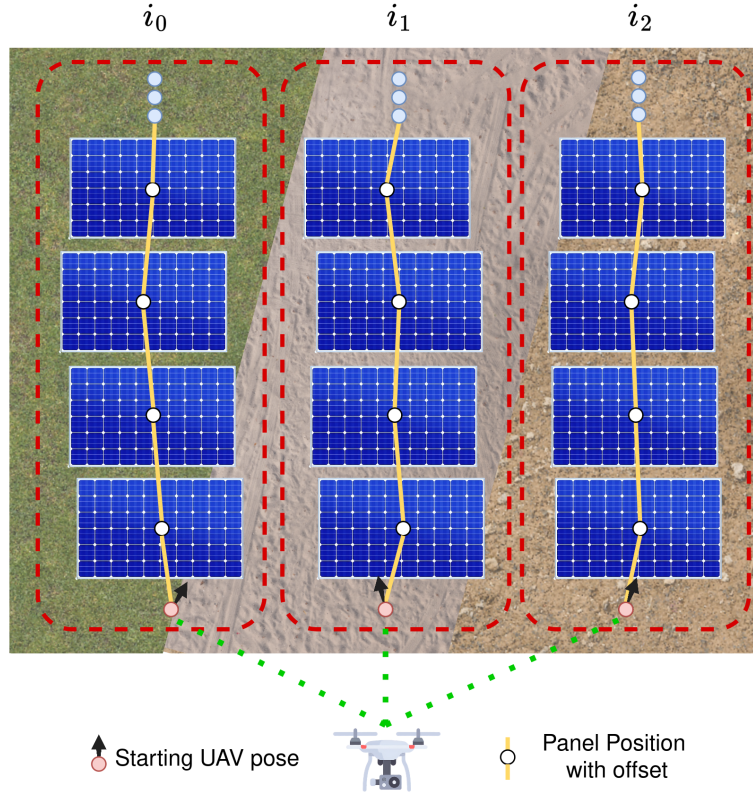


Figure 4.3: Example of training environment for the agent.

The agent will train under these conditions for a maximum of 250 000 n steps. However, early-stopping techniques may be applied if the agent is not learning effectively or if its performance degrades.

To ensure that the best model is always accessible after training, a type of validation process will also be conducted every 10 000 steps. During this process, learning is paused and exploration is disabled to promote greedy actions. In this way, the agent will perform three validation episodes E_v , and the best model, with all the weights and configurations of all the neural networks, will be saved.

In the end, to ensure that the RL agent is ready for production, the best model will go through an exhaustive evaluation phase, in which the agent will operate in unseen scenarios, each consisting of 100 validation episodes E_e . The different test scenarios are as follows:

- **Default test:** A scenario with more PV panels (bigger arrays) than those used during training, to test whether the agent can navigate through PV solar plants of different dimensions.
- **Aligned test:** The PV panels will be perfectly aligned, to test whether the agent can still perform relatively well without misalignments.
- **Different altitudes:** The agent will fly at a finite set of unseen altitudes. In addition to altitude, this scenario indirectly tests its ability to handle different PV panel sizes.
- **Image perturbations:** Perturbations such as brightness, noise, flares, and blobs will be added to the agent's input to verify whether it behaves correctly under visual perturbations that can occur in real-world conditions.
- **Image perturbations + sunlight changing:** Besides the perturbations mentioned before, the sunlight will change at each episode, in a way that it represents the UAV operation at different daytime stages.

After the final evaluation, the agent's strengths and weaknesses will be analyzed and retrained if necessary.

4.3 RL Architectures

Based on the details and summaries presented in Chapters 2 and 3, an RL algorithm suitable for the problem requirements should support continuous actions and include at least some stochasticity in its policy, due to operating in a POMDP. Additionally, a model-free approach is preferable under these conditions and must perform well with visual inputs. With this in mind, the *Soft Actor-Critic* (SAC) was selected as

the starting algorithm since it satisfies all the necessary criteria, as can be observed in the Table 2.1 in the Chapter 2.4.

As previously mentioned, this algorithm processes a set of grayscale visual inputs through a feature extractor, which essentially serves as an observation function Z . Based on the previous chapters, the well-known CNN architecture from Mnih et al. [30] is employed, as it is widely used in end-to-end applications.

However, this CNN will not receive 84×84 images, as is commonly the case. Instead, the visual inputs will maintain the aspect ratio of the visual camera, which for this application will be 4:3. Preserving the aspect ratio is crucial for accurately observing the state, as it provides a more precise perception of the camera's alignment relative to the panel.

Due to the high FPS relative to the UAV movement that a real camera possesses, frame skipping of 4 is applied. Since the image buffer is composed of a queue with 20 frames organized in a *First In, First Out* (FIFO) style, this leaves us only with 5 different frames corresponding to different time instances.

Since the original observations have an RGB configuration and a moderate resolution of 1080×1440 , they are converted to grayscale and resized to 96×76 . This resolution is similar in size to the default 84×84 , while maintaining the aspect ratio. Additionally, a slight blur with a (5,5) kernel is applied after resizing to introduce a bias in the CNN network, encouraging it to learn image patterns rather than textures, which ideally promotes better generalization.

The CNN block consists of three convolutional layers:

- Conv1: 32 filters, 8×8 kernel, stride 4
- Conv2: 64 filters, 4×4 kernel, stride 2;
- Conv3: 64 filters, 3×3 kernel, stride 1.

Each layer is followed by a ReLU activation function. After the last ReLU, the output is flattened into a vector, which is then fed into the projection block, consisting of a single linear layer with an additional ReLU activation, that projects the high-dimensional input into a 512-dimensional embedding.

These observations can be fed directly to the image feature extractor component and then into the remaining SAC algorithm. Each observation is also stored in the replay buffer, which will be available for mini-batch processing of size N . Different feature extractor models with the same architecture will be used for the actor and critic networks, since this promotes a better learning performance.

Following this logic, the overall architecture is illustrated in Figure 4.4. Note that the SAC block is represented as a “black box” and it is overly simplified.

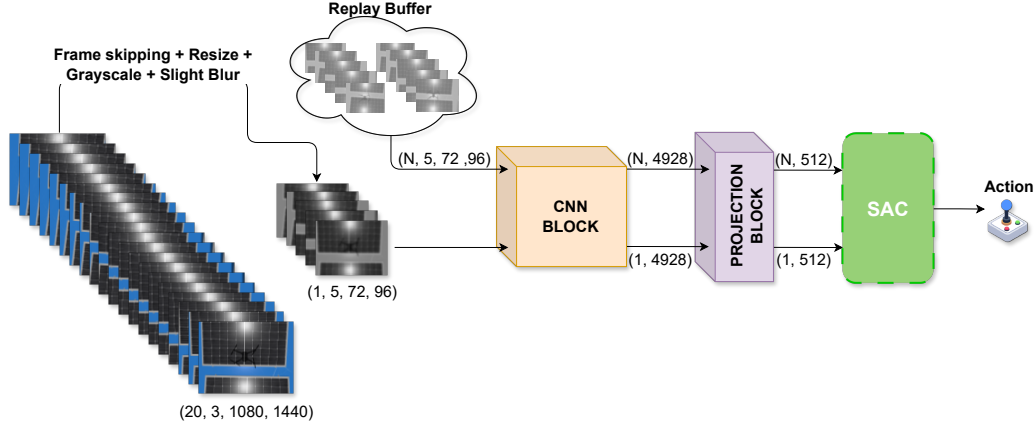


Figure 4.4: Default Feature extractor architecture for the SAC algorithm.

Another architecture will be implemented and compared against the baseline architecture, which only includes the image component. This new architecture adds an additional slight memory mechanism, also based on the concept of stacking information, but for the actions. In theory, as studied in Chapters 2 and 3, this can be fundamental for improving agent performance in POMDP formulation. Specifically, this architecture can receive the past 10 actions ($a_t, \dots, a_{t-9}$) at a specific control rate, such as 10 Hz. This corresponds to approximately 1 second of past information. All of this information is also stored in a replay buffer, which can also be processed by the SAC algorithm, using a mini-batch of size N .

In this paradigm, the actions tensor is ingested into the *Multi-Layer Perceptron* (MLP) block, where the last two dimensions are flattened, converting it from $(N, 10, 2) \implies (N, 10 \cdot 2)$. The resulting tensor is then passed through the actual MLP, which consists of one hidden layer with 128 neurons and an output layer with 64 neurons, both followed by a ReLU activation function. The embedding obtained from this network is concatenated with the embedding from the image projection block, resulting in a final embedding of size $(N, 576)$.

This alternative architecture approach can be depicted in Figure 4.5. In this same architecture, the input actions to the MLP can be easily replaced with other agent information, allowing for the testing of new possibilities.

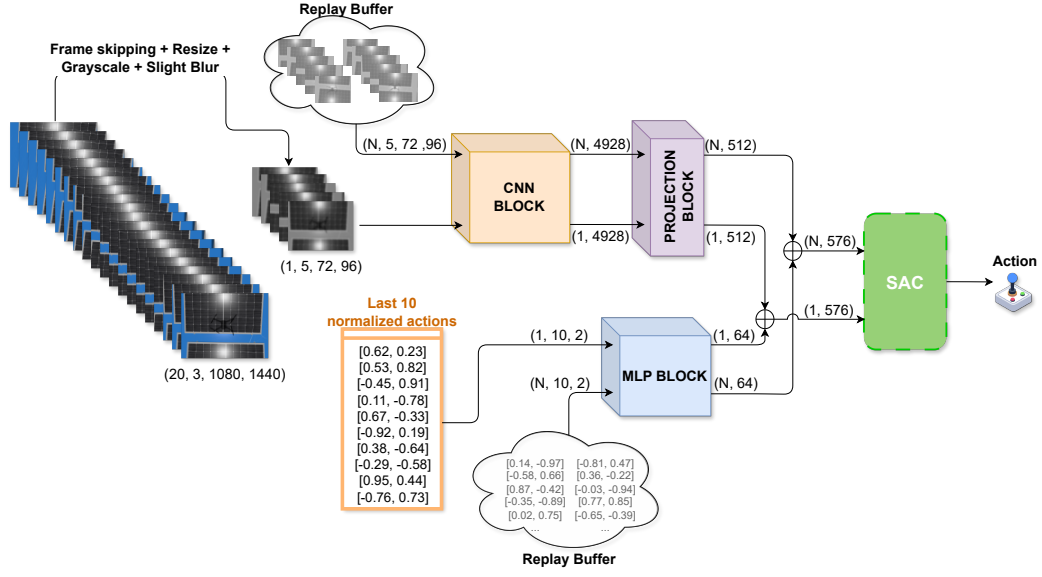


Figure 4.5: Feature extractor architecture with normalized actions for the SAC algorithm.

4.4 Summary

After defining the problem correctly, including the reward function, the training, validation, and evaluation conditions, two main architectures were considered for designing the solution.

With this two architectures, at least three models will be implemented: the default with only visual input using the architecture in 4.4 and another two models using the architecture in 4.5, where one will include the stacked actions a_{v_y} and a_{ω_z} as input, and another version will also be implemented, where these actions are substituted by the actual agent's body velocity v_y and w_z . Ultimately, the most suitable model will be selected.

In the next Chapter 5, the implementation details will be presented, and the results will be analyzed in Chapter 5.

Chapter 5

Development of the Solution

This chapter documents the implementation of the proposed solution, including the rationale behind tool selection, the configuration and tuning of the selected tools, and the design and development of the data processing, training, and deployment pipeline.

5.1 Simulation Environment

The first step in developing the solution is to create a realistic environment that supports domain randomization and is compatible with the stack and tools used in the real-world application.

Autonomous UAV real-world operations typically rely on ArduPilot [90] or the PX4 [91] Autopilot firmware. As the hosting research group currently employs ArduPilot, the training environment will support both ArduPilot and the ROS 2 middleware [92]. The environment refers to the simulated world. Given the nature of the reinforcement learning task, the model requires a continuous stream of high-fidelity images as observations to generate actions, forming a cause-and-effect loop.

When it comes to choosing a simulation platform, the options are limited, with Isaac Sim [93], AirSim [94], and Gazebo Sim [95] being the top candidates. Each has its advantages and disadvantages.

The official Isaac Sim does not possess native compatibility with the ArduPilot firmware, besides having some non-official developments that work on top of this

framework, such as the Pegasus Simulator [96]. It enables the creation of hyperrealistic worlds due to the use of Open *Universal Scene Description* (USD) technology. AirSim, despite supporting the Unreal Engine and Unity game engines, has been discontinued by Microsoft, becoming outdated for modern workflows that require *Robot Operating System* (ROS) 2 and other updated dependencies. As of the date of this writing, Gazebo Sim remains the simulator with the highest compatibility with the technical stack used in the research group. However, the old Gazebo Classic is often perceived by the community as a non-realistic simulation platform. The modern Gazebo Sim has considerably improved its rendering capabilities, supporting *Physically Based Rendering* (PBR) textures and *Digital Asset Exchange* (DAE) models¹. Gazebo’s main limitation remains its lack of parallelization, a critical factor in the development of RL applications. With this in mind, and given prior experience and current requirements, the modern Gazebo simulator was selected as the simulation environment.

5.2 Gazebo Configuration

To ensure that the Gazebo simulator provides a realistic simulation, it is necessary to use DAE models and PBR textures as previously mentioned. The first step was to find a realistic PV solar panel model that could be easily imported into Gazebo. However, most realistic solar panel models available online are not provided in the DAE format, necessitating file conversion. The first selected model, shown in Figure 5.1, was downloaded from the Epic Games online store [98]. The 3D files downloaded from the store were converted to DAE format with the Blender software [97].

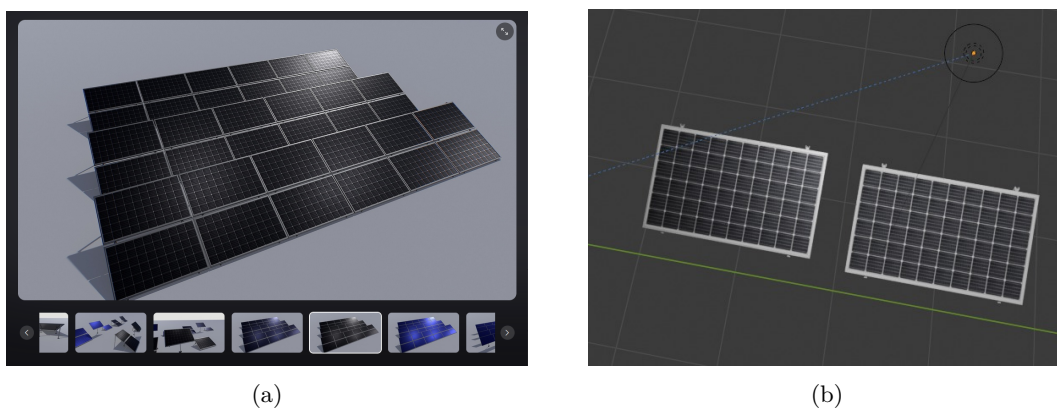


Figure 5.1: Solar panel model in the online store (a) and in Blender after configuration (b).

¹DAE files are actually COLLADA files, natively supported by Blender, an open-source 3D graphics tool [97]

In addition to adjusting the metallic properties of the panel to enhance reflection, the main texture was modified to include a vertical white line. This detail, commonly found in real-world PV solar panels, typically indicates the separation between two different modules within the same panel.

Finally, after being exported to the DAE format using Blender, the model was deployed in Gazebo. As expected, it retained the properties defined in Blender and exhibited a realistic appearance, as shown in Figure 5.2.

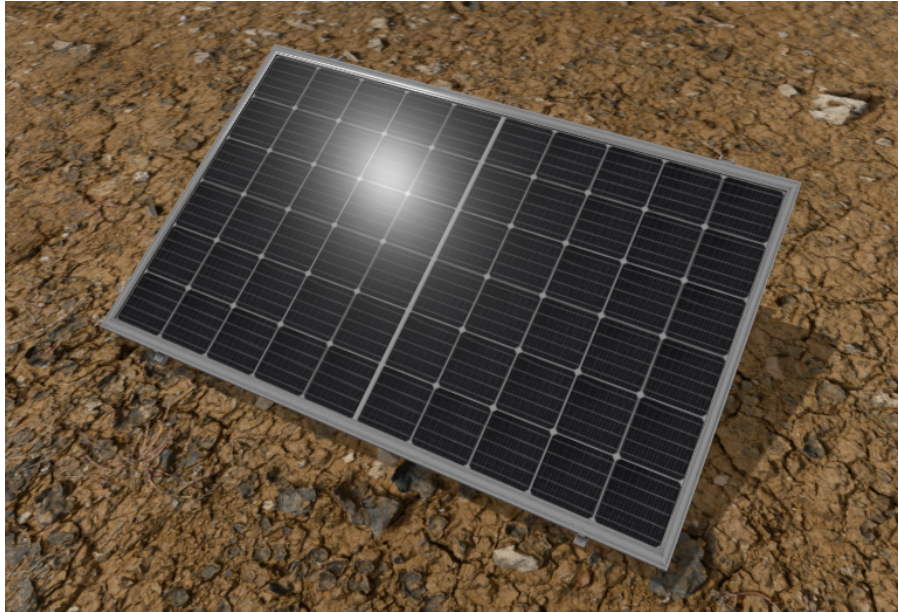


Figure 5.2: Final panel model used in the simulation.

With the PV model defined, it was necessary to implement some realistic textures for the background, since the textures present in the default Gazebo world models are basic in detail. In this way, due to the support of PBR textures on the new Gazebo Harmonic, these textures were downloaded from open-source online fonts such as the Poligon website [99].

The textures chosen are shown in Figure 5.3, which will be useful for the domain randomization scenario, ensuring that the RL agent does not overfit across different types of backgrounds.

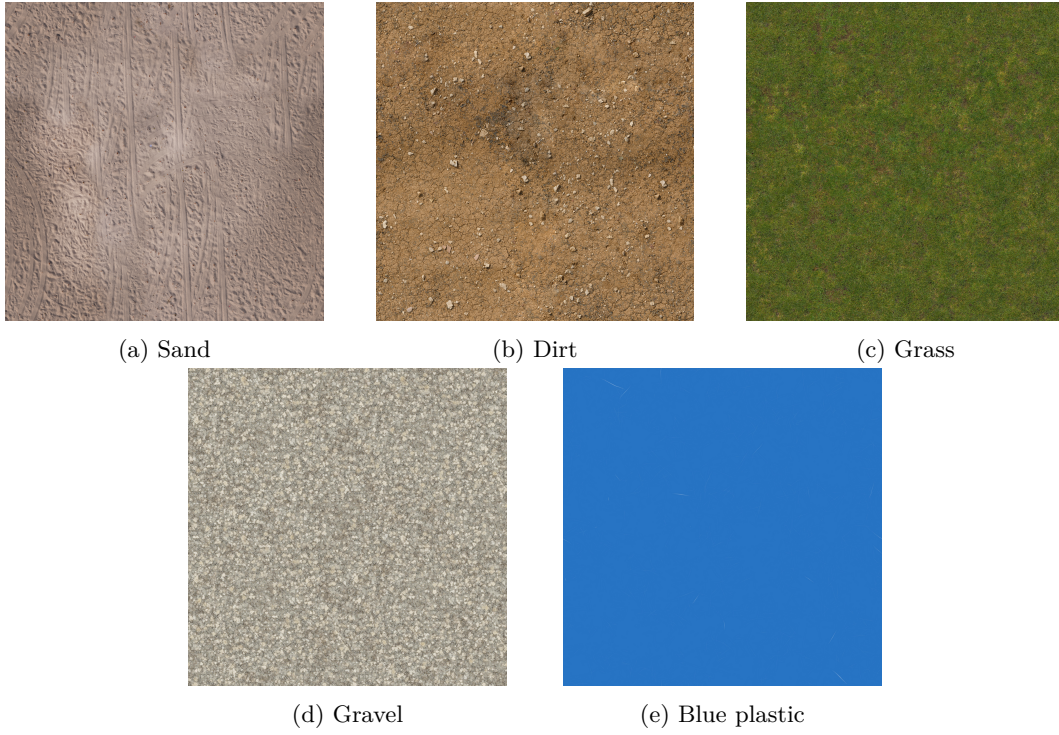
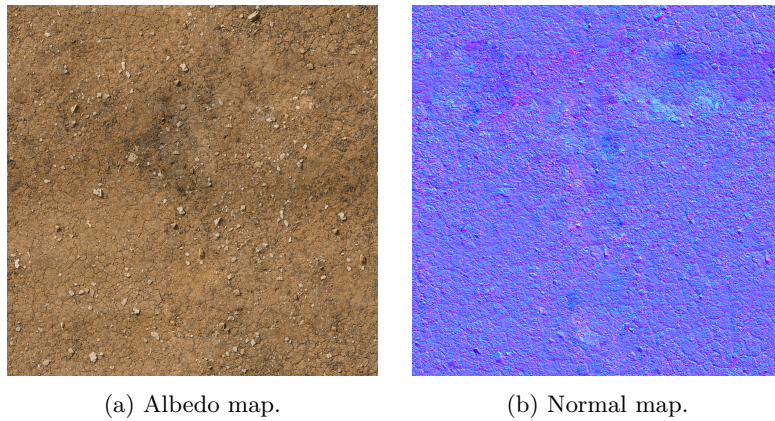


Figure 5.3: Different types of textures used in the simulation.

PBR textures consist of multiple channels, each encoding specific material properties. When combined, these channels reproduce realistic surface appearances under different lighting conditions, resulting in a highly detailed and visually accurate representation of the material. Since Gazebo uses the Ogre2 engine [100] and *Simulation Description Format* (SDF) files for model specification, it supports only the basic texture layers [101]. So, in this work, only the albedo (color map), normal map, roughness map, and ambient occlusion were used, as demonstrated in Figure 5.4 for the example of a grass background.



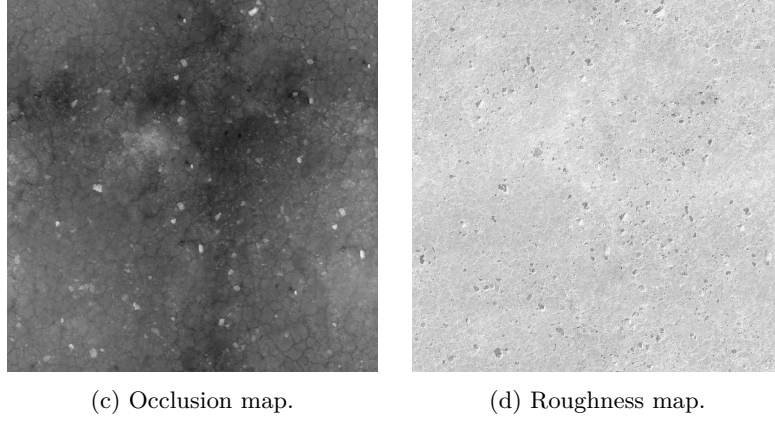


Figure 5.4: Different texture layers used in the simulation.

The UAV used in the simulation was the standard Iris quadcopter equipped with a gimbal-camera setup. To train the RL agent under realistic conditions, the gimbal option was selected to replicate real-world behavior, including the compensation movements performed by the gimbal during operation.

The camera’s intrinsic parameters present in one of the MOTUS’s UAV, represented by the matrix K in Equation 5.1, were used in the simulation with a 4:3 aspect ratio and a resolution of 1440×1080 pixels. The refresh rate was set at 30 Hz, providing the closest possible representation of the real-world UAV camera within the simulation.

$$K = \begin{bmatrix} 1779.3015 & 0 & 705.5547 \\ 0 & 1779.6183 & 505.0019 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.1)$$

With all of these Gazebo configurations, it becomes possible to create a world that supports domain randomization, as discussed in Chapter 4.2. The panels were slightly tilted to match the configuration of the Alqueva reservoir. A water background was not included, as the UAV does not observe the water surface while flying over the panels. In the Alqueva reservoir, a layer of blue plastic is placed beneath the panels. Consequently, the blue plastic background, depicted in Figure 5.3, was incorporated instead.

The remaining backgrounds were used primarily to promote the learning of a robust policy under different conditions. Figure 5.5 shows an example of the simulation with all these configurations combined, but with the sandy background.

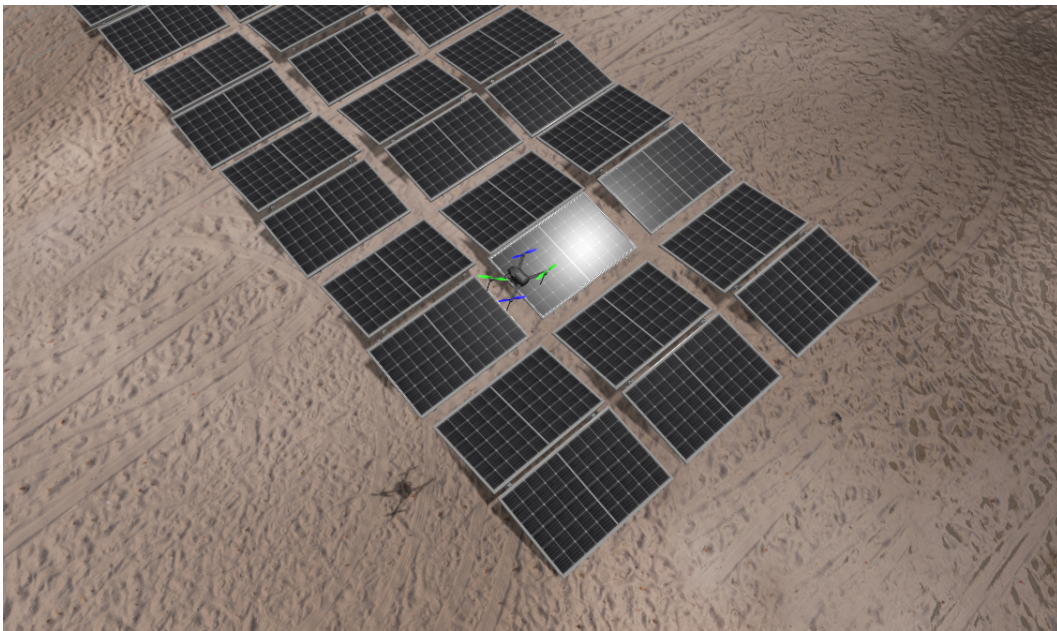


Figure 5.5: Final appearance of the simulation world with sandy background.

5.3 RL Agent Integration

With the Gazebo simulation correctly configured, it was necessary to integrate the reinforcement learning agent. As outlined in Chapter 4.2, this involved setting up training and evaluation pipelines supporting domain randomization. Concurrently, a feature extractor capable of processing visual inputs was implemented and adapted, as described in Chapter 4.3.

Due to the high complexity of the RL algorithms, combined with the highly customized setup defined in this report, a reliable open-source framework is almost a necessity for working efficiently. With this in mind, and since the chosen agent was the SAC algorithm, the *Stable-Baselines3* (SB3) framework [102] was selected, as it provides a reliable implementation of the SAC agent [103]. It allows for a customization of the training pipelines, while following and recommending the best implementation practices that these RL agents require to learn.

This framework also allows for straightforward customization of the feature extractor network, as well as of the actor-critic networks, if needed.

After reviewing the documentation, it was concluded that for the initial implementation of the Mnih et al. [30] CNN, represented in the Figure 4.4, it was necessary to use a `CNNPolicy` class that is defined in the SB3 framework [104]. In this configuration, it incorporates the desired feature extractor architecture by default, as shown in Appendix A.1.

For the combination of the agent’s velocities/actions with the visual input, which corresponds to the design detailed in the Figure 4.5, it was necessary to use a

`MultiInputPolicy` class [104]. The code corresponding to the design of this feature extractor can be found in the Appendix A.2.

Since the actor and critic optimize different objectives, it is sensible to use distinct feature representations for each network. In this way, the actor and critic have separate architectures and weights, which can improve learning stability and overall performance [104].

The main architecture of SAC, for both the actor and critic networks, as detailed in Chapter 2.3, was kept unchanged, preserving the original SB3 implementation and remaining faithful to the design proposed in the original paper [39].

All the training and evaluation were developed in a way that could be easily configured and manipulated. This was achieved by defining a *YAML Ain't Markup Language* (YAML)² file that specifies the overall structure of the training. This file is divided into two main categories: the first corresponds to the training configuration, and the second to the environment configuration. An example of how this file is defined can be found in Appendix A.4. These different types of configurations are also illustrated in Figure 5.6, which shows how the SB3 main loop parses this information and orchestrates the rollout training, which in the Equation 5.10 is explained in detail.

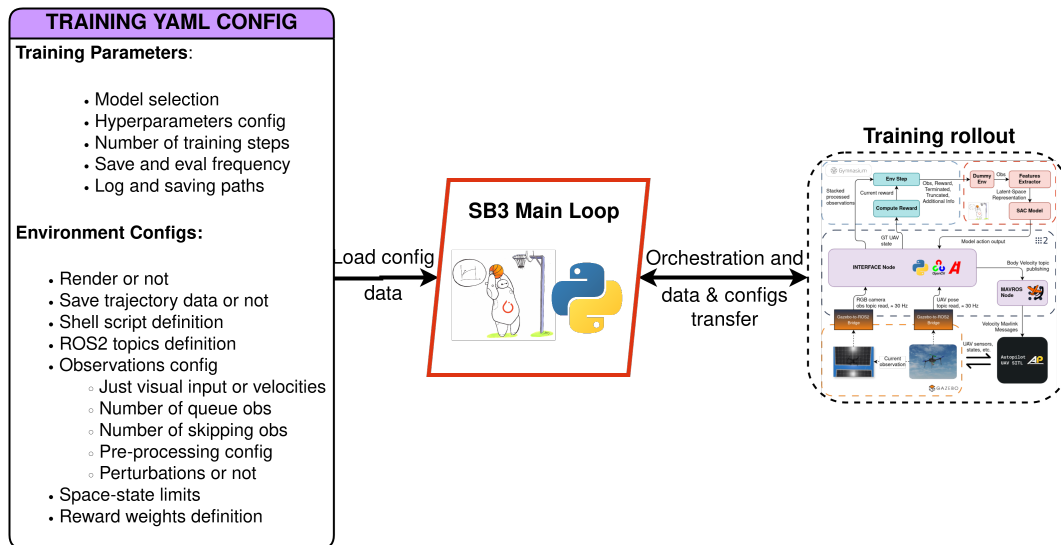


Figure 5.6: RL training configuration and orchestration.

To work with SB3, it is necessary to use a Gymnasium (Gym) environment [105]. Due to the use of Gazebo, a custom Gym environment becomes essential to develop the necessary pipelines [106]. The main functions of a custom Gym environment are the `step()` and `reset()` functions, where both are defined in Python.

The `step()` function is responsible for updating the environment with actions and returning the next agent observation, the reward for taking those specific actions,

²The developers defined this funny recursive acronym on purpose.

and information on whether the episode is terminated (reached the end or state-space limits) or truncated (if the agent reached the time-step limit for a specific episode).

The reward function called in this function follows Equation 4.3, defined in Chapter 4.1, and its application can be found in the Appendix A.7. It receives the *Ground Truth* (GT) UAV state-error to compute its values. The weights are normalized with the following characteristics: $w_{\text{yaw}} = 0.35$, $w_{\text{pos}} = 0.40$, $w_{\text{smooth}} = 0.15$. As a reminder from Equation 4.1, Gaussian functions are used to compute the position and yaw rewards. Each has its own standard deviation, with $\sigma_{\text{yaw}} = 0.10$ and $\sigma_{\text{pos}} = 0.25$, which directly relate to the bounds accepted in the simulation. The state-error bounds were set to $e_{d_{\text{max}}} = 1.0$ m and $e_{\psi_{\text{max}}} = 25^\circ$.

The smoothness action component, as defined in Equation 4.2, lacks a normalisation function. Nevertheless, it is assigned an action importance value, which in this instance is evenly distributed across the position and yaw control actions $k_{\text{pos}} = k_{\text{yaw}} = 0.5$. The minimum and maximum action values were $[-0.5, 0.5]$ for v_y and $[-0.3, 0.3]$ for ω_z .

Other useful information, such as metrics or debugging data, can also be returned from this *step()* function. This function plays a key role in the training process, as illustrated in Figure 5.10, and its integration with the overall pipeline is explained later.

On the other hand, the *reset()* function, as the name suggests, resets the environment to its initial state. It must be called before the first step. This function returns only the first agent observation, along with any other necessary information.

In practice, the *reset()* function is also responsible for randomizing the domain of the environment. It triggers a set of shell scripts that clean and restart the environment, along with a Gazebo launch file that randomizes the predefined possibilities specified in another YAML file, as defined in the Appendix A.5. After the randomization, this information is stored in a temporary *JavaScript Object Notation* (JSON) file, which is then used for logging and for computing the reward function, based on GT. This process is visually described in the Figure 5.7. A simplified code snippet of the *step()* and *reset()* functions is also present in the Appendix A.8.

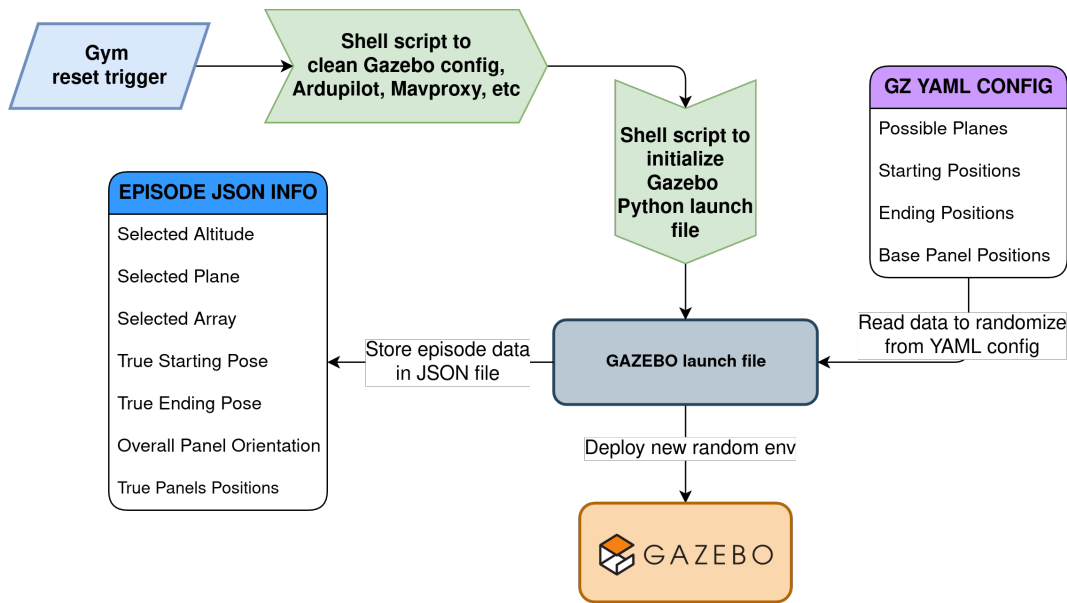


Figure 5.7: Implementation of the domain randomization at each reset step.

Once the environment is set up, it is important to ensure that the UAV is ready for the learning process of the new episode. To achieve this, a ROS 2 node was developed to guarantee that the UAV performs takeoff correctly. The challenge arises because, when using the Ardupilot *Software in the Loop* (SITL), it usually takes a long time before all the state estimators – such as the *Extended Kalman Filter* (EKF), which relies on the simulated GNSS – are ready for operation. Initially, the GNSS was disabled, but this caused the UAV to become unstable and drift over time.

The proposed solution was to speed up the simulation by adjusting the Gazebo RTF parameter and the Ardupilot `SPEED_UP` parameter to the maximum values supported by the computer. After the takeoff and the gimbal configuration, these parameters were reset to their default real-time values, allowing the agent to experience learning at a real-time pace.

To apply these configurations, the MAVROS node, that acts as a bridge between the MAVLink communication and the ROS 2 middleware [107], was integrated. MAVROS is a valuable tool available for the UAVs present in the lab and for the problem in question. In a nutshell, this setup provides the following ROS 2 topics and services:

- `/mavros/state`: A topic that provides the Autopilot state, such as the mode and the armed status, by reusing the `mavros_msgs/State.msg` message.
- `/odometry`: A topic that publishes the altitude by reusing the `nav_msgs/msg/Odometry` message.

- `/uav_ready`: A topic that informs that the UAV is ready by reusing the `std_msgs/msg/Bool` message.
- `/mavros/cmd/arming`: A service to request arming with an interface of type `mavros_msgs/CommandBool.srv`.
- `/mavros/set_mode`: A service to request mode change with an interface of type `mavros_msgs/SetMode.srv`.
- `/mavros/cmd/takeoff`: A service to request a takeoff with an interface of type `mavros_msgs/CommandTOL.srv`.

This node also employs a Python library called Pymavlink [108], which allows for direct communication with the simulated *Flight Control Unit* (FCU). This was needed to ensure a more precise and direct control of the gimbal. It opens a master connection and uses a Gimbal V2 protocol [109], with the following message IDs to point down the camera correctly, independent of its starting position:

- `GIMBAL_MANAGER_INFORMATION`
- `GIMBAL_MANAGER_CONFIGURE`
- `GIMBAL_MANAGER_PITCHYAW`

This master connection is then reused to change the `SIM_SPEEDUP` parameter, after which it is properly closed. Following that, a temporary Gazebo node is created to communicate with the `/world/map/set_physics` gz topic, ensuring that Gazebo runs the *Real-Time Factor* (RTF) at the intended speed.

This node is shown in Figure 5.8, essentially representing a monotonic state machine. At the end, the topic `/uav_ready` signals that this task was completed successfully.

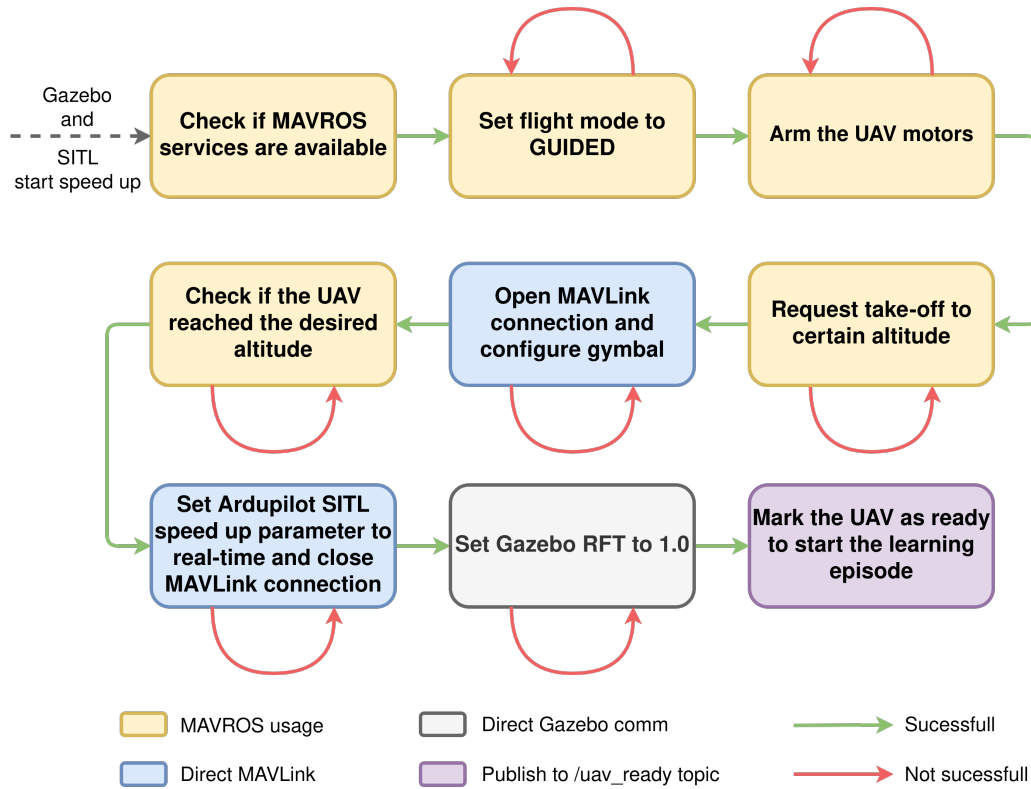


Figure 5.8: UAV preparation before the learning process.

To handle the data processing between the Gym/SB3 and the Gazebo with ROS2 topics, an interface node was developed. This node is essential for the training rollout and processes differently depending on the given configuration on the YAML, detailed in the Figure 5.6.

Its main role is to stack and pre-process the observations of the agent, before they are fed into the feature extractor networks, as can be observed in the Chapter 4.3 in the Figures 4.4 and 4.5. It works directly with the following ROS 2 topics:

- `/camera/image`: Reads images of type `sensor_msgs/msg/Image`
- `/odometry`: Reads the pose of type `nav_msgs/msg/Odometry`
- `/mavros/setpoint_raw/local`: Publishes the velocities output by the model with message type `mavros_msgs/msg/PositionTarget`
- `/uav_ready`: Reads the topic by the previously explained node, indicating that is ready to start the learning. Message of type `(std_msgs/msg/Bool)`.

An example of these topics working together with the ROS 2 interface node is displayed in Appendix A.1.

It uses a resizable queue to store the current observations. The observations/images, after being obtained through the topic, are processed using OpenCV, where

they are turned into grayscale, resized to the size of [96, 72], maintaining the 4:3 aspect ratio, and a small blur of the kernel [5, 5] is applied to introduce some bias to the CNN feature extractor. All these values can also be easily configured, as explained in Figure 5.6.

The computed GT errors are sent to the Gym *step()* function to calculate the step reward. For large evaluation sessions, the node additionally supports the application of image perturbations using the Albumentations library [110], well known for image augmentation. This library provides a wide variety of image effects, which can be easily configured to be applied randomly. For the problem in question, the selectable effects were the following:

- **RandomShadow** (probability $p = 0.5$)
 - Region of interest: [0.2, 0.2, 0.8, 0.8]
 - Number of shadows: 1-2
 - Shadow dimension: 4
 - Shadow intensity range: 0.2-0.6
- **RandomSunFlare** (probability $p = 0.5$)
 - Flare ROI: [0, 0, 1, 1]
 - Source radius: 20
 - Source color: [255, 255, 255]
 - Angle range: 0-1 (normalized)
 - Number of flare circles: 1-3
 - Method: physics-based
- **ISONoise** (probability $p = 0.5$)
 - Color shift range: 0.01-0.05
 - Intensity range: 0.1-0.3
- **Illumination** (probability $p = 0.5$)
 - Mode: linear
 - Intensity range: 0.1-0.15
 - Effect type: both (brighten and darken)
 - Angle range: 0-360°
 - Center range: 0.1-0.9
 - Sigma range: 0.5-1

An additional effect can be applied via this interface node, enabling the generation of small random globs using OpenCV's polygon function. Combined with the previous library, this produces the visual result shown in Figure 5.9.

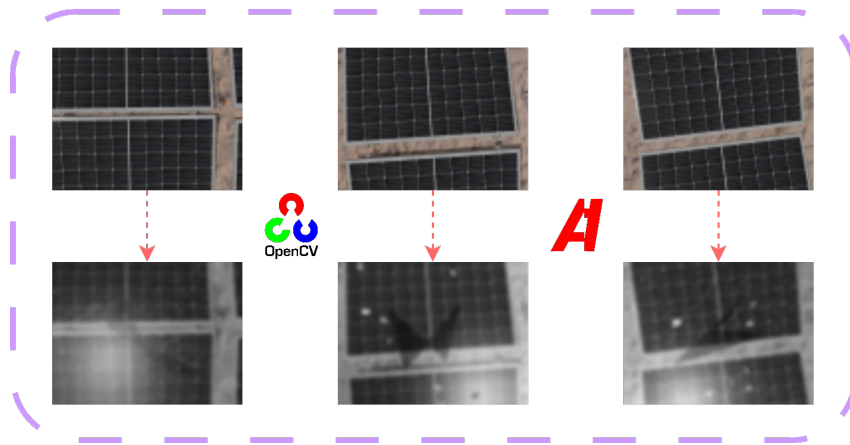


Figure 5.9: Image perturbation example using OpenCV and Albumentations function.

The interface node is also responsible for translating the model action output into a valid body message for the UAV, which is then received by the MAVROS node and converted into a valid MAVLink message. The UAV then acts on this command and a new observation arises, creating an iterative RL loop.

Figure 5.10 presents a high-level architecture of the training rollout, summarizing much of the information described earlier.

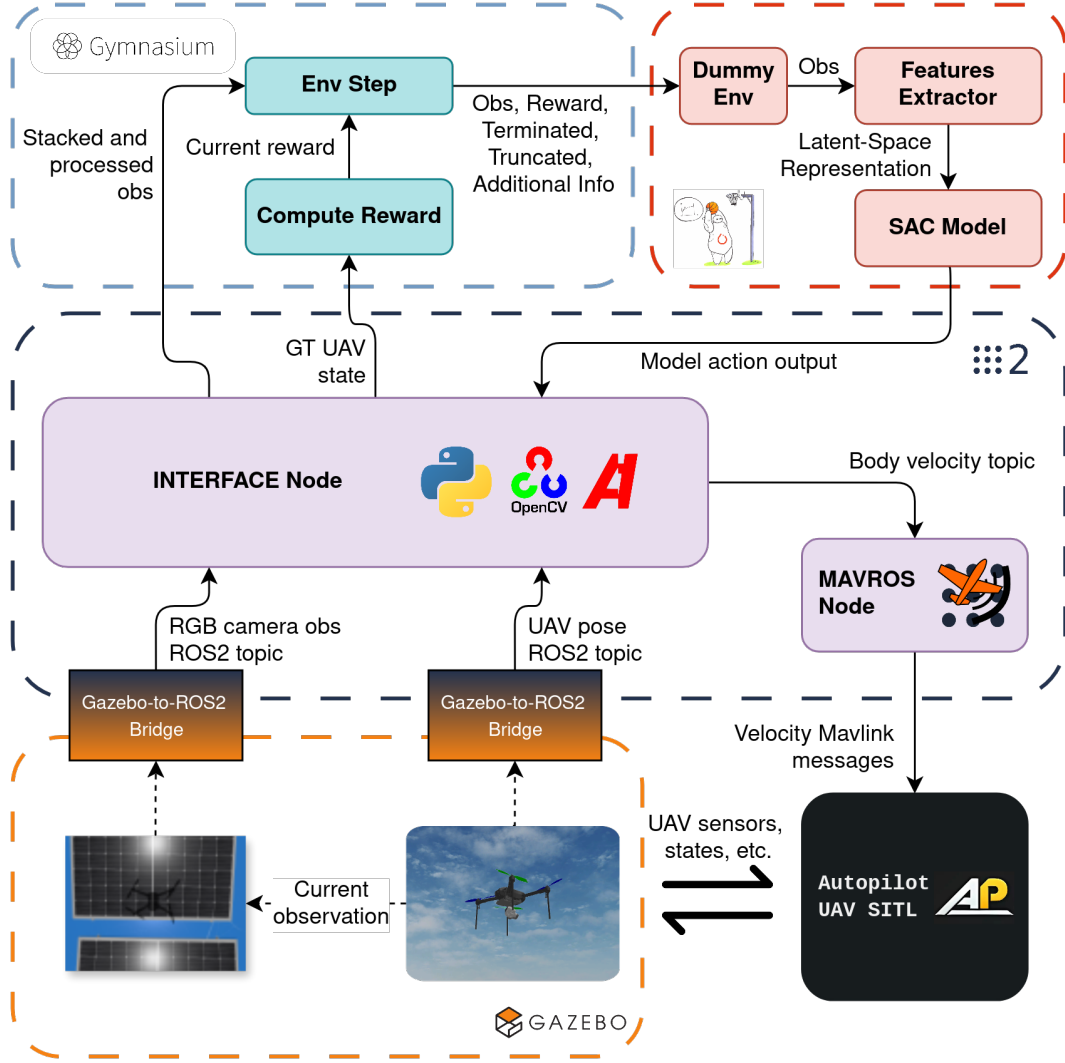


Figure 5.10: High-level overview of the training rollout pipeline.

5.4 Summary

This chapter detailed the implementation of the proposed solution, outlining the tools, configurations, and pipeline design used to train and deploy the RL agent. The first stage focused on building a realistic simulation environment. After evaluating available platforms, the modern Gazebo simulator was selected due to its compatibility with ArduPilot and ROS 2, and its support for domain randomization. Realistic solar panel models and high-fidelity PBR textures were integrated to enhance visual realism and prevent overfitting. The UAV model, camera parameters, and environmental backgrounds were configured to closely match real-world conditions.

The second stage addressed the integration of the reinforcement learning agent using the SAC algorithm from SB3. Custom Gym environments were created to

bridge Gazebo with the training loop, defining *step()* and *reset()* functions that incorporate reward shaping, domain randomization, and environment resets. Additional ROS 2 nodes were developed to manage UAV initialization, gimbal configuration, and communication through MAVROS and Pymavlink.

Finally, an interface node was implemented to preprocess visual observations, apply configurable perturbations for robustness, and convert model outputs into UAV commands. These components collectively form a robust data-processing and training pipeline that supports domain randomization and enables the deployment of stable, generalizable reinforcement learning policies. Chapter 6 presents the results obtained using this implementation.

Chapter 6

Tests and Results

In this section, the results of the predefined tests, using the architectures and details described in Chapter 4 and the implementation specifications in Chapter 5, are presented and analyzed. Based on the initial conclusions, it is also inferred whether any changes or additional tests are needed to achieve the expected outcomes of the thesis.

6.1 Experimental Configuration

Before presenting the experimental results, it is relevant to first provide an overview of the computational platform on which the tests were executed. All experiments were carried out on a standard home desktop computer, whose hardware and software specifications are summarized in Table 6.1.

This information is included to ensure the reproducibility of the experiments and to clarify the computational resources required to run the proposed methodology.

Table 6.1: Hardware and software specification.

Component	Specification
CPU	Intel Core i7-14700 K
RAM	32 GB
GPU	NVIDIA RTX 4060 Ti
VRAM	16 GB
OS	Ubuntu 22.04
Python	3.10.12
ROS	ROS 2 Humble
Gazebo	Harmonic
Arducopter	4.7.0-dev
StableBaselines3	2.6.0
Pytorch	2.5.1
OpenCV	4.10.0
Albumentations	2.0.8

For simplicity, the SAC agent will be trained using the default hyperparameter configuration shown in Table 6.1, providing an effective baseline for comparing different architectures.

Learning begins only after 2000 environment steps, allowing the replay buffer B to accumulate sufficient data before training starts. The mini-batch size and learning rate have the recommended values.

The entropy coefficient α was initialized to 1.0 and is automatically tuned during training (set as "auto_1.0" in the implementation) to achieve the desired target entropy H_{target} . The value $-\dim(\mathcal{A})$ is the SAC heuristic for the target entropy and, for this application, equals -2 . Both the optimizer gradient updates ($G = 1$) and the Polyak target updates ($T_{\text{target}} = 1$) are applied at each training update. Since the f_{train} , these updates happen at each environment step.

Table 6.2: Default SAC hyperparameters used.

Parameter	Symbol	Value
Learning rate	lr	3×10^{-4}
Replay buffer size	B	10^5
Learning starts	N_{start}	2000
Mini-batch size	N	256
Soft update coefficient	τ	0.01
Discount factor	γ	0.99
Train frequency	f_{train}	1
Gradient steps	G	1
Initial entropy coefficient	α	1.0
Target entropy	H_{target}	$-\dim(\mathcal{A})$
Target update interval	T_{target}	1

6.2 Model Training and Comparison

In this section, the training of three different models will be compared, based on two main architectures: the default and the custom architectures mentioned in 4.3. The models will be such:

- **Model 1:** Default RL architecture, with only stacked images as input.
- **Model 2:** Custom RL architecture, with stacked images and real stacked velocities as input.
- **Model 3:** Custom RL architecture, with stacked images and real stacked past actions as input.

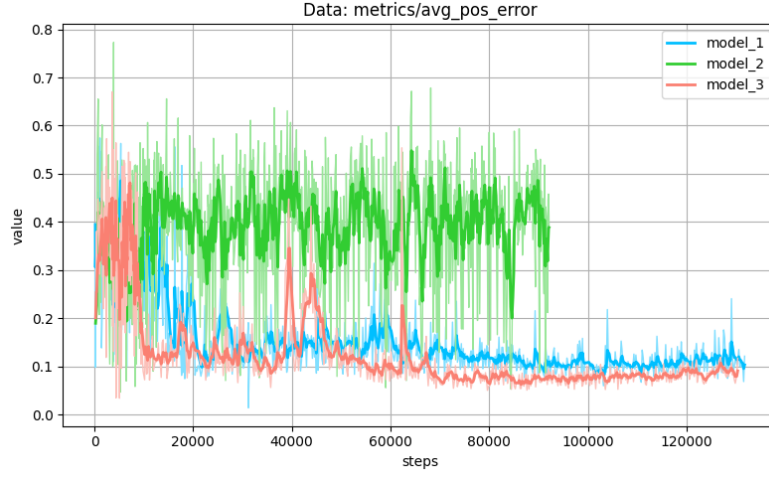
Each model had a different training time, depending on the number of steps. Model 1 took 622.8 min (approximately 10.38 h), similar to Model 3, which took 612.6 min (10.21 h). Model 2 had a shorter training time of 532.08 min (8.87 h) due to early stopping.

The following figures present the data explored by each model using a soft color for the real values and a darker color for the smoothed values to facilitate interpretation.

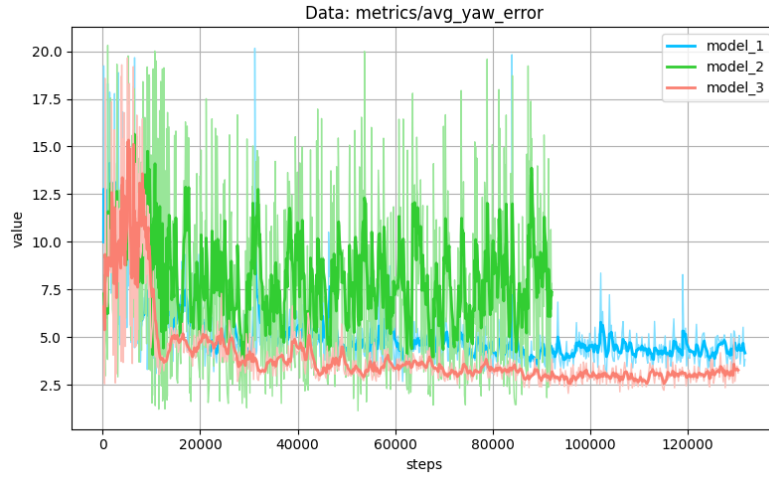
6.2.1 Overall Data Analysis

In Figure 6.1, the metrics related to average position error and yaw error are shown. Model 3 achieved the best results during training, converging to an average position error of approximately 0.08 m and an average yaw error of 3.0° , slightly outperforming Model 1, which reached around 0.11 m and 4.3° for positional and orientation errors, respectively. Model 2, although based on the same architecture as Model 3, failed to learn properly and produced highly noisy metrics, resulting in higher average errors, which is why it was stopped early.

Since the architecture of Model 2 and Model 3 is the same, it can be inferred that when the real past velocities are introduced into the custom RL architecture, the model struggles to capture the underlying dynamics of the UAV. This is reasonable, as real velocities can sometimes be noisy and exhibit complex temporal relationships that the feature extractor cannot accurately represent.



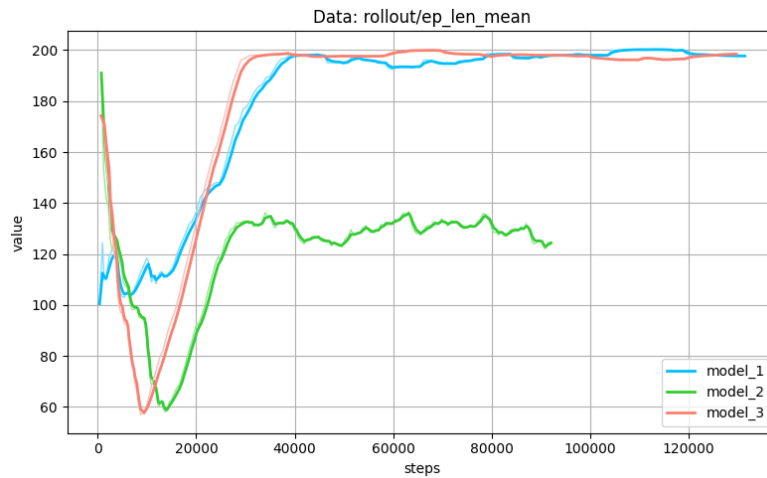
(a)



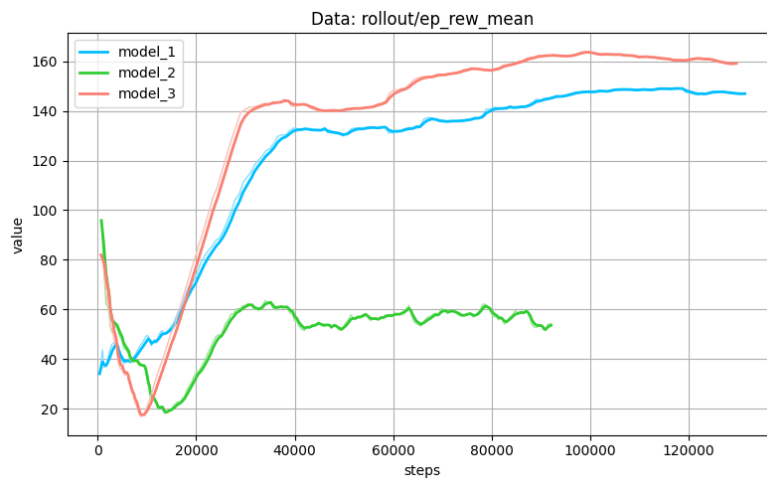
(b)

Figure 6.1: Average position metrics in (a) and yaw in (b) of the three models.

In Figure 6.2, both the smoothed average episode length and the smoothed average episode reward are shown. Model 2 did not converge to a higher episode length or reward compared to the other approaches. The reduced episode length indicates that the agent often went out of bounds and was unable to complete the episode. Although Model 3 achieved an episode length of around 198 steps, which is identical to that of Model 1, it produced a higher reward accumulation of approximately 162, compared to the converged reward of approximately 148 of Model 1. This improvement indicates that the agent is able to navigate with fewer errors, as reflected in the previous metrics shown in Figure 6.1.



(a)



(b)

Figure 6.2: Smoothed episode length in (a) and smoothed episode reward in (b).

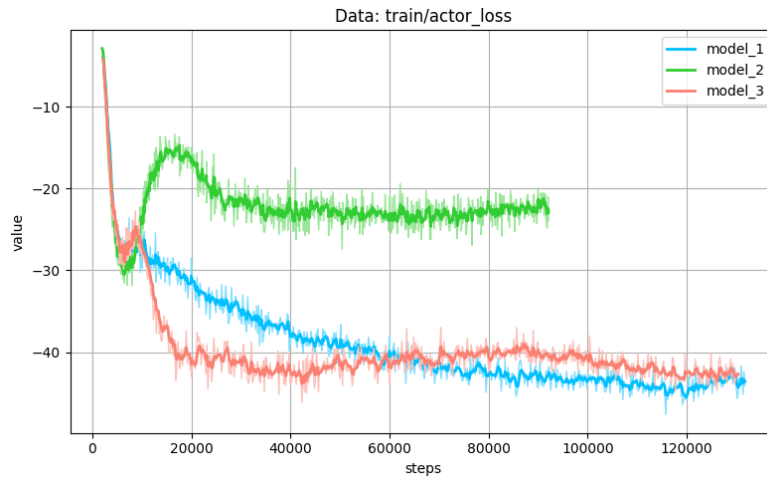
The corresponding critic loss, actor loss, entropy loss, and the evolution of the entropy coefficient are shown in Figure 6.3. Actor loss in actor-critic methods is a surrogate training objective that measures how the policy parameters are being updated according to the critic's estimates; it is not the same as the task reward. Because the surrogate is estimated from finite (and noisy) batches of experience, changes in actor loss reflect the dynamics of learning (how aggressively the policy is changing) rather than direct task performance [111].

All variants reduce actor loss during training, which indicates learning activity. The Model 3 curve converges earlier and is visually much smoother than the other variants, which suggests the policy updates are more consistent and less prone to

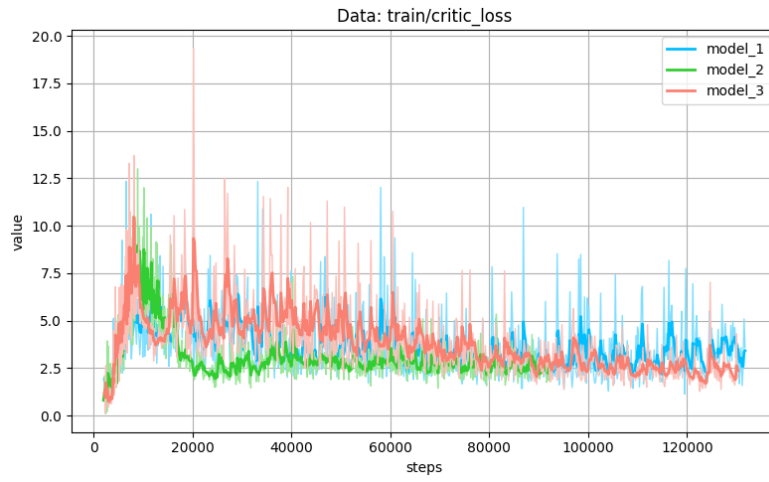
large, noisy swings. The Model 1 variant reaches slightly lower surrogate loss values but at a longer stage on the learning episode; this does not necessarily imply it attains better task performance. The Model 2 variant stabilizes at a substantially higher actor surrogate loss and appears to converge to a different regime, which could indicate weaker or more conservative policy updates.

The critic loss represents the error in predicting value targets, that is, how closely the critic’s value estimates match the bootstrapped returns used as training signals. While lower critic loss indicates more accurate fitting of these targets, it does not necessarily correlate with higher task rewards, since the targets themselves are only approximations of the true returns. This is reflected in the critic loss of Model 2, which stabilizes faster than the other models, despite its overall poorer performance.

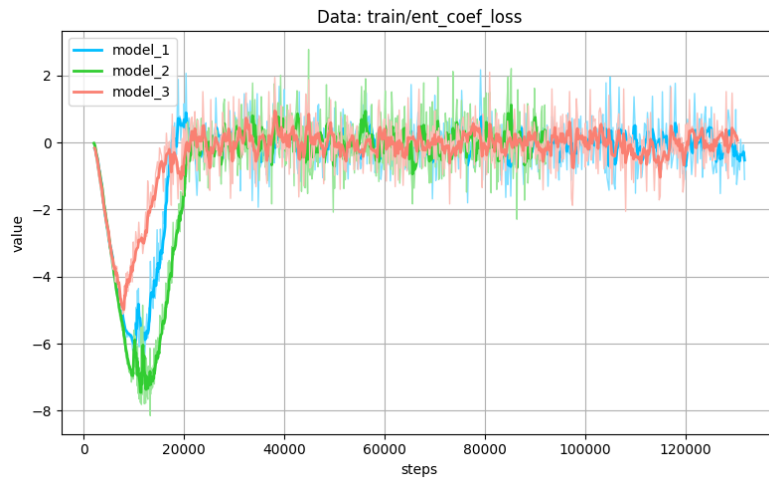
Regarding the entropy loss and its corresponding entropy coefficient, the coefficient starts at 1, promoting stochastic behavior to encourage exploration. It can be observed that, in all trained models, the agents quickly shifted to focus primarily on highly deterministic actions early in training ($\approx 20\,000$ steps) to achieve better performance.



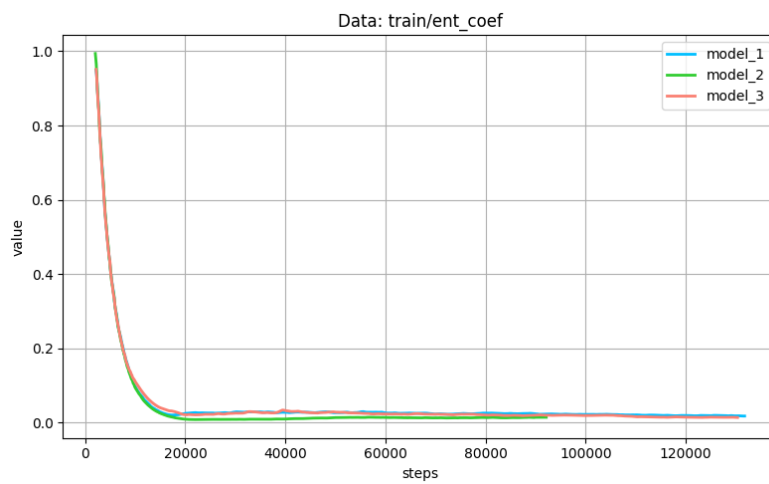
(a)



(b)



(c)



(d)

Figure 6.3: Evolution of the actor loss, critic loss, entropy loss, and entropy coefficient during training, shown in (a), (b), (c), and (d), respectively, for each model training.

Since all models were evaluated every 10 000 steps during training, it was possible to identify the point at which each model achieved its highest reward. As shown in Figure 6.4, the Model 3 reached its peak reward at 110 000 steps. The weights corresponding to this step were then used for the subsequent intensive tests.

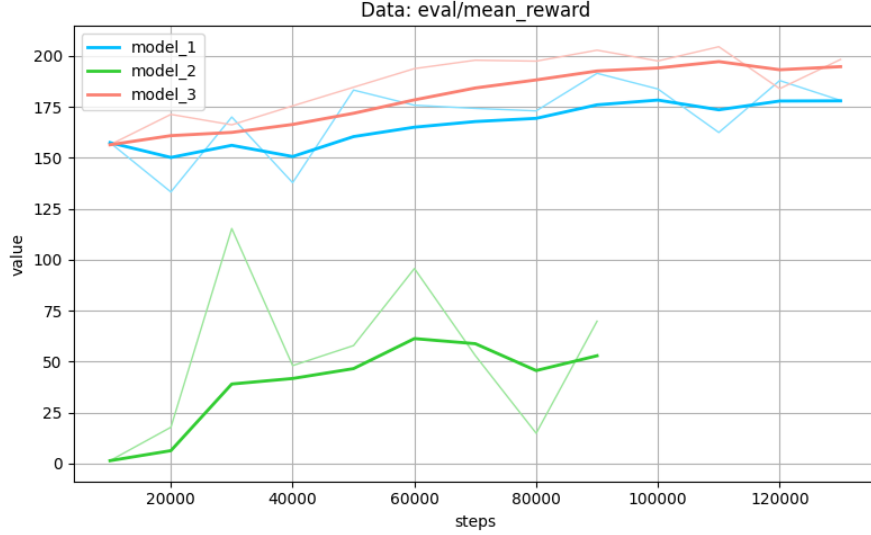


Figure 6.4: Reward obtained in each evaluation phase over the training.

6.2.2 Remarks

After training, the data analysis shows that Model 3, based on stacked images and action inputs, performed better compared to the other models. These observations are consistent with the theory that incorporating past actions introduces a form of memory into the agent, thereby providing a more accurate approximation of the belief state. This, in turn, helps the agent understand how its actions affect the observations over time, without the need for a model dynamics.

With this model selected, the next step was a thorough evaluation under a predefined set of test conditions.

6.3 Best Model Evaluation

The following tests, previously defined and described in Chapter 4.2, were conducted with the best model:

1. **Default test**: demonstrated in Subchapter 6.3.1;
2. **Aligned test**: demonstrated in Subchapter 6.3.2;
3. **Different altitudes test**: demonstrated in Subchapter 6.3.3;
4. **Image perturbations test**: demonstrated in Subchapter 6.3.4;
5. **Image perturbations + changing sunlight test**: demonstrated in Subchapter 6.3.5;

All tests were performed with a new plane/background, the “Rocky” background, to further evaluate the agent’s generalization capabilities to a new type of scene. The main texture of this background is represented in Figure 6.5.



Figure 6.5: Rocky background PBR texture used in all evaluation phases.

Each evaluation comprises 100 episodes, each lasting roughly 132 min, amounting to 660 min in total, or 11 h.

6.3.1 Default Test

The overall test, depicted in Figure 6.6a, concluded that despite bigger PV solar panel arrays, the agent could still navigate with a performance similar to training. It always completed the task (100 %), reaching the end of all arrays, and was perfectly aligned with the panels 71 % of the time, corresponding to a terminal position error below 0.25 m and a yaw error below 3° . This resulted in an average position error $\bar{e}_d$ of 0.0629 m and an average yaw error $\bar{e}_\psi$ of 2.4359° . An example trajectory of the UAV, from the start to end, is also displayed 6.6b.

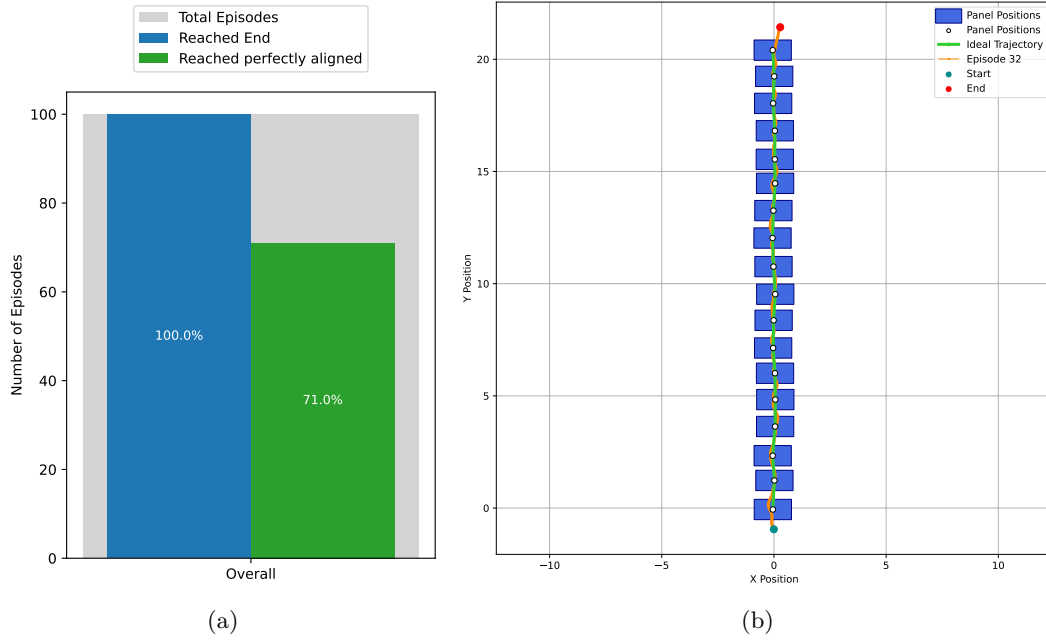


Figure 6.6: Overall reach-end behavior (a) and trajectory example (b) in the default test.

As shown in Figure 6.7, the model demonstrates successful generalization to the new rocky background under background domain randomization. Although there was a slight decrease in perfect panel alignment, the remaining panels showed similar performance.

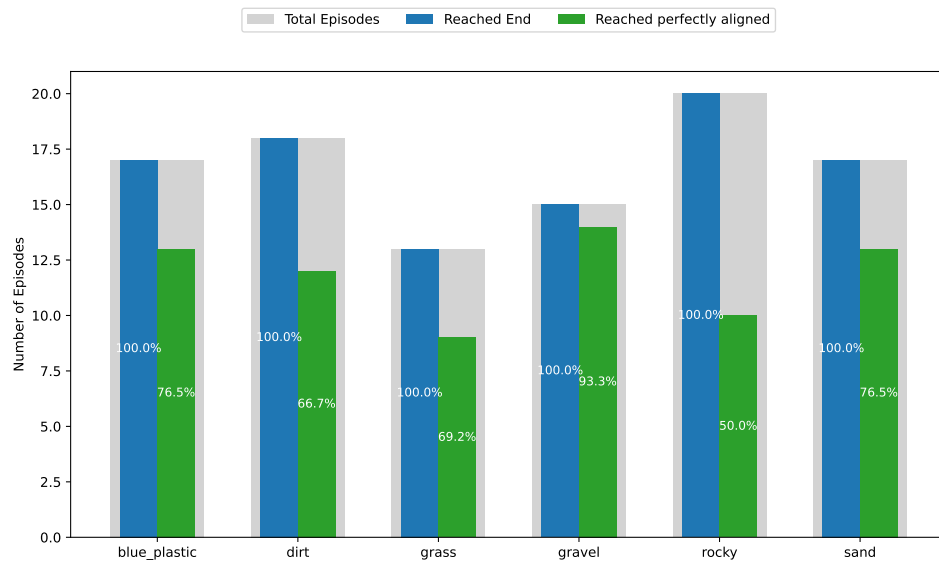


Figure 6.7: Reach-end behavior for each evaluated background in the default test.

Figure 6.8 shows the end behavior across arrays, revealing no significant differences.

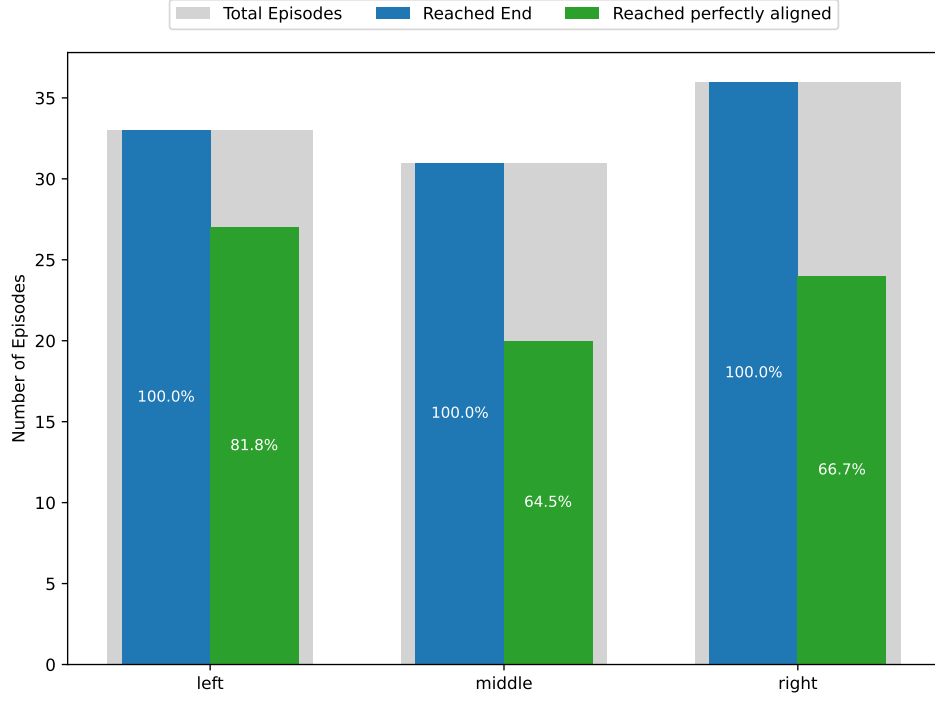


Figure 6.8: Reach-end behavior for each individual array in the default test.

Table 6.3 presents the average errors $\bar{e}_\psi$ and $\bar{e}_d$ together with their standard deviations σ_{e_ψ} and σ_{e_d} for each plane and array. Values are reported as $\bar{e}_\psi \pm \sigma_{e_\psi}$ and $\bar{e}_d \pm \sigma_{e_d}$. Overall, the results are very similar across surfaces and arrays; in particular, the *rocky* background again exhibits good performance even though the agent was never exposed to that surface during training.

Table 6.3: Average Yaw and Position Errors with Standard Deviations for all the planes and arrays in the default test.

Surface/Array	$\bar{e}_\psi$ (°)	σ_{e_ψ} (°)	$\bar{e}_d$ (m)	σ_{e_d} (m)
By Plane				
Blue Plastic	2.5724	0.2070	0.0652	0.0101
Dirt	2.3570	0.2473	0.0613	0.0079
Grass	2.4074	0.2289	0.0643	0.0069
Gravel	2.5487	0.2939	0.0624	0.0095
Rocky	2.3396	0.2592	0.0610	0.0063
Sand	2.4048	0.1941	0.0632	0.0081
By Array				
Left	2.4368	0.2717	0.0643	0.0097
Middle	2.4587	0.2580	0.0642	0.0076
Right	2.4109	0.2396	0.0601	0.0069

6.3.2 Aligned Test

When it comes to the aligned test, the results were also very similar to the default test, with slightly better performance in the reach-end conditions, increasing from 71 % to 81 % in the perfectly aligned condition, as shown in Figure 6.9a. The overall $\bar{e}_\psi$ and $\bar{e}_d$ are also very similar, being 0.0627 m and 2.3488° respectively.

In the trajectory presented in Figure 6.9b, it is possible to notice that being perfectly aligned produces a more precise trajectory.

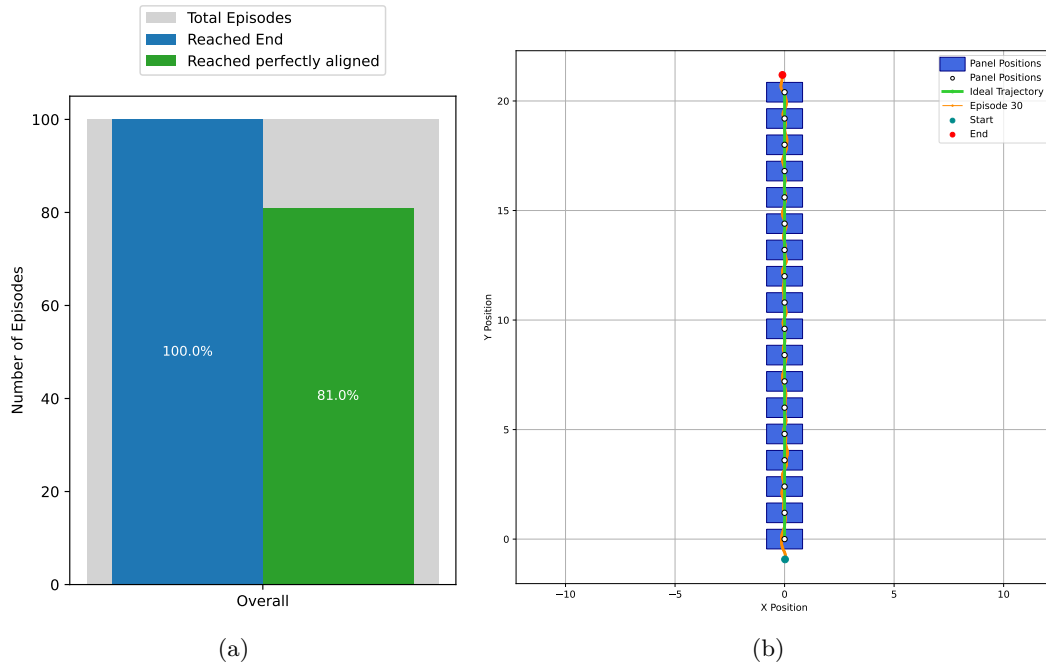


Figure 6.9: Overall reach-end behavior (a) and trajectory example (b) in the aligned test.

The generalization is also present for the new background, just like in the default test, as can be observed in Figure 6.10. As for the individual array performance, the performance is also slightly better, as depicted in Figure 6.11.

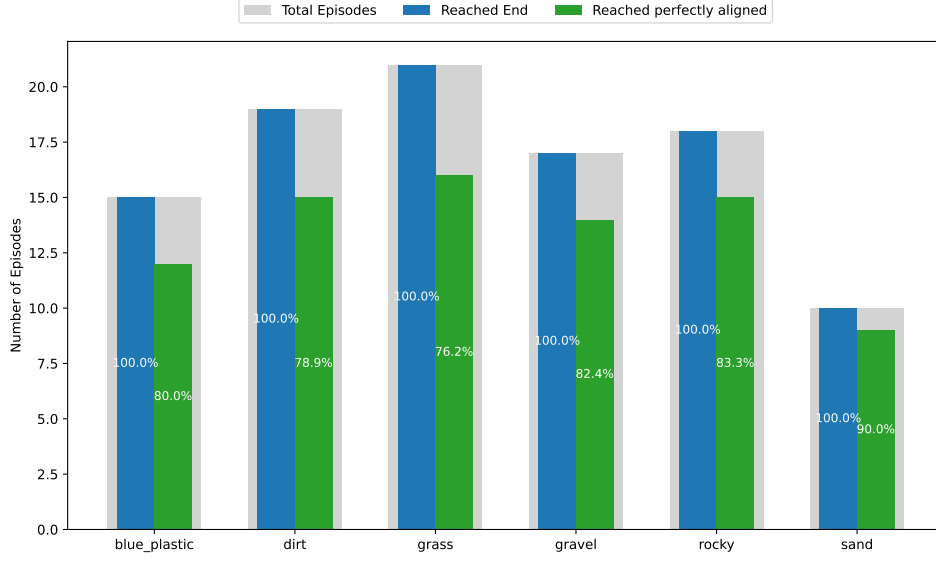


Figure 6.10: Reach-end behavior for each evaluated background in the aligned test.

As reported in Table 6.4, the mean and standard deviation metrics for each element demonstrate robust and reliable performance across conditions.

Table 6.4: Average Yaw and Position Errors with Standard Deviations for all the planes and arrays in the aligned test.

Surface/Array	$\bar{e}_\psi$ ($^\circ$)	σ_{e_ψ} ($^\circ$)	$\bar{e}_d$ (m)	σ_{e_d} (m)
By Plane				
Blue Plastic	2.4147	0.1997	0.0639	0.0063
Dirt	2.3255	0.1485	0.0625	0.0071
Grass	2.3874	0.2620	0.0631	0.0056
Gravel	2.3267	0.1489	0.0603	0.0076
Rocky	2.2943	0.2052	0.0631	0.0052
Sand	2.3511	0.1445	0.0648	0.0078
By Array				
Left	2.3460	0.1861	0.0621	0.0072
Middle	2.3269	0.1517	0.0629	0.0049
Right	2.3641	0.2299	0.0633	0.0069

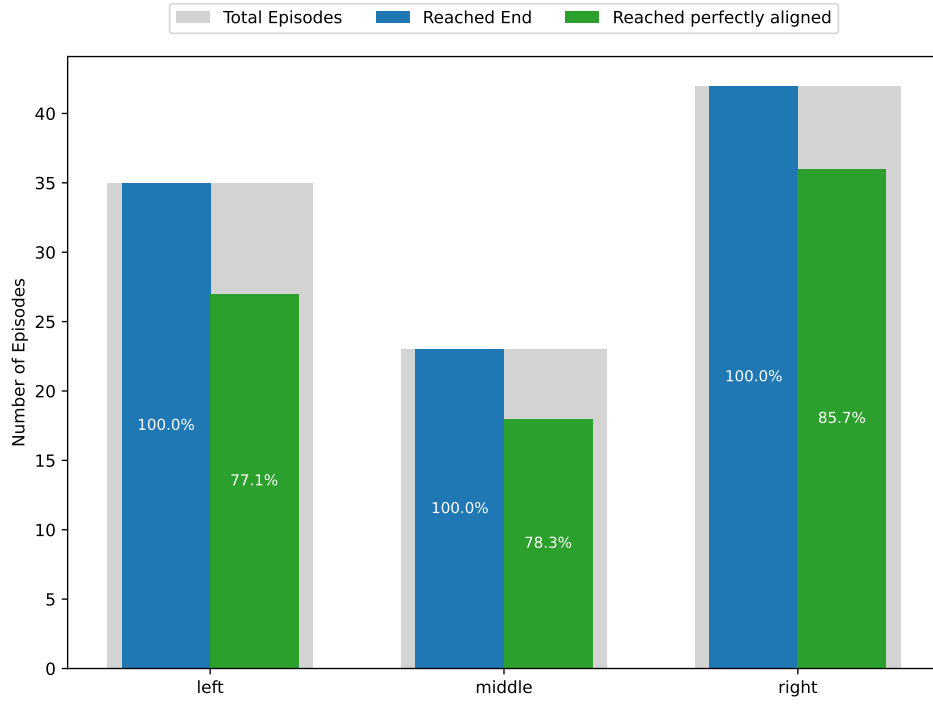


Figure 6.11: Reach-end behavior for each individual array in the aligned test.

6.3.3 Altitude Test

Since the UAV was trained at a fixed flying altitude of 2.4 m, its performance was evaluated at different altitudes, namely 2.5 m, 2.7 m and 2.9 m. The UAV reached the end of the arrays 100% of the time and was perfectly aligned, inside the predefined bounds, 64% of the time, as can be visualized in Figure 6.12a. The overall errors remain consistent with $\bar{e}_d$ equal to 0.0665 m and $\bar{e}_\psi$ equal to 2.4085°, as shown by the trajectory in Figure 6.12b.

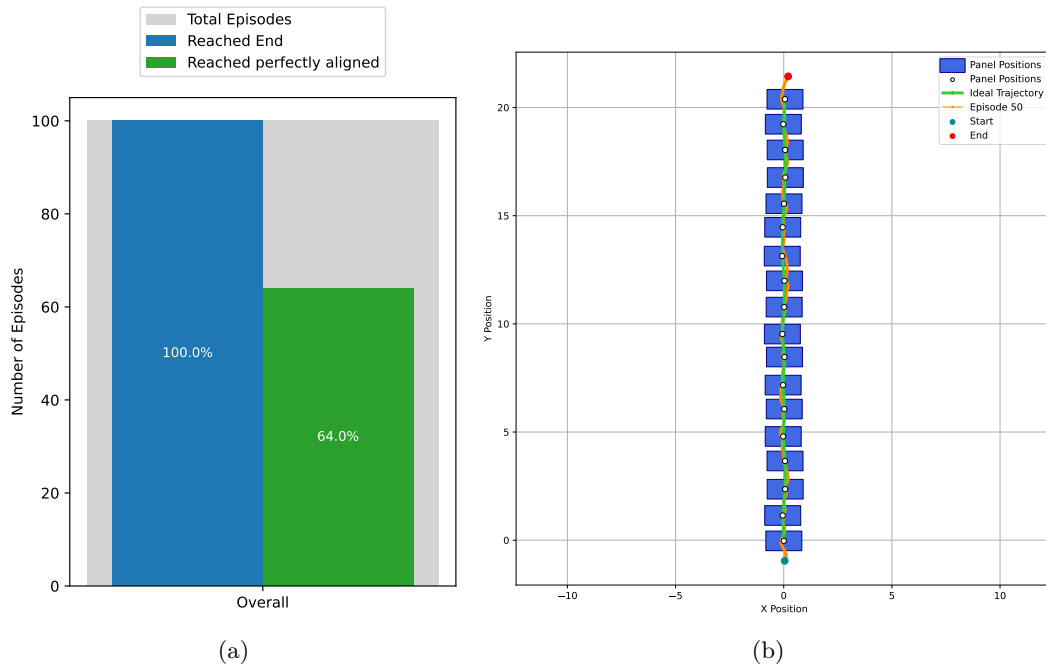


Figure 6.12: Overall reach-end behavior (a) and trajectory example (b) in the altitude test.

Performance across different plane types or backgrounds remained largely consistent. However, an outlier was observed for the “Blue Plastic” background, where perfect alignment was achieved in only 20 % of the episodes. This may be attributed to the limited number of samples for this background, making the result potentially non-representative. In contrast, performance across the different arrays remained stable, with only a slight decrease noted in the left array.

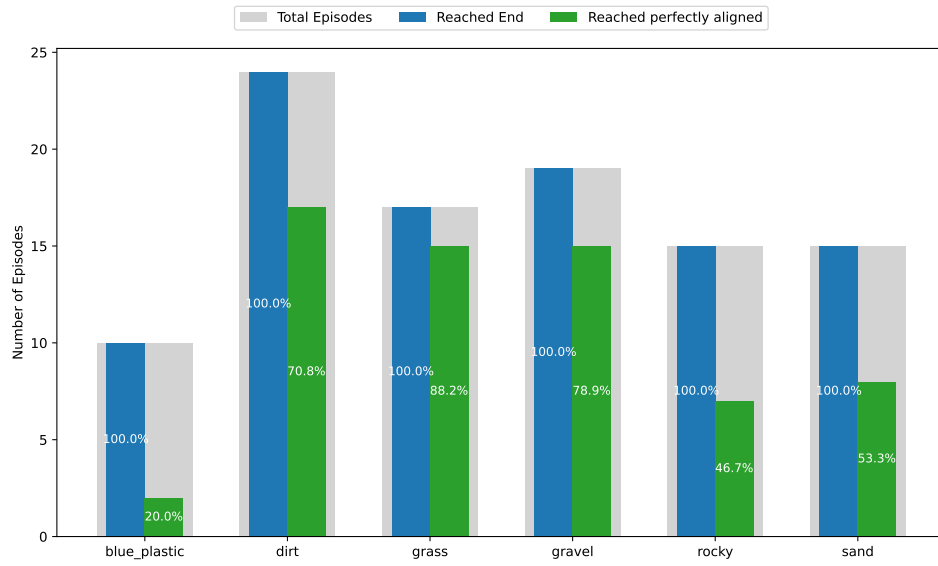


Figure 6.13: Reach-end behavior for each evaluated background in the altitude test.

This test was performed with three different altitudes: 2.5 m, 2.7 m and 2.9 m. Figure 6.15 displays the reach-end performance graph. Surprisingly, it performed slightly worse at the lowest altitude (2.5 m) and generalized well across the other altitudes.

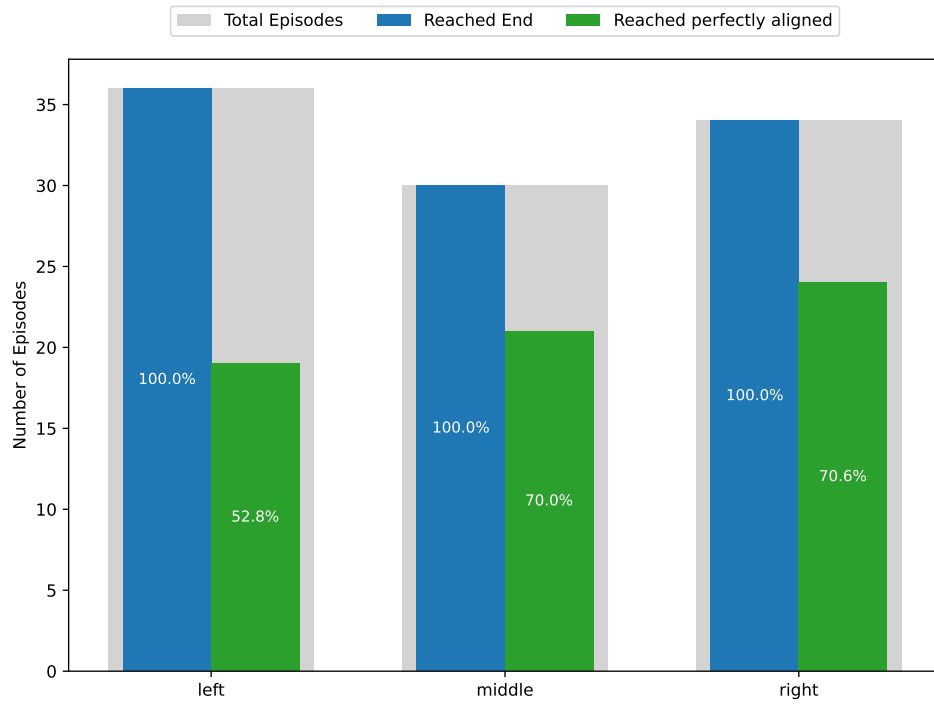


Figure 6.14: Reach-end behavior for each individual array in the altitude test.

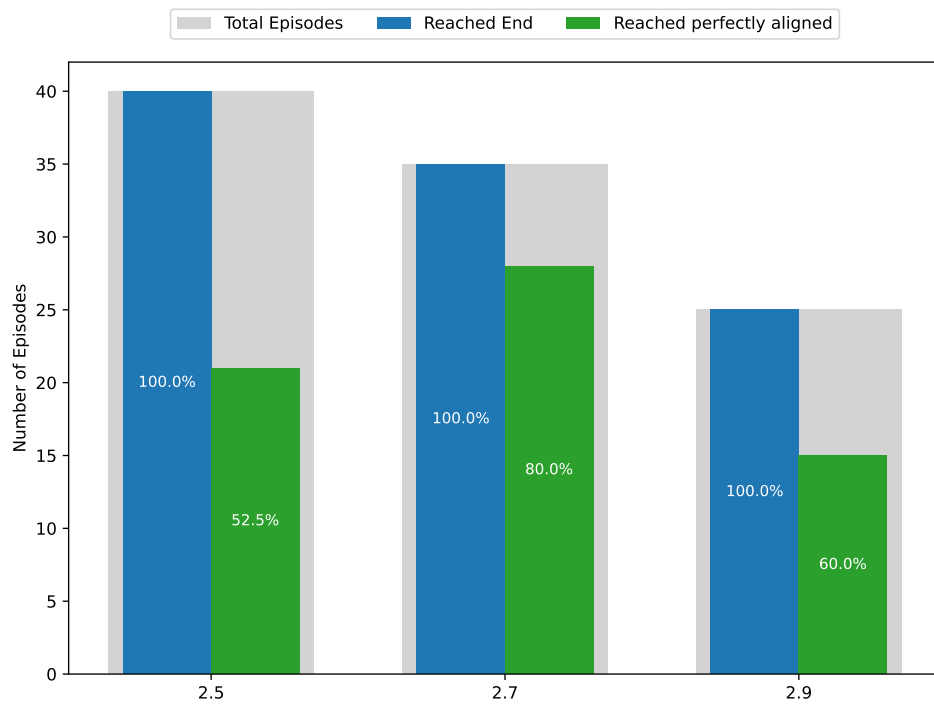


Figure 6.15: Reach-end behavior at 2.5 m, 2.7 m and 2.9 m in the altitude test.

Table 6.5 summarizes the mean and standard deviation metrics for each element, including the different tested altitudes. It indicates that performance remains stable and reliable across conditions.

Table 6.5: Average Yaw and Position Errors with Standard Deviations for all planes, arrays and altitudes, in the altitude test.

Surface/Array/Altitude	$\bar{e}_\psi$ (°)	σ_{e_ψ} (°)	$\bar{e}_d$ (m)	σ_{e_d} (m)
By Plane				
Blue Plastic	2.4115	0.2608	0.0677	0.0114
Dirt	2.3966	0.2887	0.0656	0.0087
Grass	2.4365	0.2775	0.0695	0.0103
Gravel	2.3835	0.2410	0.0663	0.0115
Rocky	2.4040	0.1908	0.0632	0.0060
Sand	2.4429	0.2116	0.0674	0.0089
By Array				
Left	2.3772	0.2173	0.0659	0.0107
Middle	2.4067	0.2416	0.0688	0.0093
Right	2.4490	0.2888	0.0653	0.0089
By Altitude				
2.5 m	2.4009	0.2464	0.0656	0.0098
2.7 m	2.4833	0.2794	0.0686	0.0093
2.9 m	2.3204	0.1796	0.0652	0.0099

6.3.4 Image Perturbation Test

Under perturbation conditions, the overall position state error $\bar{e}_d$ increased slightly, whereas the yaw error $\bar{e}_\psi$ showed no substantial change relative to the default test (0.0800 m for $\bar{e}_d$ and 2.3424° for $\bar{e}_\psi$). Reach-end success remained 100 % across episodes, with perfect alignment achieved in 59 % of trials.

The representative trajectory presented in Figure 6.16b reveals small oscillatory deviations in position, which account for the modest increase in $\bar{e}_d$.

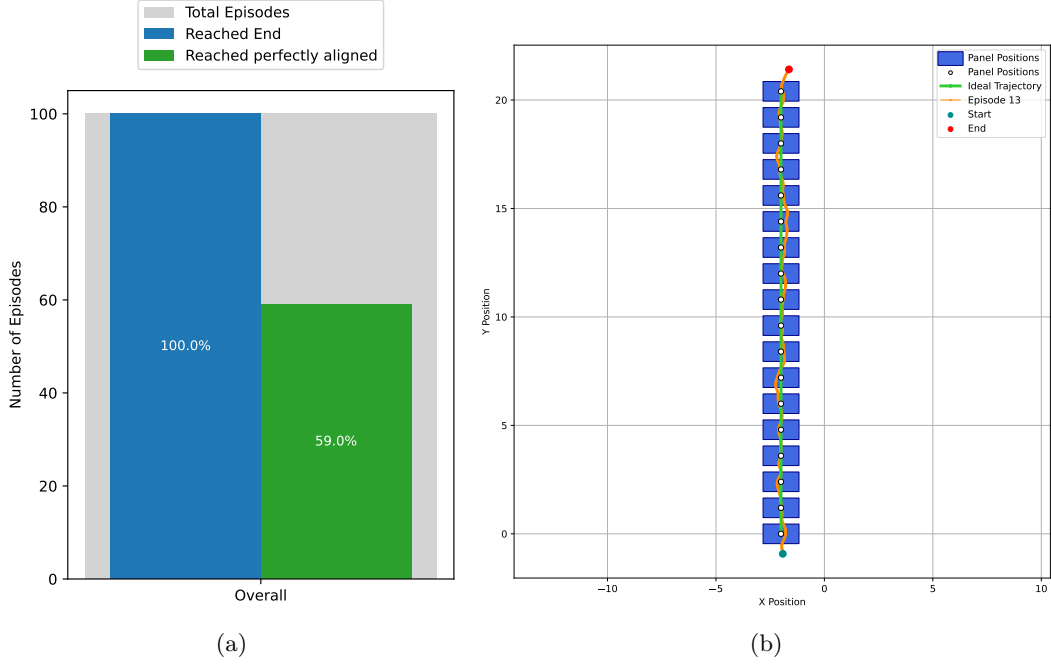


Figure 6.16: Overall reach-end behavior (a) and trajectory example (b) in the image perturbation test.

Just like in the default test, the generalization is also present for the new background, as can be noted in Figure 6.17. The rocky environment shows slightly lower perfect alignment rates at the end of episodes, followed by the grass environment. This suggests that these backgrounds are more affected by perturbations, challenging the model's generalization capabilities.

Considering individual array performance, Figure 6.18 shows a slight drop in perfect reach-end alignment percentage when compared to the default test. This probably also results in the slight increase of the $\bar{e}_d$ state error.

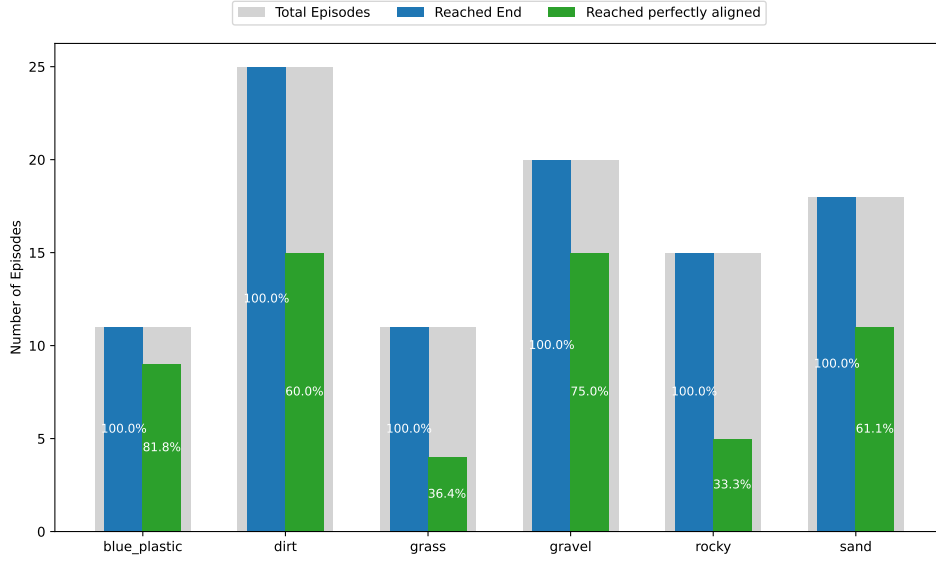


Figure 6.17: Reach-end behavior for each evaluated background in the image perturbation test.

Despite the many perturbations, the UAV agent performed well across all metrics. The “Gravel” background yielded the weakest results with an $\bar{e}_d$ of 0.0888 m and the biggest standard deviation σ_{e_d} of 0.0539 m. The right array also showed a slightly lower performance of 0.0888 m for $\bar{e}_d$, and 0.0504 m for σ_{e_d} .

Due to the similarity, one may conclude that there was a significant overlap of testing episodes when the agent was evaluated with the “Gravel” background on the right array.

Table 6.6: Average Yaw and Position Errors with Standard Deviations for all the planes and arrays in the image perturbation test.

Surface/Array/Altitude	$\bar{e}_\psi$ (°)	σ_{e_ψ} (°)	$\bar{e}_d$ (m)	σ_{e_d} (m)
By Plane				
Blue Plastic	2.2167	0.2398	0.0722	0.0132
Dirt	2.3591	0.2266	0.0765	0.0167
Grass	2.2787	0.2533	0.0825	0.0183
Gravel	2.3440	0.2003	0.0888	0.0539
Rocky	2.3563	0.2492	0.0755	0.0120
Sand	2.4321	0.2539	0.0831	0.0230
By Array				
Left	2.3009	0.2532	0.0754	0.0140
Middle	2.3230	0.1932	0.0789	0.0133
Right	2.4380	0.2494	0.0890	0.0504

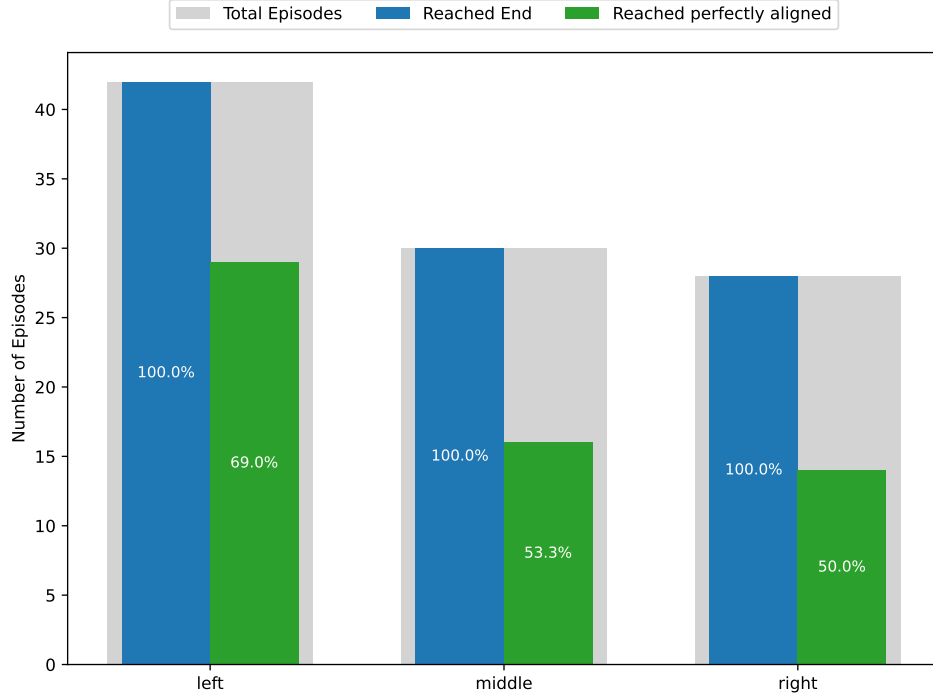


Figure 6.18: Reach-end behavior for each individual array in the image perturbation test.

6.3.5 Image Perturbation and Changing Sunlight Test

The final test combines image perturbations with changes in the sunlight position, with the Sun placed at a different random position in each evaluation episode, representing different times of day. The overall results of the agent were not satisfactory. The position state error $\bar{e}_d$ increased to a value of 0.1517m, while the $\bar{e}_\psi$ also increased substantially to 5.1056°.

This decrease in performance is also evident in the reach-end conditions: the agent reached the end only in 79% of the episodes, being perfectly aligned just 31% of the episodes. Figure 6.19b shows that the agent oscillates and then drifts excessively, eventually going out of bounds.

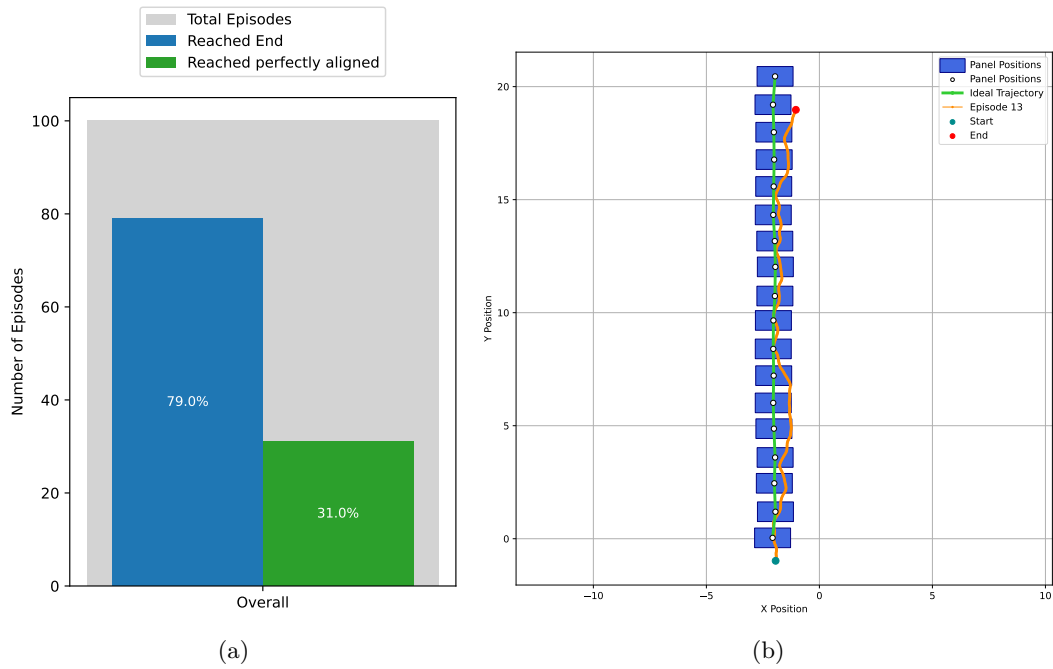


Figure 6.19: Overall reach-end behavior (a) and trajectory example (b) in the image perturbation and changing sunlight test.

This same lack of performance is universal across all the backgrounds/planes present in Figure 6.20, as well as with all the arrays in Figure 6.20.

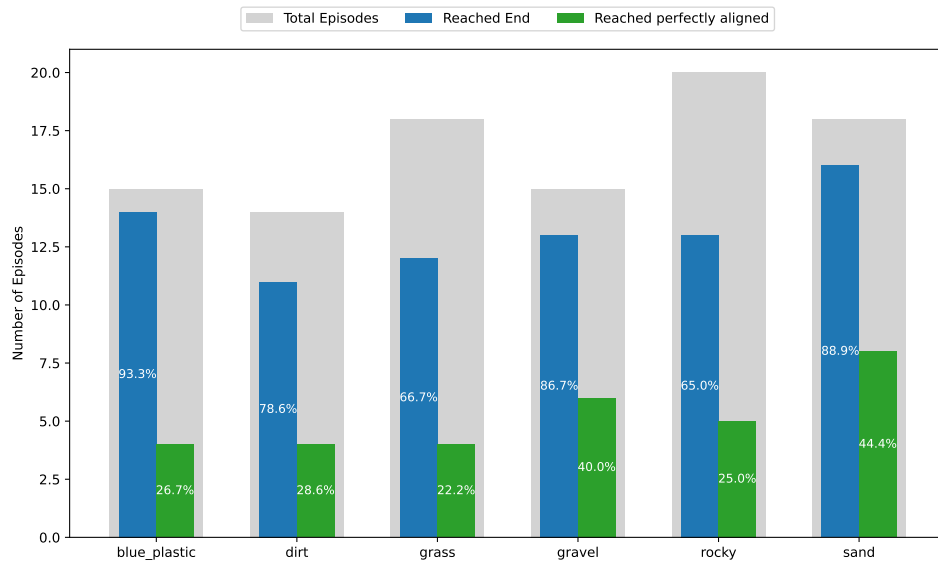


Figure 6.20: Reach-end behavior for each evaluated background in the image perturbation and changing sunlight test.

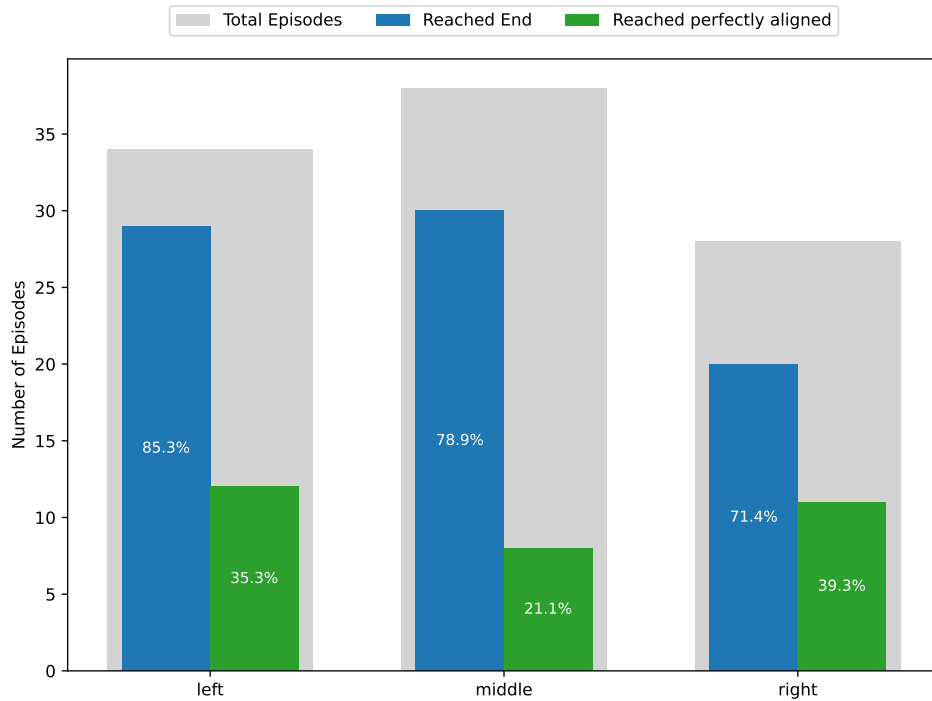


Figure 6.21: Reach-end behavior for each individual array in the image perturbation and changing sunlight test.

The universal decrease in performance is also evident in the specific statistics in Table 6.7. In addition to elevated means ($\bar{e}_\psi$, $\bar{e}_d$), the standard deviations (σ_{e_ψ} , σ_{e_d}) are large, indicating substantial undesirable oscillation. The middle array exhibits the largest errors and increased variability. When it oscillates excessively, it can accidentally observe too much of neighboring arrays, which, in turn, confuses the agent about which track to follow.

Table 6.7: Average Yaw and Position Errors with Standard Deviations for all the planes and arrays in the image perturbation and changing sunlight test.

Surface/Array/Altitude	$\bar{e}_\psi$ (°)	σ_{e_ψ} (°)	$\bar{e}_d$ (m)	σ_{e_d} (m)
By Plane				
Blue Plastic	5.2722	4.1122	0.1429	0.1079
Dirt	5.1238	2.7035	0.1568	0.0970
Grass	5.4220	2.8648	0.1800	0.1171
Gravel	4.0518	1.7538	0.1442	0.1401
Rocky	5.6216	4.7428	0.1567	0.1289
Sand	5.0656	2.7327	0.1306	0.0630
By Array				
Left	4.3962	2.0725	0.1360	0.0941
Middle	6.2507	4.5358	0.1909	0.1457
Right	4.5353	2.4562	0.1218	0.0582

6.3.6 Remarks

The evaluation tests confirmed that the agent performed consistently in the larger PV solar plant under the Default Test and generalized effectively in both the Aligned and Altitude tests. It also showed satisfactory robustness in the Image Perturbation Test, handling common non-ideal conditions. However, in the final test—combining image perturbation with changing sunlight—the RL agent failed to meet expectations, yielding poor results across all metrics.

Further analysis revealed partial overfitting to the training environment. Specifically, the agent had implicitly learned to rely on its projected shadow, which remained centered due to the fixed midday sunlight during training. When the sunlight position varied during evaluation, the shadow shifted, causing the agent to misinterpret its position and leading to increased oscillations and degraded performance.

Given the need for real-world deployment, this behavior represents a critical limitation. As a result, a revised training strategy must be considered to ensure robustness under varying lighting conditions.

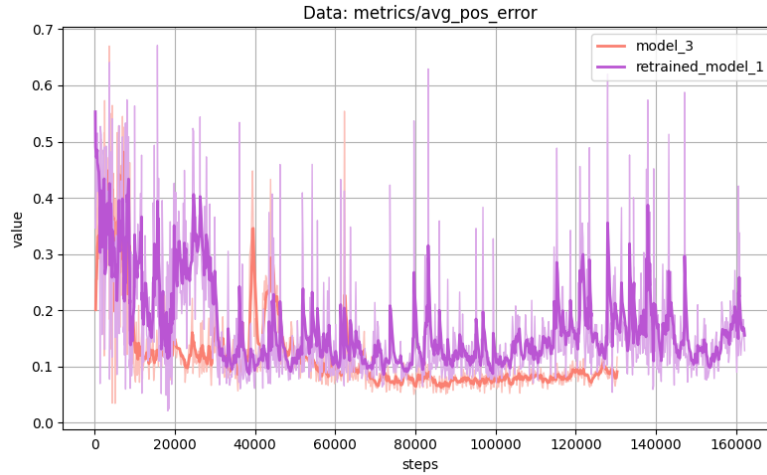
6.4 RL Agent Retraining

The first attempted focused on training the agent with varying sunlight conditions, without modifying Model 3. This first retraining took 908 min, practically 15.13 h.

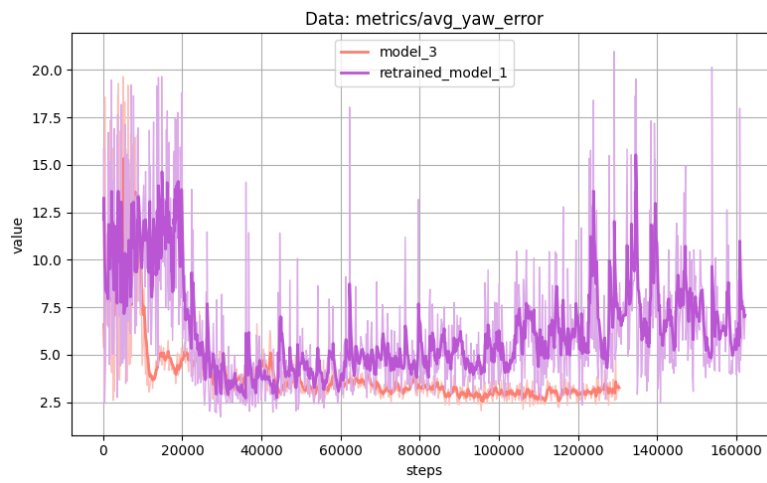
6.4.1 First Retraining

After the first retraining, it became clear that the process would not be so straightforward. Training in the updated environment, where sunlight position varied across episodes, resulted in noisier outcomes and lower average performance. Even after extended training periods, the agent failed to learn effectively, with performance degrading as training progressed.

Figure 6.22 displays the metrics related to $\bar{e}_\psi$ and $\bar{e}_d$. The average yaw error ($\bar{e}_\psi$) begins to degrade noticeably after approximately 40 000 training steps. This supports the observation that the agent struggled to adapt effectively to sunlight variation during retraining.



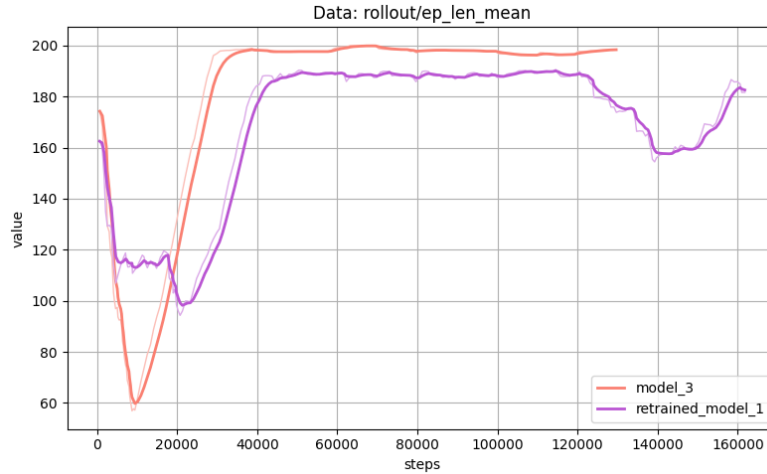
(a)



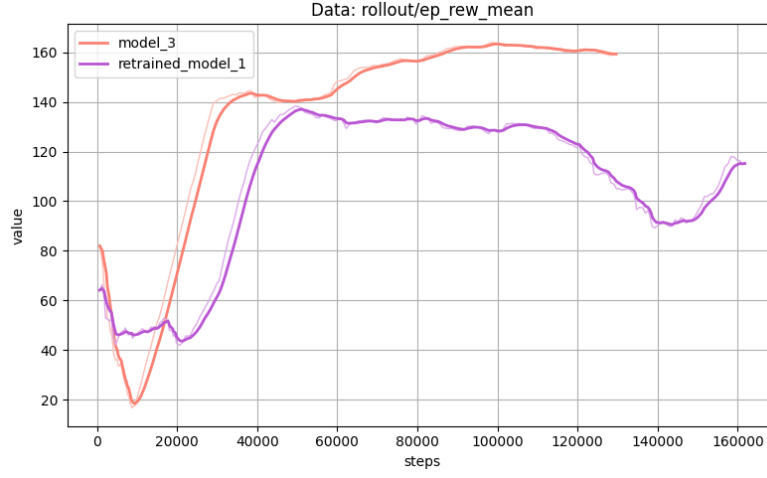
(b)

Figure 6.22: Average position metrics (a) and average yaw metrics (b) of Model 3 and the retrained model.

The same behavior can be observed in the smoothed episode length and average reward shown in Figure 6.23. The agent never reached the maximum episode length, and the reward exhibited a downward trend, consistent with the other metrics.



(a)



(b)

Figure 6.23: Smoothed episode length (a) and smoothed episode reward (b) of Model 3 and the retrained model.

With these data, a series of tests were conducted to identify the main problem restricting the model from learning correctly.

6.4.2 Regularization

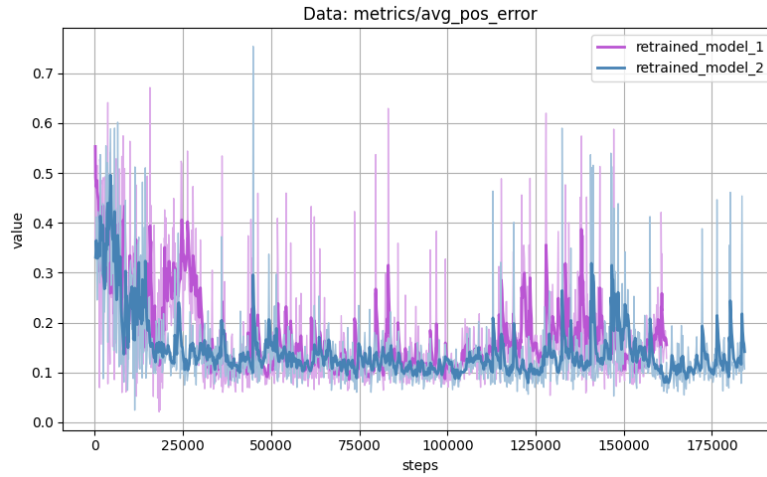
The high training variance observed suggested that regularization could improve learning. To minimize architectural changes, normalization layers were added to the feature extractor.

While the PyTorch framework provides several types of normalization layers, not all are equally suitable for RL. The classic Batch Normalization (BatchNorm) is often not recommended for reinforcement learning problems because it relies on minibatch statistics. When minibatches are small, correlated, or non-stationary, as is common in RL, the estimated means and variances are noisy, destabilizing the learning process [112]. By contrast, batch-independent normalizers are usually preferable for RL feature extractors. Group Normalization (GroupNorm) is well suited for convolutional blocks: it normalizes each sample over groups of channels (and spatial dimensions), preserving spatial structure while avoiding dependence on the batch [113]. Layer Normalization (LayerNorm) is also a natural fit for MLPs since it normalizes each sample across feature dimensions, improving RL results [114].

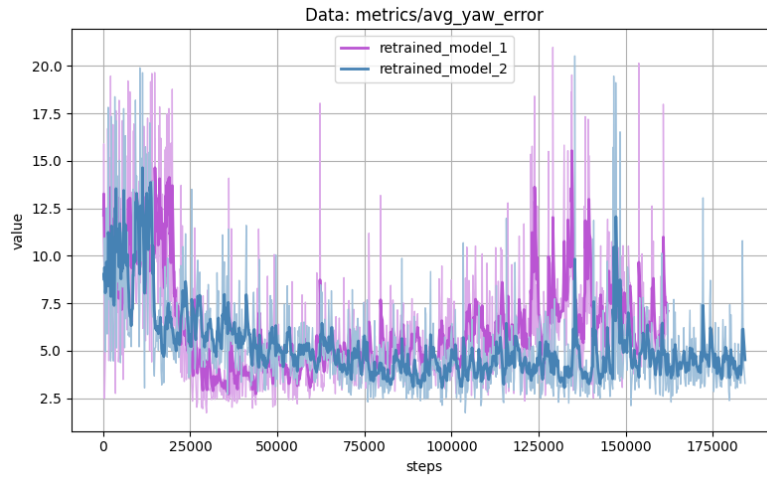
Consequently, an updated feature extractor architecture was developed, as shown in Appendix A.3. GroupNorm layers were added between convolutional blocks and ReLU activations in the CNN, while LayerNorm was applied between linear layers and ReLU activations in the MLP. This retraining required 954 min (15.9 h), slightly

longer than previous sessions, with the aim of potentially achieving improved results through extended training.

Figure 6.24 holds the retraining metrics with the regularization techniques. These changes made a significant impact on the stability of the training, resulting in better overall $\bar{e}_\psi$ and $\bar{e}_d$ space state errors.



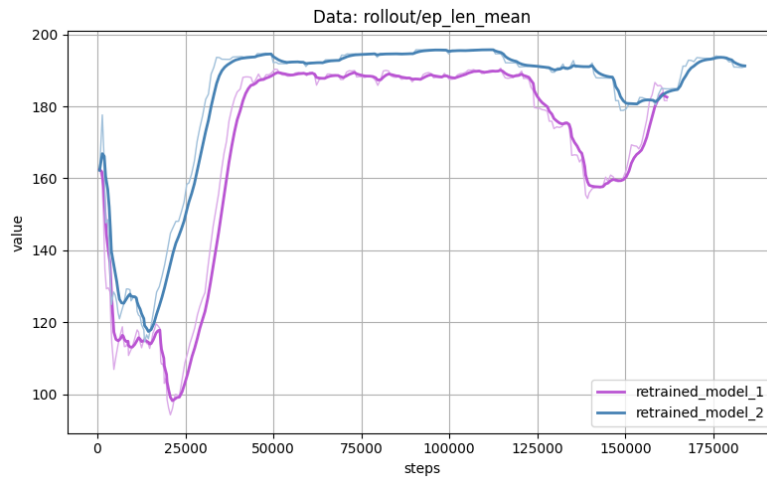
(a)



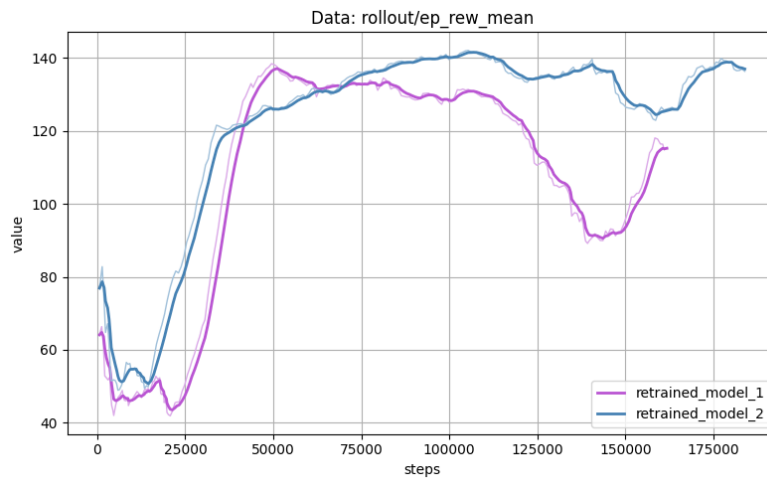
(b)

Figure 6.24: Average position (a) and average yaw metrics (b) for the retraining of the first and second (with regularization) retrained models.

More noticeable is the increase in episode length, indicating that the agent can avoid going out of bounds for a longer time. It also achieves a more stable reward with a slightly higher smoothed average, resulting from the accumulation of better metrics. This can be confirmed in Figure 6.25.



(a)



(b)

Figure 6.25: Smoothed episode length (a) and smoothed episode reward (b) for the retraining of the first and second (with regularization) retrained models.

Regardless of the improvement in stability, the agent still falls behind of the previous “overfitted” performance, of Model 3. To reduce this gap, another modification was implemented.

6.4.3 Hyperparameter Tuning

To minimize architectural changes, the hyperparameters listed in Table 6.2 were tuned. For simplicity, some results are described without a graphical representation.

The mini-batch size (N), number of gradient steps (G), and soft-update coefficient (τ) were first adjusted. Increasing the mini-batch size degraded performance, consistent with the known sensitivity of RL algorithms to large batches. Additionally, real-time computation was affected: as the Gazebo simulator runs continuously during gradient updates, longer computations slowed the simulated environment.

Consequently, increasing the number of gradient steps (G) produces a similar adverse effect, effectively increasing the environment’s real-time step and further degrading performance.

The soft-update coefficient (τ) was also tested by reducing it to the default value of 0.005 reported in the original SAC implementation of the SB3 [103]. But surprisingly, decreasing τ in this case led to worse overall performance. The τ parameter controls the Polyak (soft) updates, which are responsible for gradually updating the target-actor and target-critic networks, as previously explained in Chapter 2.3. Reducing τ slows these updates, producing smoother, more stable target networks; however, overly slow updates can cause the target networks to lag behind the online networks, resulting in stale targets that slow the propagation of learning signals and degrade performance.

After further testing, a bottleneck was identified. Since the agent’s environment now exhibits greater variance, caused by the randomized sunlight conditions at each episode, the previously used replay buffer size became insufficient to represent this variability. However, after testing the limits, a replay buffer of size 50 000 was already close to the limit of the available memory on the training machine. To address this issue, an optimization flag provided in the SAC implementation was enabled, which reduces *Random-Access Memory* (RAM) usage by storing observations and next-observations within a single shared array.

In this way, a replay buffer of size 80 000 was achieved. To further account for the increased complexity and variance of the environment, two additional adjustments were applied. First, the learning rate lr was reduced from 3×10^{-4} to 1×10^{-4} . Second, the “learning starts” parameter N_{start} was set to 5000, ensuring that the replay buffer was filled with a larger amount of diverse experience before the learning process began.

With all of this information, the final version of SAC algorithm was trained with the hyperparameters depicted in Table 6.8.

Table 6.8: Final SAC hyperparameters used in the retraining.

Parameter	Symbol	Value
Learning rate	lr	1×10^{-4}
Replay buffer size	B	80^5
Learning starts	N_{start}	5000
Batch size	N	256
Soft update coefficient	τ	0.01
Discount factor	γ	0.99
Train frequency	f_{train}	1
Gradient steps	G	1
Initial entropy coefficient	α	1.0
Target entropy	H_{target}	$-\dim(\mathcal{A})$
Target update interval	T_{target}	1

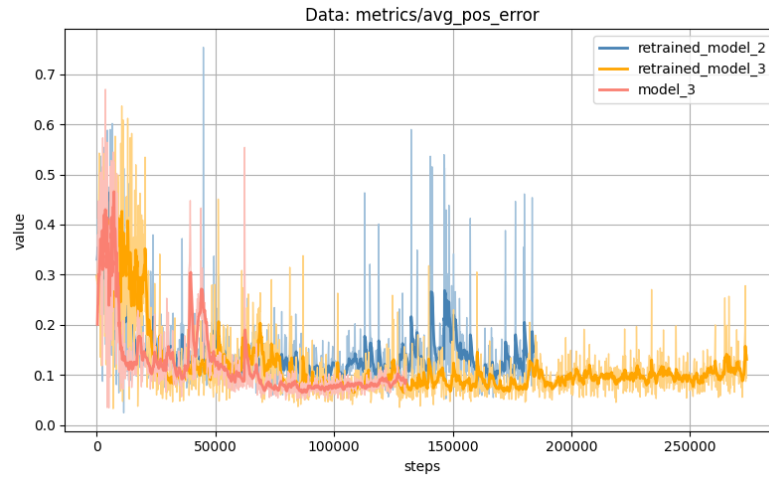
6.4.4 Final Model Analysis

With all the configurations previously mentioned, overall positive results were obtained. In Figure 6.26, the “retrained_model_3” is presented and compared with the previously retrained model (“retrained_model_2”) and the original overfitted “model_3”.

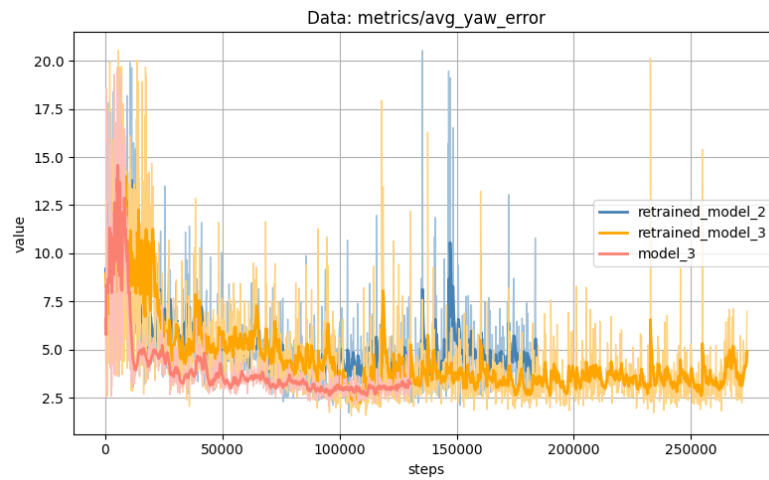
The retraining of the new model required more iterations, primarily due to the reduced learning rate and the increased amount of replay buffer data. This, in turn, represents the longest-trained model, with a total training time of approximately one day, more precisely, 1462 min (24.37 h).

Analysis of the results indicates that the newly trained model achieved improved performance compared to the previously retrained model. The most significant finding is that it reached precision metrics comparable to those of the original Model 3. In particular, it achieved its best performance between 150 000 and 200 000 steps, with an average position error approximately equal to that of the original Model 3, and only a slightly worse yaw error.

In this retraining, the new retrained model was also trained with a new rocky background, introducing even more diversity to the learning process.



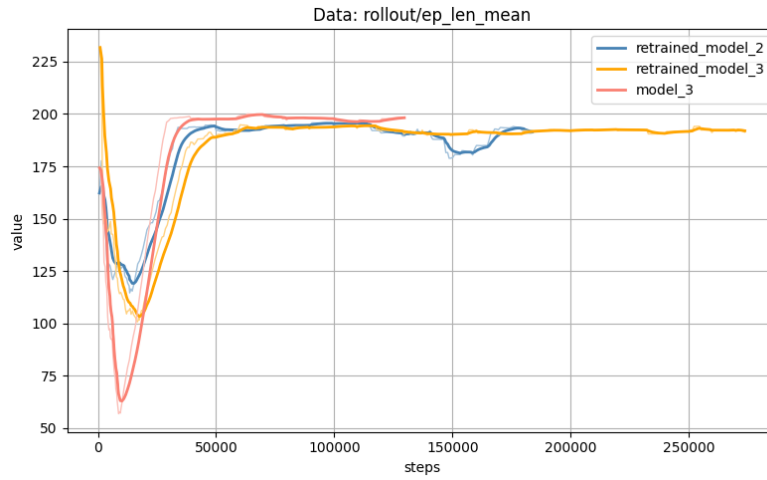
(a)



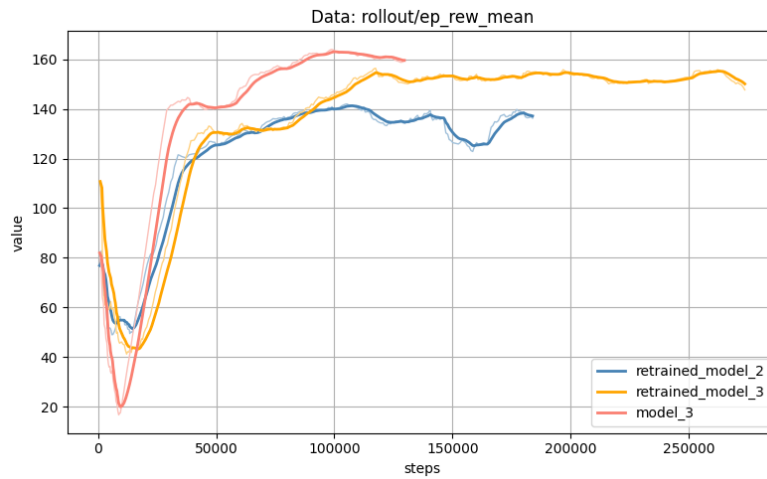
(b)

Figure 6.26: Average position metrics are represented in a) and average yaw metrics are represented in b), comparing the performance of the new retrained model.

In Figure 6.27, the smoothed episode lengths shown in Figure 6.27a are very similar across all models, with the original Model 3 exhibiting the longest episode duration. Furthermore, Figure 6.27b shows that the new retrained model, although significantly better than the previous retrained model, never reached the reward performance of the overfitted model. This outcome can also be interpreted as a positive sign, as it suggests that the model is not merely memorizing the environment but is instead genuinely learning. In addition, the new retrained model displays a smoother learning curve, with reward accumulation increasing more gradually and consistently compared to the other models.



(a)



(b)

Figure 6.27: Smoothed episode length in a) and smoothed episode reward in b) comparing the final retrained model, first retrained model, and the original Model 3 model.

As mentioned previously, losses in RL can provide only a limited indication of

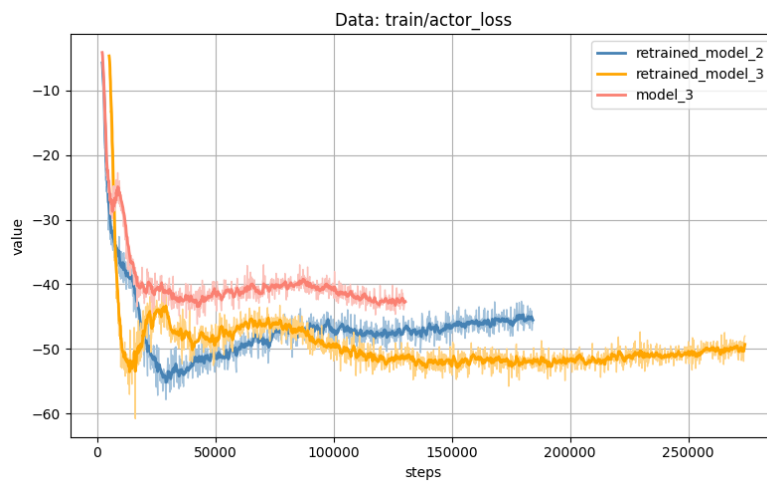
performance and do not always correspond directly to better outcomes. Although in this case, the final retrained model achieves the lowest loss, which may indicate a more robust policy compared to the overfitted model. The second retrained model also exhibits a lower loss than the original model; however, it is less stable, displaying an increasing tendency over time. This is all observed in Figure 6.28a.

Examining all the critics' losses in Figure 6.28b, it is possible to observe that both retrained models exhibit high-variance losses. This increased variability may be a consequence of the higher variance in the environment, which can negatively impact the critic's performance.

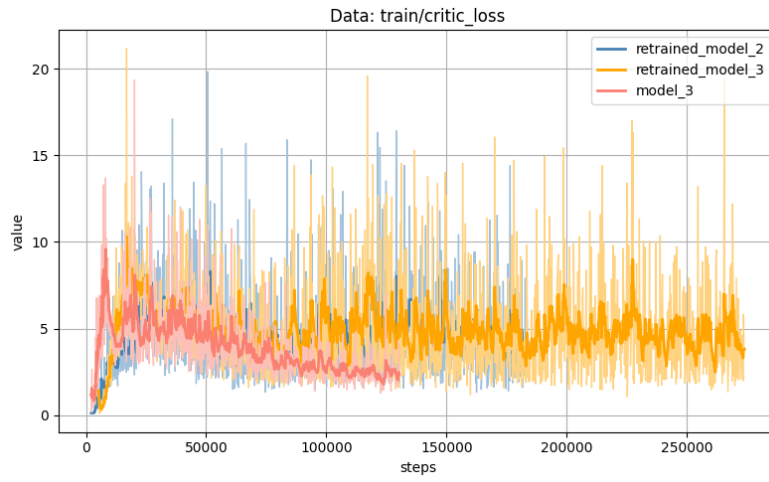
Indeed, the critic's role is to accurately evaluate the value of the current state. When environmental variance is high, it becomes more difficult for the critic to determine whether a given state is valuable, thereby increasing the instability of its estimates.

Looking at the entropy coefficient loss of all the models in Figure 6.28c. All of them converged to a more deterministic policy. But it is possible to observe the impact of the slower learning rate in the new retrained model, which made the exponential decline of the curve smoother compared to the others.

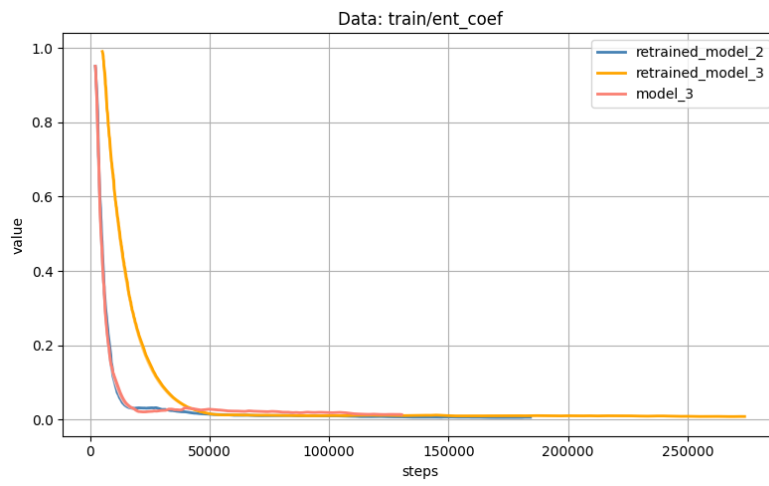
The same impact of the lower learning rate is observed in the entropy loss in Figure 6.28d. However, all the models converged to the same loss value and similar variance.



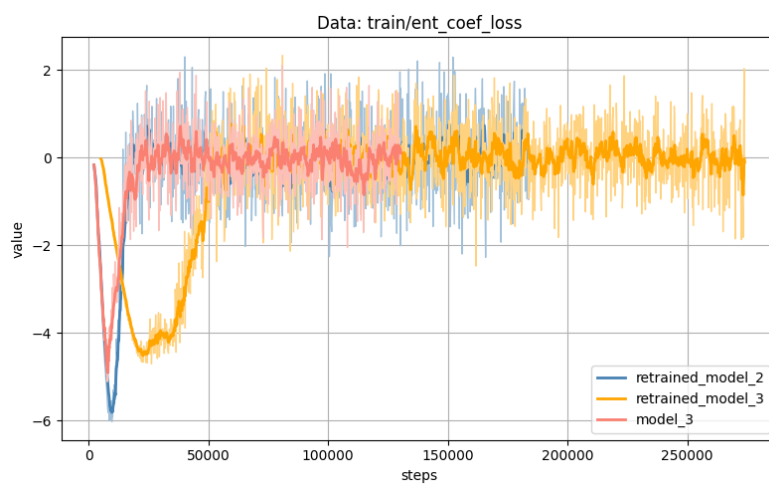
(a)



(b)



(c)



(d)

Figure 6.28: Training evolution of actor, critic, and entropy losses and the entropy coefficient, shown in (a)–(d), for the final retrained, first retrained, and original Model 3.

The retrained models were also validated every 10 000 steps to identify the best-performing model during training. As shown in Figure 6.29, the new retrained model achieved its highest validation performance at 170 000 steps, which corresponds to the optimal performance window observed previously in the training metrics. Although it never surpassed the overfitted model, it reached a validation reward that was very close to that of the overfitted model.

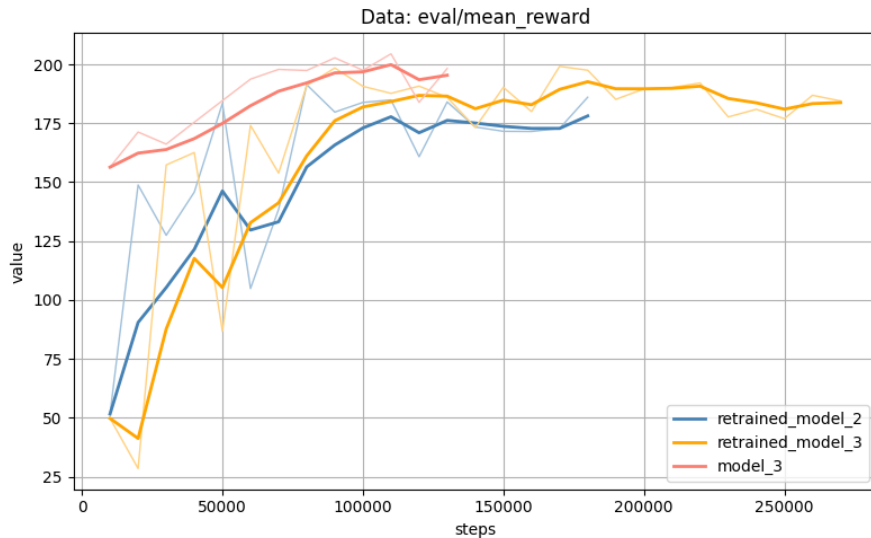


Figure 6.29: Reward obtained in each evaluation phase over the training.

6.4.5 Remarks

Examining the retraining results, it is evident that the new regularization techniques and hyperparameter tuning significantly improved the model's performance. Although it never reached the precision achieved by the overfitted model, the training data suggest that this new model is less likely to suffer from the same issues encountered by the previous model.

However, to confirm this hypothesis, a final evaluation must be conducted to verify that the model can reliably achieve its intended performance. This will be conducted in the final Subchapter 6.5.

6.5 Final Evaluation

In the final evaluation, a series of tests was conducted to assess the performance evolution of the final retrained model. This time, since the model was trained with varying sunlight conditions between episodes, the default, aligned and image perturbation tests were repeated under the same randomness. This approach removes

the need for separate fixed-condition evaluations, while remaining aligned with the project’s objectives. Specifically, the repeated tests were the following:

1. **Default test:** demonstrated in Subchapter 6.5.1;
2. **Aligned test:** demonstrated in Subchapter 6.5.2;
3. **Different altitude test:** demonstrated in Subchapter 6.5.3;
4. **Image perturbation test:** demonstrated in Subchapter 6.5.4;

To recap, each evaluation consists of 100 episodes, corresponding approximately to 132 min. In this final evaluation, the 4 tests took 528 min, or 8.8 h.

6.5.1 Default Test

The overall result is presented in Figure 6.30a. It achieved 100 % task completion with 72 % of episodes perfectly aligned. The calculated average position error $\bar{e}_d$ was 0.0930 m and the average yaw error $\bar{e}_\psi$ was 3.5760° . These values are slightly increased when compared with those of the original model in Chapter 6.3.1. However, considering trajectories, such as the one portrayed in Figure 6.30b, the additional error is insignificant.

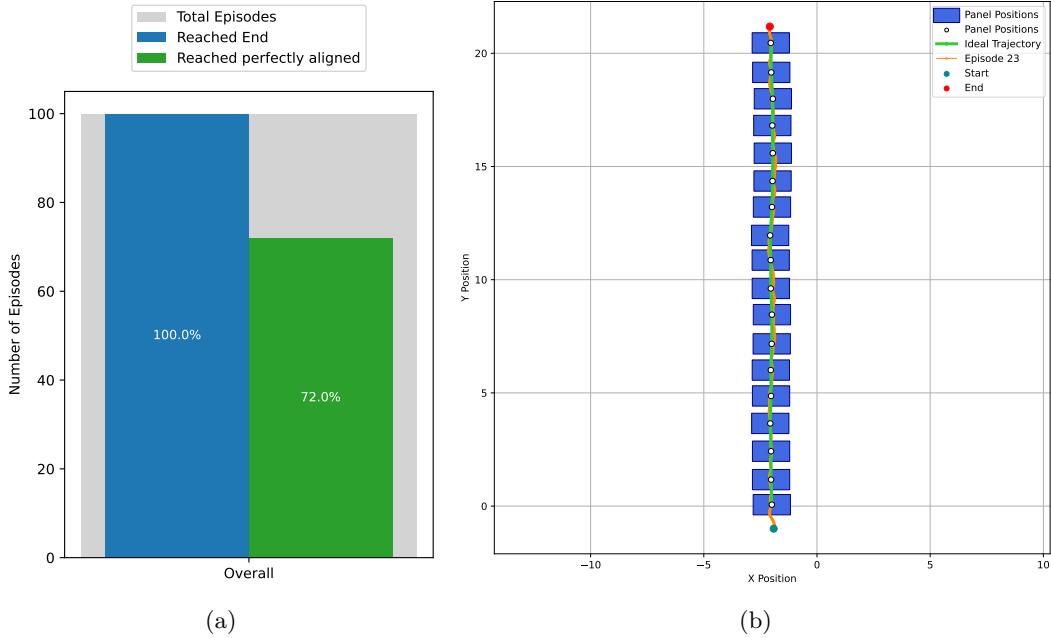


Figure 6.30: Overall reach-end behavior (a) and trajectory example (b) in the default test with the retrained model.

Figure 6.31 demonstrates that the model has strong reach-end performance across all backgrounds, performing best in the grass background and worst in the rocky background.

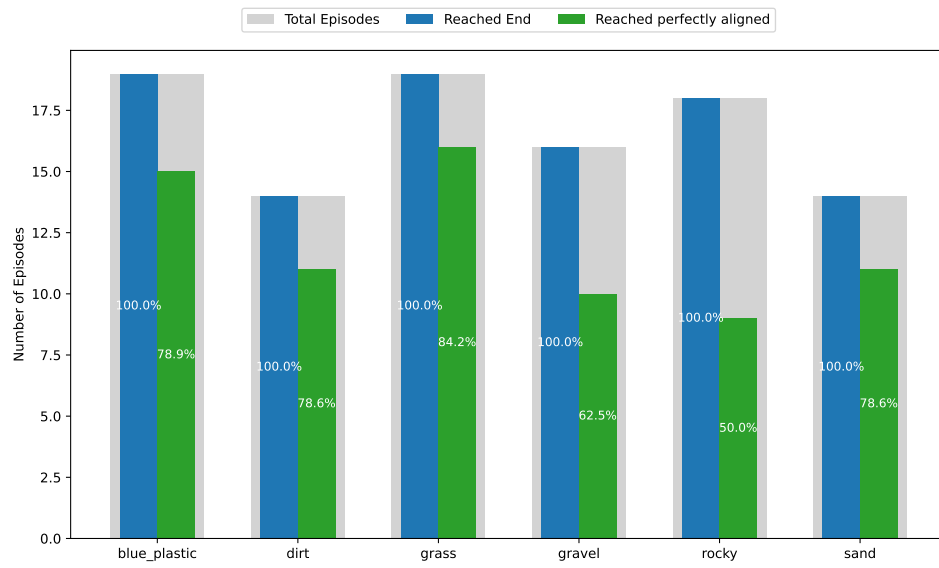


Figure 6.31: Reach-end behavior for each evaluated background in the default test with the retrained model.

Figure 6.32 shows that, in terms of individual array reach-end performance, there is no significant difference, besides performing slightly worse on the left array.

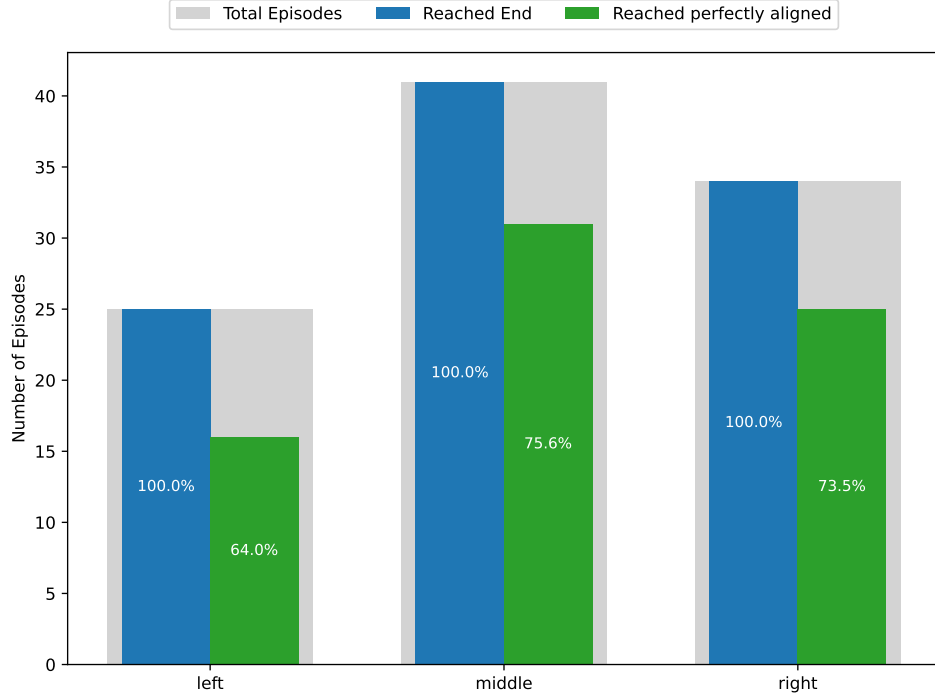


Figure 6.32: Reach-end behavior for each individual array in the default test with the retrained model.

Table 6.9 shows the average and standard deviation errors of the position and yaw errors. The model performance is mostly submetric (0.10 m), with the grass background displaying the highest $\bar{e}_d$ state-error. The top $\bar{e}_\psi$ of 4.5074° occurs with the dirt background. Regarding the different arrays, there is no significant disparity.

Table 6.9: Average Yaw and Position Errors with Standard Deviations for all the planes and arrays in the default test with the retrained model.

Surface/Array	$\bar{e}_\psi$ ($^\circ$)	σ_{e_ψ} ($^\circ$)	$\bar{e}_d$ (m)	σ_{e_d} (m)
By Plane				
Blue Plastic	3.3847	1.7652	0.0839	0.0147
Dirt	4.5074	2.6001	0.0946	0.0371
Grass	3.1757	0.9368	0.0956	0.0491
Gravel	3.6129	0.9580	0.1051	0.0439
Rocky	3.6891	0.9536	0.0866	0.0191
Sand	3.2979	1.0343	0.0956	0.0259
By Array				
Left	3.0380	0.7617	0.0853	0.0162
Middle	3.7657	1.4874	0.0941	0.0397
Right	3.7545	1.8677	0.0974	0.0365

6.5.2 Aligned Test

A surprising result was observed in the aligned test displayed in Figure 6.33a. The UAV RL agent reached the end in only 99 % of the episodes. Under the perfectly aligned condition bounds, the rate dropped to 71 %. Since the PV panels remain unchanged across episodes, this outcome may be explained by rare sunlight conditions – known as outliers – corresponding to improbable configurations underrepresented during training.

Nonetheless, the overall average position error $\bar{e}_d$ and the overall average yaw error $\bar{e}$ reached 0.0863 m and 3.5480° , respectively. These results surpass those from the new default test and are only marginally inferior to those from the aligned test using the overfitted model. Figure 6.33b illustrates a sample trajectory, showing how the UAV behaves under these conditions.

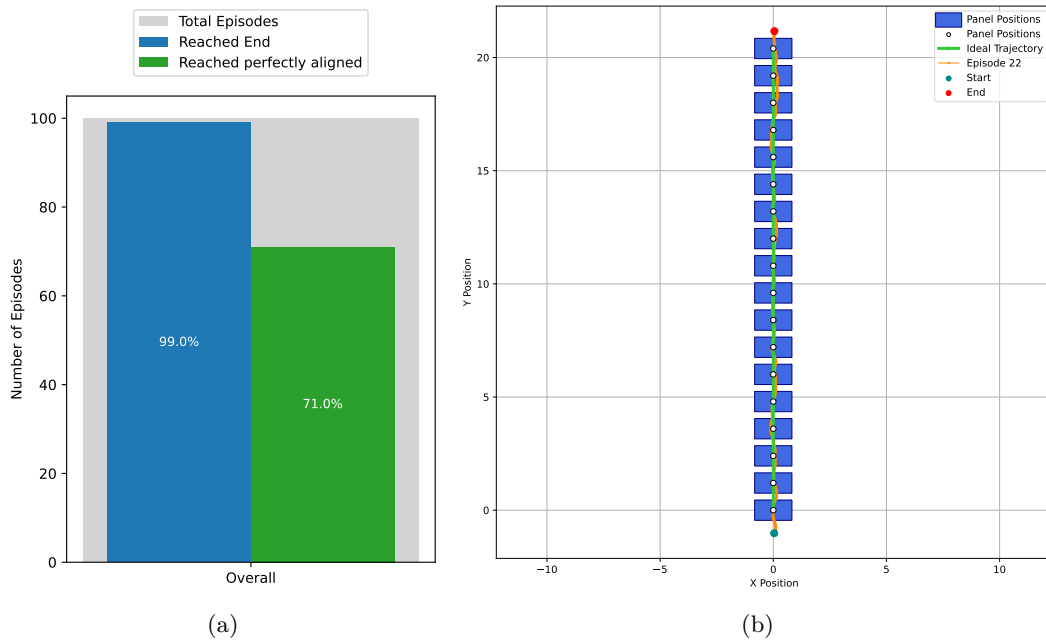


Figure 6.33: Overall reach-end behavior (a) and trajectory example (b) in the aligned test with the retrained model.

Figure 6.34 depicts the episode in which the RL agent in the rocky background did not reach the end. In general, it achieved an acceptable performance but was slightly worse than the overfitted model.

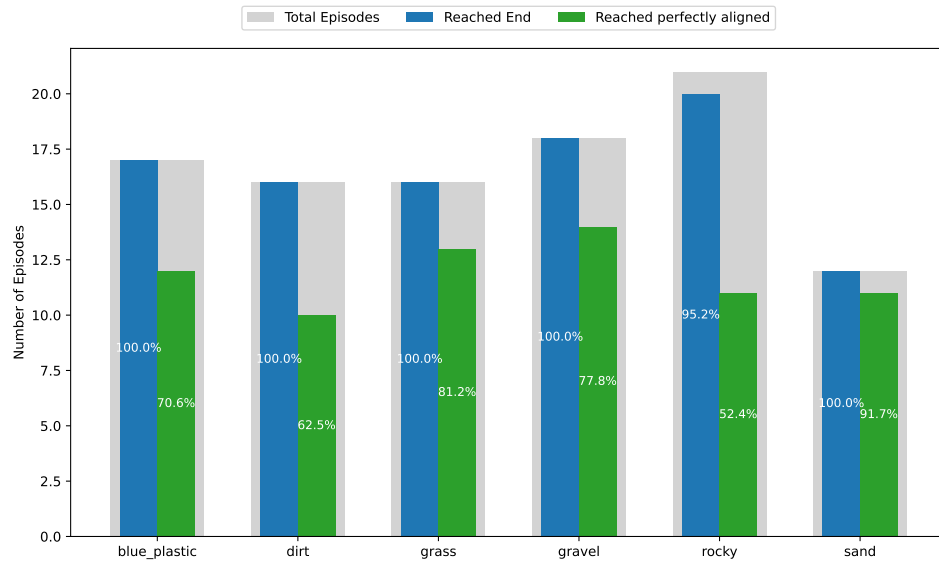


Figure 6.34: Reach-end behavior for each evaluated background in the aligned test with the retrained model.

Figure 6.35 illustrates an episode where the RL agent did not reach the end in the case of middle array. Other than that, there is no significant difference between panels.

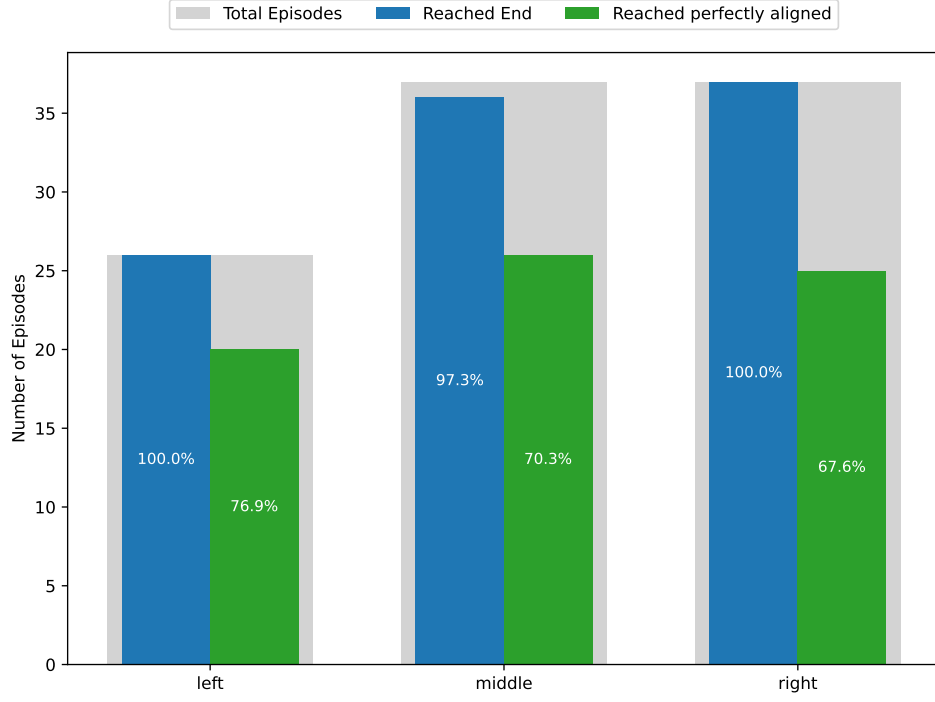


Figure 6.35: Reach-end behavior for each individual array in the aligned test with the retrained model.

The worst average yaw error ($\bar{e}_\psi$) and the worst average position error ($\bar{e}_d$) occurred with the blue plastic background, which is one of the simplest scenarios. This may indicate that, in this condition, the UAV occasionally failed to reach the end. Nevertheless, the agent achieved submetric performance (0.010 m) across all conditions, as reported in Table 6.10.

Table 6.10: Average Yaw and Position Errors with Standard Deviations for all the planes and arrays in the aligned test with the retrained model.

Surface/Array	$\bar{e}_\psi$ (°)	σ_{e_ψ} (°)	$\bar{e}_d$ (m)	σ_{e_d} (m)
By Plane				
Blue Plastic	4.2916	2.1349	0.0998	0.0361
Dirt	2.9583	1.1765	0.0763	0.0134
Grass	3.4474	2.1487	0.0851	0.0199
Gravel	3.4739	0.9466	0.0809	0.0219
Rocky	3.5642	1.6410	0.0888	0.0348
Sand	3.7315	1.1025	0.0881	0.0328
By Array				
Left	3.3895	1.1968	0.0792	0.0269
Middle	3.5831	1.5824	0.0841	0.0286
Right	3.6836	1.9723	0.0940	0.0287

6.5.3 Altitude Test

Figure 6.36a presents the overall results for the different altitudes. The agent successfully reached the end in 100 % of the episodes, with 62 % of them falling within the perfectly aligned bounds. Interestingly, the evaluation reported an average position error $\bar{e}_d$ of 0.0765 m and an average yaw error $\bar{e}_\psi$ of 3.509° . These results are better than those obtained in both the default test and the aligned test, suggesting that the RL agent performs better at higher altitudes than at the default altitude of 2.4 m.

It is important to note that the model was retrained with variable altitudes, rather than a fixed altitude as in the initial default training described in Chapter 6.2. Figure 6.36b illustrates a sample trajectory under these conditions.

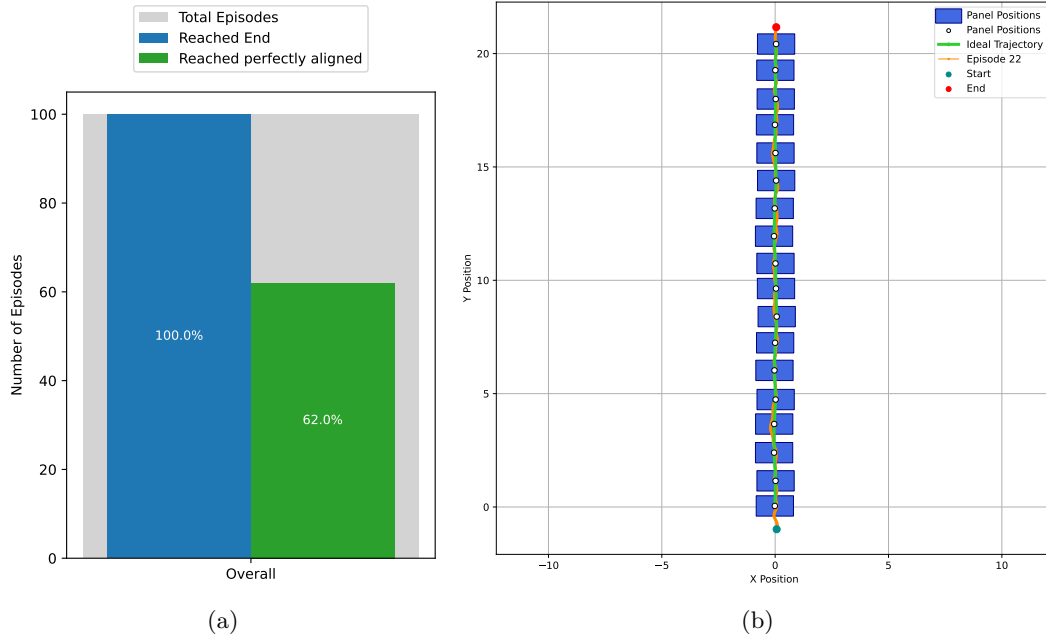


Figure 6.36: Overall reach-end behavior (a) and trajectory example (b) in the altitude test with the retrained model.

In Figure 6.37, the rocky background achieved the lowest perfectly aligned performance, with only 33.3 %, despite being well represented. Surprisingly, the grass background followed a similar trend, with only 43.8 % of perfectly aligned performance. On the other hand, the sand and blue plastic backgrounds exhibited excellent reach-end performance.

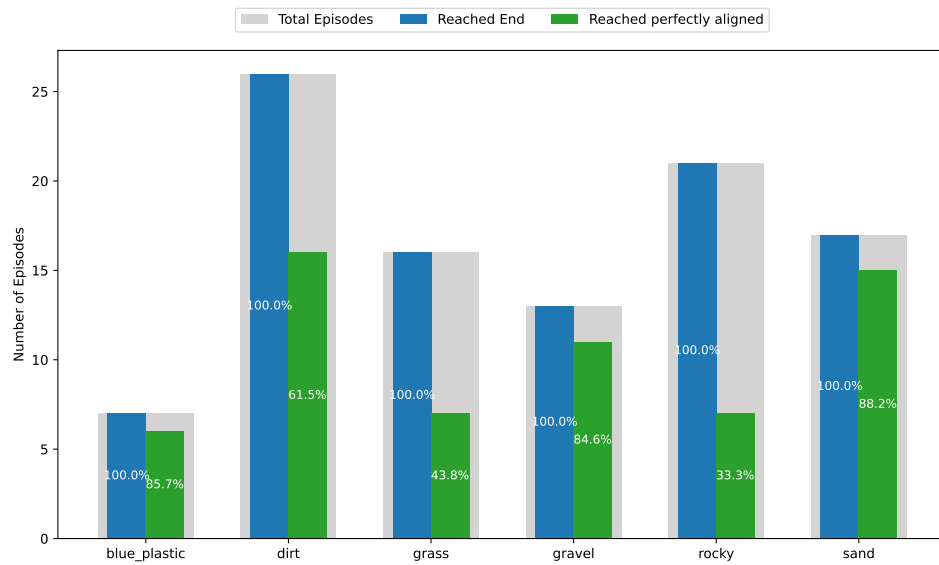


Figure 6.37: Reach-end behavior for each evaluated background in the altitude test with the retrained model.

Figure 6.38 shows that the middle array achieved the lowest, but not critically poor reach-end performance (approximately 52.6 %).

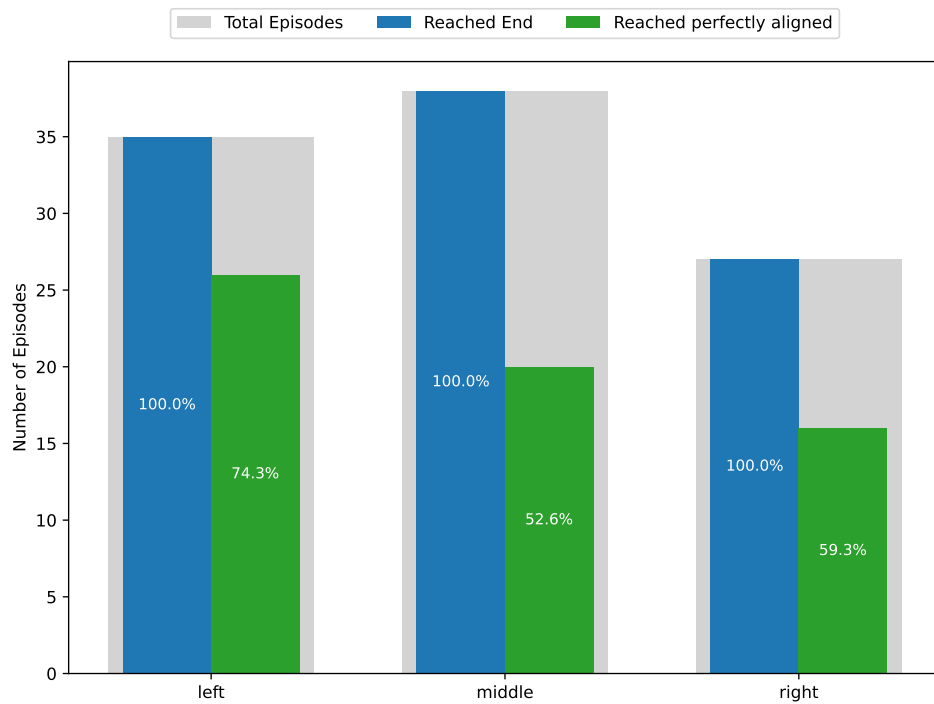


Figure 6.38: Reach-end behavior for each individual array in the altitude test with the retrained model.

This test included three distinct altitudes: 2.5 m, 2.7 m and 2.9 m. Figure 6.39 holds the corresponding reach-end performance graph. In contrast to the original model, the 2.7 m of altitude has the lowest reach-end performance. This behavior may result from a set of episodes with challenging sunlight conditions.

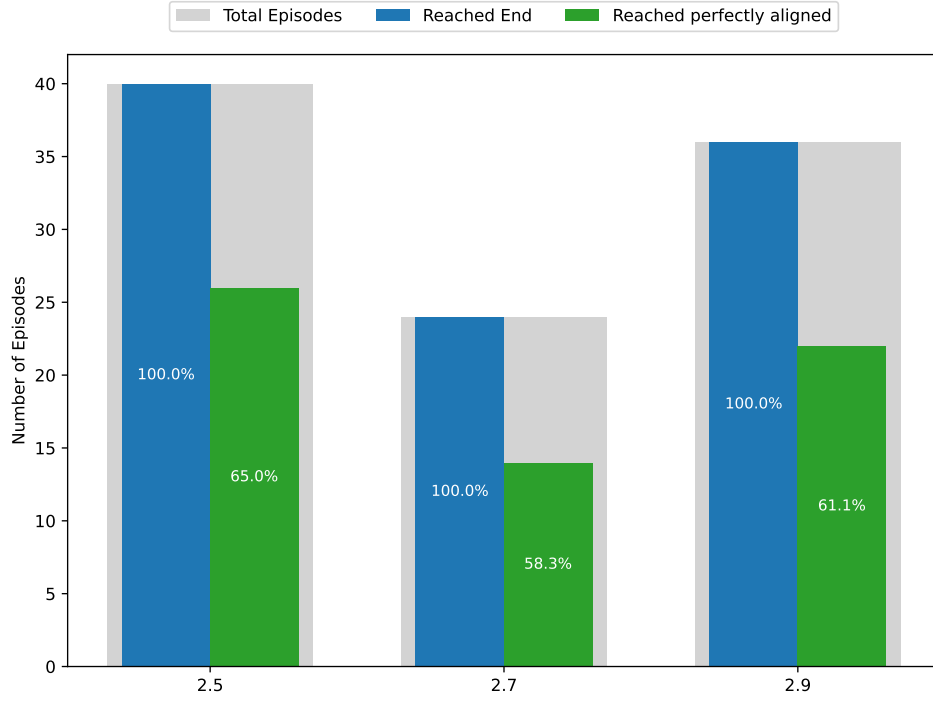


Figure 6.39: Reach-end behavior for each specific altitude (2.5 m, 2.7 m and 2.9 m) in the altitude test with the retrained model.

The statistics in Table 6.11 show that the grass background has the best average position state-error ($\bar{e}_d$) of around 0.0598 m and an incredibly low position standard deviation (σ_{e_d}) of 0.0071°. The gravel background displays the worst average yaw error of $\bar{e}_\psi$ and the biggest σ_{e_ψ} .

In terms of arrays and altitudes, there is no considerable difference between them.

Table 6.11: Average Yaw and Position Errors with Standard Deviations for all the planes, arrays and altitudes in the altitude test with the retrained model.

Surface/Array/Altitude	$\bar{e}_\psi$ (°)	σ_{e_ψ} (°)	$\bar{e}_d$ (m)	σ_{e_d} (m)
By Plane				
Blue Plastic	3.6611	1.1119	0.0849	0.0361
Dirt	3.4187	1.4997	0.0807	0.0345
Grass	2.8299	0.9561	0.0598	0.0071
Gravel	4.7450	3.4786	0.0744	0.0257
Rocky	4.1555	2.9424	0.0942	0.0709
Sand	2.6416	0.7733	0.0634	0.0101
By Array				
Left	3.5064	1.8405	0.0694	0.0203
Middle	3.6558	2.4148	0.0756	0.0313
Right	3.3562	2.1929	0.0882	0.0663
By Altitude				
2.5 m	3.7126	2.3997	0.0708	0.0225
2.7 m	3.4163	2.0627	0.0819	0.0379
2.9 m	3.3832	1.9400	0.0800	0.0570

6.5.4 Image Perturbation Test

This test is identical to the one conducted in Chapter 6.3.5, with all evaluations performed with randomized sunlight conditions at each episode.

There was a clear improvement in both the reach-end and reached perfectly aligned performances, jumping from a 79 % to 98 % and from 31 % to 40 %, respectively. This means that the new model is considerably more robust to overall perturbations independent of the sunlight, despite not being perfect, as seen in Figure 6.40a.

The overall state error decreased. The average position state error $\bar{e}_d$ was reduced to 0.1103 m, while the average orientation error $\bar{e}_\psi$ maintained a value of 5.3474°. This improvement is visible in the trajectory portrayed in Figure 6.40b, where, regardless of the oscillations caused by the perturbations, the UAV reached the end.

Regarding the background-specific reach-end performance, a performance gap is observed with the rocky background, which failed to reach the end in 2 out of 19 episodes, corresponding to approximately 89.5 % success. The gravel background achieved the highest perfectly aligned performance. Nonetheless, the performance with the different backgrounds, shown in Figure 6.41, is similar.

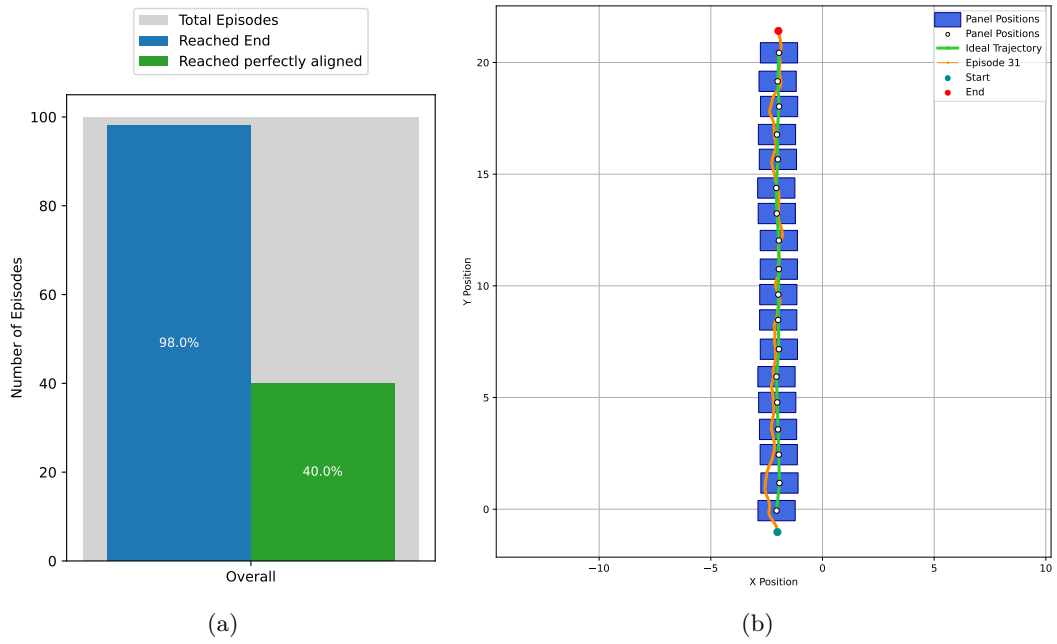


Figure 6.40: Overall reach-end behavior (a) and trajectory example (b) in the perturbation test with the retrained model.

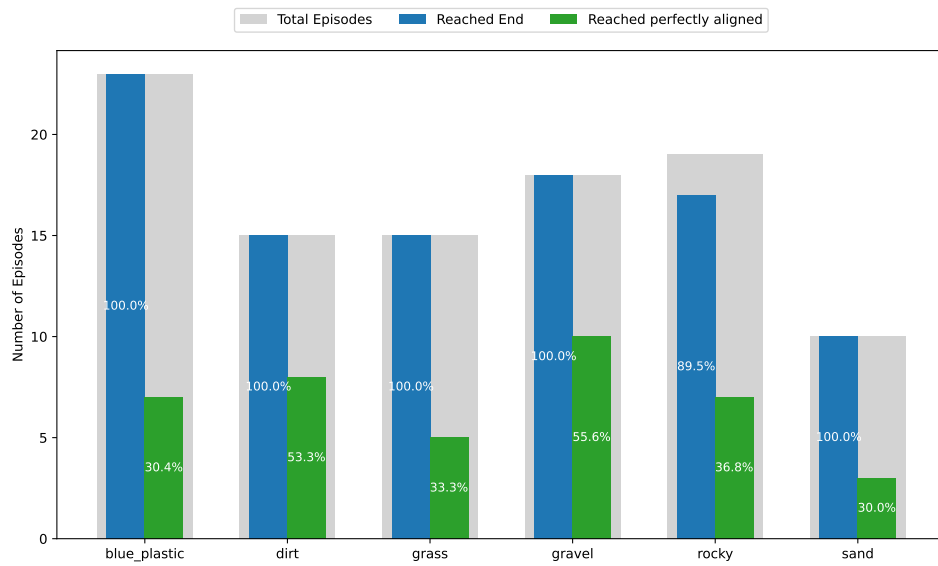


Figure 6.41: Reach-end behavior for each evaluated background in the perturbation test with the retrained model.

The analysis of Figure 6.42 indicates that the reach-end performance is largely consistent across the individual arrays, with only a slightly lower performance observed for the left array. The agent failed to reach the end once in the middle and right arrays.

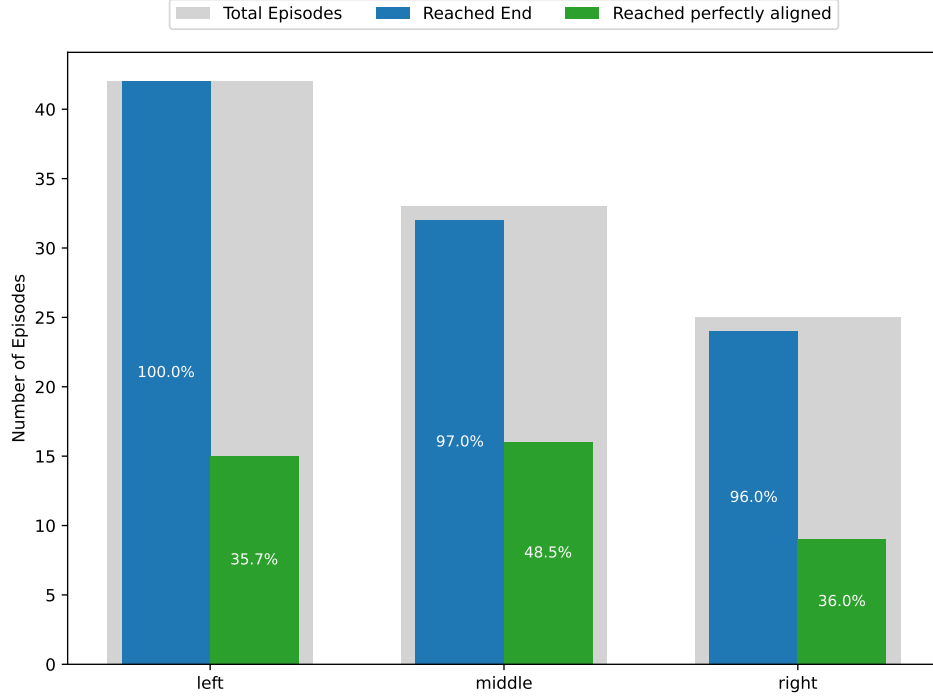


Figure 6.42: Reach-end behavior for each individual array in the perturbation test with the retrained model.

Finally, regarding the statistics of each category, the average position error $\bar{e}_d$ for each section is generally around 0.10 m, representing a significant improvement compared to the results in Chapter 6.3.5. The average yaw error $\bar{e}_\psi$ is around 5.00° , and the standard deviations are small, indicating a robust policy. These results are summarized in Table 6.12.

Table 6.12: Average Yaw and Position Errors with Standard Deviations for all the planes and arrays in the perturbation test with the retrained model.

Surface/Array	$\bar{e}_\psi$ (°)	σ_{e_ψ} (°)	$\bar{e}_d$ (m)	σ_{e_d} (m)
By Plane				
Blue Plastic	5.8103	1.6589	0.1084	0.0172
Dirt	5.0792	1.1787	0.1068	0.0222
Grass	5.1750	1.5375	0.1023	0.0155
Gravel	5.6747	0.9994	0.1040	0.0193
Rocky	5.3440	1.1211	0.1104	0.0202
Sand	5.3183	1.1759	0.1108	0.0205
By Array				
Left	5.2624	1.3758	0.1060	0.0186
Middle	5.2736	1.1636	0.1060	0.0194
Right	5.5876	1.3298	0.1103	0.0208

6.6 Summary

Reflecting on the findings and observations, several conclusions can be drawn. The architecture that combines past actions with visual input proved to be the most effective, achieving the best overall performance. This is consistent with theoretical expectations, as the agent can construct a better approximate belief state in a partially observed environment by using past observations.

Although this model initially demonstrated excellent performance during training, the first evaluation phase revealed a significant performance drop under varying sunlight conditions, indicating overfitting to the training environment.

To mitigate this issue, the model was retrained using regularization techniques and hyperparameter tuning, enabling the RL agent to learn effectively in an environment characterized by more complex domain randomization.

The retrained model successfully learned a robust policy across diverse conditions. However, despite improved generalisation, its accuracy remained below that of the original overfitted model. In this online *video*¹, a quick demonstration of the agent’s evolution from the retraining phase to the final evaluation tests can be observed.

Even with a deterministic policy, it remains difficult to explain the agent’s superior performance with certain backgrounds or arrays. The most pronounced performance variations are likely caused by rare or out-of-distribution sunlight conditions, which are inherently challenging to quantify. Additionally, the non-deterministic nature of the experimental conditions prevents direct comparison of results.

Overall, these results underscore the trade-off between robustness and accuracy in RL agents: strong generalization across variable environments may come at the expense of reduced performance under familiar, overfitted conditions.

¹<https://www.youtube.com/watch?v=5NPizSvRkqQ>

Furthermore, the choice of the SAC agent proved appropriate. However, its dependence on a replay buffer introduces challenges when training on low- to mid-range computers, due to the RAM required to store extensive experience data.

Chapter 7

Conclusions

This dissertation addressed the autonomous inspection of floating photovoltaic panel arrays using an *Unmanned Aerial Vehicle* (UAV). It defines the problem, outlines the system requirements and reviews *Reinforcement Learning* (RL) and key deep learning algorithms, with *Soft Actor-Critic* (SAC) emerging as the most promising candidate. The *State-of-the-art* (SOTA) analysis examines current methods and technologies. The related work section introduces advanced techniques to enhance robustness and efficiency in visual data processing. This study identified no robust autonomous AI-based solutions for navigating and inspecting floating solar installations, underlining the relevance of this dissertation.

7.1 Contributions

This work makes the following contributions:

- Proposed and evaluated two feature extractor architectures for SAC: a unimodal visual-only baseline model and a multimodal model combining visual input with past actions and velocities, resulting in the implementation of three models.
- Developed a modular and extensible pipeline for training and evaluating the models, enabling systematic comparison.
- Created a high-fidelity simulation environment, modeling photovoltaic arrays after the Alqueva reservoir farm to support realistic and diverse test scenarios.

7.2 Findings

The multimodal model, which combines visual input with past actions, achieved the highest accumulated rewards. Initial training revealed robustness issues, which were mitigated through sunlight randomization, feature extractor regularization, and hyperparameter tuning.

The final model met the evaluation criteria, demonstrating a trade-off between accuracy and robustness. The results also highlight the critical role of the replay buffer and domain randomization in achieving reliable performance.

7.3 Future Work

As for future work, beyond publishing a scientific paper to share the contributions of this work with the research community, the following steps are being considered:

- Test this solution in real-world environments to gain a better understanding of how the algorithm handles the well-known sim-to-real gap.
- Retrain the model for different photovoltaic-panel grid layouts. The floating solar farm in the Alqueva reservoir features a very specific layout, and it would be valuable to evaluate how the algorithm performs in other scenarios.
- Add the ability to adjust altitude (movement along the z -axis) according to the morphology of the terrain where the panels are installed.
- Compare the performance of the RL algorithm with other autopilots responsible for generating high-level velocity commands, such as PX4.
- Apply the proposed approach using more modern algorithms like *Dreamer-V3* and *MR. Q*.

References

- [1] G. Masson, A. V. Rechem, M. de l’Epine, and A. Jäger-Waldau, “Snapshot 2025,” IEA PVPS, Tech. Rep., Apr. 2025. [Cited on pages xi, 1, and 2]
- [2] U. drone, M. A. for Fast, and a. i. o. s. a. A. Fischer, “Aerial inspection of solar plants goes mainstream,” Jun. 2022. [Cited on page 1]
- [3] “Increasing solar inspection efficiency with drones.” [Cited on page 1]
- [4] O. I. Olayiwola and F. Camara, “Challenges and Opportunities for Autonomous UAV Inspection in Solar Photovoltaics: International Conference on Renewable Energy and Environment Engineering,” *E3S Web Conferences*, vol. 572, Sep. 2024. [Cited on pages 2 and 30]
- [5] “Is beyond visual line of sight (BVLOS) operation possible for flying drones with goggles (first-person view (FPV)) operation? | EASA,” <https://www.easa.europa.eu/en/faq/140038>. [Cited on page 2]
- [6] “Photovoltaic Power Plant - Renewables - Inspection - DJI Enterprise,” <https://enterprise.dji.com/inspection/photo>. [Cited on page 2]
- [7] D. F. Campos, A. Matos, and A. M. Pinto, “Multi-domain inspection of off-shore wind farms using an autonomous surface vehicle,” *SN Applied Sciences*, vol. 3, no. 4, p. 455, Apr. 2021. [Cited on page 4]
- [8] P. N. Leite, P. N. Pereira, J. M. Dionisio, and A. M. Pinto, “Hybrid underwater imaging for the tri-dimensional inspection of critical structural elements in offshore platforms,” *Ocean Engineering*, vol. 314, p. 119658, 2024. [Cited on page 4]
- [9] L. S. Pinho, T. D. Sousa, C. D. Pereira, and A. M. Pinto, “Anomaly Detection for PV Modules using Multi-modal Data Fusion in Aerial Inspections,” *IEEE Access*, 2025. [Cited on page 4]
- [10] R. S. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, 2nd ed., ser. Adaptive Computation and Machine Learning. Cambridge, Massachusetts London, England: The MIT Press, 2020. [Cited on pages vii, 7, 9, 10, 11, 12, 13, 14, 15, 16, and 17]

- [11] M. I. Pereira and A. M. Pinto, “Reinforcement learning based robot navigation using illegal actions for autonomous docking of surface vehicles in unknown environments,” *Engineering Applications of Artificial Intelligence*, vol. 133, p. 108506, Jul. 2024. [Cited on page 7]
- [12] M. Naeem, S. T. H. Rizvi, and A. Coronato, “A Gentle Introduction to Reinforcement Learning and its Application in Different Fields,” *IEEE Access*, vol. 8, pp. 209 320–209 344, 2020. [Cited on page 7]
- [13] E. L. Thorndike, “The Law of Effect,” *The American Journal of Psychology*, vol. 39, no. 1/4, pp. 212–222, 1927. [Cited on page 8]
- [14] M. E. Bitterman, “Classical Conditioning since Pavlov,” *Review of General Psychology*, vol. 10, no. 4, pp. 365–376, Dec. 2006. [Cited on page 8]
- [15] R. Bellman, *Dynamic Programming*. Princeton, NJ: Princeton Univ. Pr, 1984. [Cited on page 8]
- [16] A. L. Samuel, “Some Studies in Machine Learning Using the Game of Checkers,” *IBM Journal of Research and Development*, 1959. [Cited on page 8]
- [17] R. S. Sutton, “Learning to predict by the methods of temporal differences,” *Machine Learning*, vol. 3, no. 1, pp. 9–44, Aug. 1988. [Cited on page 8]
- [18] M. Igl, L. Zintgraf, T. A. Le, F. Wood, and S. Whiteson, “Deep Variational Reinforcement Learning for POMDPs,” in *Proceedings of the 35th International Conference on Machine Learning*. PMLR, Jul. 2018, pp. 2117–2126. [Cited on page 9]
- [19] “POMDPs: Who Needs them?” <https://www.pomdp.org/talks/who-needs-pomdps/index.html>. [Cited on page 10]
- [20] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, “Deterministic Policy Gradient Algorithms,” in *Proceedings of the 31st International Conference on Machine Learning*. PMLR, Jan. 2014, pp. 387–395, iSSN: 1938-7228. [Online]. Available: <https://proceedings.mlr.press/v32/silver14.html> [Cited on page 11]
- [21] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, “High-Dimensional Continuous Control Using Generalized Advantage Estimation,” Oct. 2018, arXiv:1506.02438 [cs]. [Online]. Available: <http://arxiv.org/abs/1506.02438> [Cited on pages 11 and 26]
- [22] D. Silver, “Lecture 2: Markov Decision Processes.” [Cited on pages 11 and 12]

-
- [23] “An Intuitive Guide to Reinforcement Learning – Robotic Sea Bass,” Aug. 2020. [Online]. Available: <https://roboticseabass.com/2020/08/02/an-intuitive-guide-to-reinforcement-learning/> [Cited on pages vii and 14]
- [24] D. Silver, “Lecture 5: Model-Free Control.” [Cited on pages 18 and 19]
- [25] D. E. Neves, L. Ishitani, and Z. K. G. do Patrocínio Júnior, “Advances and challenges in learning from experience replay,” *Artificial Intelligence Review*, vol. 58, no. 2, p. 54, Dec. 2024. [Cited on page 19]
- [26] S. Levine, A. Kumar, G. Tucker, and J. Fu, “Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems,” Nov. 2020. [Cited on page 19]
- [27] D. Silver, “Lecture 9: Exploration and Exploitation.” [Cited on page 20]
- [28] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, “Curiosity-driven Exploration by Self-supervised Prediction,” May 2017. [Cited on page 21]
- [29] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing Atari with Deep Reinforcement Learning,” Dec. 2013. [Cited on pages 21 and 22]
- [30] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Cited on pages 21, 22, 33, 35, 45, and 54]
- [31] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, “Rainbow: Combining Improvements in Deep Reinforcement Learning,” Oct. 2017. [Cited on page 22]
- [32] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” Jul. 2019. [Cited on page 23]
- [33] “Deep Deterministic Policy Gradient — Spinning Up documentation,” <https://spinningup.openai.com/en/latest/algorithms/ddpg.html>. [Cited on page 24]
- [34] S. Fujimoto, H. van Hoof, and D. Meger, “Addressing Function Approximation Error in Actor-Critic Methods,” Oct. 2018. [Cited on page 24]

- [35] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, “Trust Region Policy Optimization,” Apr. 2017. [Cited on page 25]
- [36] “Trust Region Policy Optimization — Spinning Up documentation,” <https://spinningup.openai.com/en/latest/algorithms/trpo.html>. [Cited on page 25]
- [37] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms,” Aug. 2017. [Cited on page 25]
- [38] “Proximal Policy Optimization — Spinning Up documentation,” <https://spinningup.openai.com/en/latest/algorithms/ppo.html>. [Cited on page 25]
- [39] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor,” Aug. 2018. [Cited on pages 26 and 55]
- [40] “Soft Actor-Critic — Spinning Up documentation,” <https://spinningup.openai.com/en/latest/algorithms/sac.html#relevant-papers>. [Cited on pages 26 and 27]
- [41] Y. Matsuo, Y. LeCun, M. Sahani, D. Precup, D. Silver, M. Sugiyama, E. Uchibe, and J. Morimoto, “Deep learning, reinforcement learning, and world models,” *Neural Networks*, vol. 152, pp. 267–275, Aug. 2022. [Cited on pages 27 and 28]
- [42] S. Arora and P. Doshi, “A survey of inverse reinforcement learning: Challenges, methods and progress,” *Artificial Intelligence*, vol. 297, p. 103500, Aug. 2021. [Cited on page 28]
- [43] M. Zare, P. M. Kebria, A. Khosravi, and S. Nahavandi, “A Survey of Imitation Learning: Algorithms, Recent Developments, and Challenges,” *IEEE Transactions on Cybernetics*, vol. 54, no. 12, pp. 7173–7186, Dec. 2024. [Cited on page 28]
- [44] L. Brunke, M. Greeff, A. W. Hall, Z. Yuan, S. Zhou, J. Panerati, and A. P. Schoellig, “Safe Learning in Robotics: From Learning-Based Control to Safe Reinforcement Learning,” *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 5, no. Volume 5, 2022, pp. 411–444, May 2022. [Cited on page 28]
- [45] C. Tessler, Y. Efroni, and S. Mannor, “Action Robust Reinforcement Learning and Applications in Continuous Control,” in *Proceedings of the 36th International Conference on Machine Learning*. PMLR, May 2019, pp. 6215–6224. [Cited on page 28]

-
- [46] M. Meribout, V. Kumar Tiwari, J. Pablo Peña Herrera, and A. Najeeb Mahfoudh Awadh Baobaid, “Solar panel inspection techniques and prospects,” *Measurement*, vol. 209, p. 112466, 2023. [Cited on pages 29 and 30]
- [47] A. Niccolai, F. Grimaccia, S. Leva, and P. Eleftheriadis, “Photovoltaic Plant Inspection by means of UAV: Current practices and future perspectives,” in *2021 IEEE International Conference on Environment and Electrical Engineering and 2021 IEEE Industrial and Commercial Power Systems Europe (EEEIC / I&CPS Europe)*, Sep. 2021, pp. 1–6. [Cited on page 30]
- [48] H. Choset and P. Pignon, “Coverage Path Planning: The Boustrophedon Cellular Decomposition,” in *Field and Service Robotics*, A. Zelinsky, Ed. London: Springer, 1998, pp. 203–209. [Cited on page 30]
- [49] A. Cardoso, D. Jurado-Rodríguez, A. López, M. I. Ramos, and J. M. Jurado, “Automated detection and tracking of photovoltaic modules from 3D remote sensing data,” *Applied Energy*, vol. 367, p. 123242, Aug. 2024. [Cited on pages vii and 30]
- [50] L. Morando, C. T. Recchiuto, J. Calla, P. Scuteri, and A. Sgorbissa, “Thermal and Visual Tracking of Photovoltaic Plants for Autonomous UAV Inspection,” *Drones*, vol. 6, no. 11, p. 347, Nov. 2022. [Cited on pages vii, 30, and 31]
- [51] S. Gallardo-Saavedra, L. Hernández-Callejo, and O. Duque-Perez, “Image Resolution Influence in Aerial Thermographic Inspections of Photovoltaic Plants,” *IEEE Transactions on Industrial Informatics*, vol. 14, no. 12, pp. 5678–5686, Dec. 2018. [Cited on page 30]
- [52] G. Roggi, A. Niccolai, F. Grimaccia, and M. Lovera, “A Computer Vision Line-Tracking Algorithm for Automatic UAV Photovoltaic Plants Monitoring Applications,” *Energies*, vol. 13, no. 4, p. 838, Jan. 2020. [Cited on pages vii, 31, and 32]
- [53] N. O’Mahony, S. Campbell, A. Carvalho, S. Harapanahalli, G. V. Hernandez, L. Krpalkova, D. Riordan, and J. Walsh, “Deep Learning vs. Traditional Computer Vision,” in *Advances in Computer Vision*, K. Arai and S. Kapoor, Eds. Cham: Springer International Publishing, 2020, pp. 128–144. [Cited on page 31]
- [54] T. Mu, Z. Li, S. W. Strzelecki, X. Yuan, Y. Yao, L. Liang, and H. Su, “When Should We Prefer State-to-Visual DAgger Over Visual Reinforcement Learning?” Dec. 2024. [Cited on page 32]
- [55] Y. Ajmera and S. P. Singh, “Autonomous UAV-based Target Search, Tracking and Following using Reinforcement Learning and YOLOFlow,” in *2020 IEEE*

- International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, Nov. 2020, pp. 15–20. [Cited on page 32]
- [56] N. Andrade, T. Ribeiro, J. Coelho, G. Lopes, and A. F. Ribeiro, “Combining YOLO and deep reinforcement learning for autonomous driving in public roadworks scenarios,” in *Proceedings of the 14th International Conference on Agents and Artificial Intelligence (ICAART 2022) – Volume 3*. SCITEPRESS, 2022. [Cited on page 32]
- [57] Y.-L. Chen, Y.-R. Cai, and M.-Y. Cheng, “Vision-Based Robotic Object Grasping—A Deep Reinforcement Learning Approach,” *Machines*, vol. 11, no. 2, p. 275, Feb. 2023. [Cited on page 32]
- [58] C. Finn, X. Y. Tan, Y. Duan, T. Darrell, S. Levine, and P. Abbeel, “Deep spatial autoencoders for visuomotor learning,” in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, May 2016, pp. 512–519. [Cited on page 32]
- [59] B. Prakash, M. Horton, N. R. Waytowich, W. D. Hairston, T. Oates, and T. Mohsenin, “On the use of Deep Autoencoders for Efficient Embedded Reinforcement Learning,” Mar. 2019. [Cited on page 32]
- [60] X. Wang, L. Lian, and S. X. Yu, “Unsupervised Visual Attention and Invariance for Reinforcement Learning,” Apr. 2021. [Cited on page 32]
- [61] F. S. Neves, R. M. Claro, and A. M. Pinto, “End-to-End Detection of a Landing Platform for Offshore UAVs Based on a Multimodal Early Fusion Approach,” *Sensors*, vol. 23, no. 5, p. 2434, Jan. 2023. [Cited on page 33]
- [62] A. Dionigi, S. Felicioni, M. Leomanni, and G. Costante, “D-VAT: End-to-End Visual Active Tracking for Micro Aerial Vehicles,” *IEEE Robotics and Automation Letters*, vol. 9, no. 6, pp. 5046–5053, Jun. 2024. [Cited on page 33]
- [63] J. Wang, Z. Yu, D. Zhou, J. Shi, and R. Deng, “Vision-Based Deep Reinforcement Learning of UAV Autonomous Navigation Using Privileged Information,” Dec. 2024. [Cited on page 33]
- [64] M. Laskin, A. Srinivas, and P. Abbeel, “CURL: Contrastive Unsupervised Representations for Reinforcement Learning,” in *Proceedings of the 37th International Conference on Machine Learning*. PMLR, Nov. 2020, pp. 5639–5650. [Cited on page 33]
- [65] D. Yarats, R. Fergus, A. Lazaric, and L. Pinto, “Mastering Visual Continuous Control: Improved Data-Augmented Reinforcement Learning,” Jul. 2021. [Cited on pages vii, 33, and 34]

-
- [66] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient Estimation of Word Representations in Vector Space,” Sep. 2013. [Cited on page 33]
- [67] J. Xing, L. Bauersfeld, Y. Song, C. Xing, and D. Scaramuzza, “Contrastive Learning for Enhancing Robust Scene Transfer in Vision-based Agile Flight,” Feb. 2024. [Cited on page 33]
- [68] S. Fujimoto, P. D’Oro, A. Zhang, Y. Tian, and M. Rabbat, “Towards General-Purpose Model-Free Reinforcement Learning,” Jan. 2025. [Cited on page 33]
- [69] J. Ortiz, A. Dedieu, W. Lehrach, S. Guntupalli, C. Wendelken, A. Humayun, G. Zhou, S. Swaminathan, M. Lázaro-Gredilla, and K. Murphy, “DMC-VB: A Benchmark for Representation Learning for Control with Visual Distractors,” Sep. 2024. [Cited on page 33]
- [70] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap, “Mastering diverse control tasks through world models,” *Nature*, vol. 640, no. 8059, pp. 647–653, Apr. 2025. [Cited on pages vii and 34]
- [71] A. Romero, A. Shenai, I. Geles, E. Aljalbout, and D. Scaramuzza, “Dream to Fly: Model-Based Reinforcement Learning for Vision-Based Drone Flight,” Jan. 2025. [Cited on page 34]
- [72] “Frame Skipping and Pre-Processing for Deep Q-Networks on Atari 2600 Games,” <https://danieltakeshi.github.io/2016/11/25/frame-skipping-and-preprocessing-for-deep-q-networks-on-atari-2600-games/>. [Cited on pages vii and 35]
- [73] O. Eberhard, M. Muehlebach, and C. Vernade, “Partially Observable Reinforcement Learning with Memory Traces,” in *Forty-Second International Conference on Machine Learning*, Jun. 2025. [Cited on page 35]
- [74] R. M. Claro, F. S. P. Neves, and A. M. G. Pinto, “A Multimodal Perception System for Precise Landing of UAVs in Offshore Environments,” *Journal of Field Robotics*, vol. 42, no. 5, pp. 2151–2172, 2025. [Cited on page 35]
- [75] F. S. Neves, L. M. Branco, M. I. Pereira, R. M. Claro, and A. M. Pinto, “A Multimodal Learning-based Approach for Autonomous Landing of UAV,” in *2024 20th IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications (MESA)*, Sep. 2024, pp. 1–8. [Cited on page 35]
- [76] J. Moos, K. Hansel, H. Abdulsamad, S. Stark, D. Clever, and J. Peters, “Robust Reinforcement Learning: A Review of Foundations and Recent Advances,” *Machine Learning and Knowledge Extraction*, vol. 4, no. 1, pp. 276–315, Mar. 2022. [Cited on page 35]

- [77] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, “Sim-to-Real Transfer of Robotic Control with Dynamics Randomization,” in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, May 2018, pp. 3803–3810. [Cited on page 36]
- [78] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, “Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World,” Mar. 2017. [Cited on pages vii and 36]
- [79] M. Xu, Z. Liu, P. Huang, W. Ding, Z. Cen, B. Li, and D. Zhao, “Trustworthy Reinforcement Learning Against Intrinsic Vulnerabilities: Robustness, Safety, and Generalizability,” Sep. 2022. [Cited on page 36]
- [80] S. Chen, G. Liu, Z. Zhou, K. Zhang, and J. Wang, “Robust Multi-Agent Reinforcement Learning Method Based on Adversarial Domain Randomization for Real-World Dual-UAV Cooperation,” *IEEE Transactions on Intelligent Vehicles*, vol. 9, no. 1, pp. 1615–1627, Jan. 2024. [Cited on page 36]
- [81] J. Xue, Z. Liu, G. Liu, Z. Zhou, K. Zhang, Y. Tang, and J. Wang, “Robust Wind-Resistant Hovering Control of Quadrotor UAVs Using Deep Reinforcement Learning,” *IEEE Transactions on Intelligent Vehicles*, pp. 1–10, 2023. [Cited on page 36]
- [82] D. Wada, S. Araujo-Estrada, and S. Windsor, “Sim-to-Real Transfer for Fixed-Wing Uncrewed Aerial Vehicle: Pitch Control by High-Fidelity Modelling and Domain Randomization,” *IEEE Robotics and Automation Letters*, vol. 7, no. 4, pp. 11 735–11 742, Oct. 2022. [Cited on page 36]
- [83] M. Igl, K. Ciosek, Y. Li, S. Tschitschek, C. Zhang, S. Devlin, and K. Hofmann, “Generalization in Reinforcement Learning with Selective Noise Injection and Information Bottleneck,” Oct. 2019. [Cited on page 37]
- [84] A. Villanueva and A. Fajardo, “Deep Reinforcement Learning with Noise Injection for UAV Path Planning,” in *2019 IEEE 6th International Conference on Engineering Technologies and Applied Sciences (ICETAS)*, Dec. 2019, pp. 1–6. [Cited on page 37]
- [85] B. P. Duisterhof, S. Krishnan, J. J. Cruz, C. R. Banbury, W. Fu, A. Faust, G. C. H. E. de Croon, and V. J. Reddi, “Learning to Seek: Autonomous Source Seeking with Deep Reinforcement Learning Onboard a Nano Drone Microcontroller,” Jan. 2021. [Cited on page 37]
- [86] L. Pinto, J. Davidson, R. Sukthankar, and A. Gupta, “Robust Adversarial Reinforcement Learning,” in *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*, 2017. [Cited on page 37]

-
- [87] K. Wan, X. Gao, Z. Hu, and G. Wu, “Robust Motion Control for UAV in Dynamic Uncertain Environments Using Deep Reinforcement Learning,” *Remote Sensing*, vol. 12, no. 4, p. 640, Jan. 2020. [Cited on page 38]
- [88] H.-L. Hsu, H. Meng, S. Luo, J. Dong, V. Tarokh, and M. Pajic, “REFORMA: Robust REinFORceMent Learning via Adaptive Adversary for Drones Flying under Disturbances,” in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, May 2024, pp. 5169–5175. [Cited on page 38]
- [89] A. M. Deshpande, A. A. Minai, and M. Kumar, “Robust Deep Reinforcement Learning for Quadcopter Control,” *IFAC-PapersOnLine*, vol. 54, no. 20, pp. 90–95, Jan. 2021. [Cited on page 38]
- [90] ArduPilot. ArduPilot. ArduPilot.org. [Online]. Available: <https://ardupilot.org> [Cited on page 49]
- [91] Open Source Autopilot for Drones. PX4 Autopilot. [Online]. Available: <https://px4.io/> [Cited on page 49]
- [92] “ROS: Home,” <https://www.ros.org/>. [Cited on page 49]
- [93] “Isaac Sim,” <https://developer.nvidia.com/isaac/sim>. [Cited on page 49]
- [94] “Home - AirSim,” <https://microsoft.github.io/AirSim/>. [Cited on page 49]
- [95] “Gazebo,” <https://gazebo.org/home>. [Cited on page 49]
- [96] M. Jacinto, J. Pinto, J. Patrikar, J. Keller, R. Cunha, S. Scherer, and A. Pascoal, “Pegasus Simulator: An Isaac Sim Framework for Multiple Aerial Vehicles Simulation,” in *2024 International Conference on Unmanned Aircraft Systems (ICUAS)*, Jun. 2024, pp. 917–922. [Cited on page 50]
- [97] B. Studio, “Blender Studio,” <https://studio.blender.org/welcome/>. [Cited on page 50]
- [98] “Solar panel pack,” <https://www.fab.com/listings/bc42a1f1-e6d2-4815-919b-ae08b65f2af0>. [Cited on page 50]
- [99] “Premium PBR Textures for photorealistic 3D,” <https://www.poliigon.com/textures>. [Cited on page 51]
- [100] “Troubleshooting — Gazebo harmonic documentation,” <https://gazebo.org/docs/harmonic/troubleshooting/>. [Cited on page 52]
- [101] “SDFormat Specification,” <http://sdformat.org/spec?elem=material>. [Cited on page 52]

-
- [102] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, “Stable-baselines3: Reliable reinforcement learning implementations,” *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: <http://jmlr.org/papers/v22/20-1364.html> [Cited on page 54]
- [103] “SAC — Stable Baselines3 2.7.1a3 documentation,” <https://stable-baselines3.readthedocs.io/en/master/modules/sac.html>. [Cited on pages 54 and 95]
- [104] “Policy Networks — Stable Baselines3 2.7.1a0 documentation,” https://stable-baselines3.readthedocs.io/en/master/guide/custom_policy.html. [Cited on pages 54 and 55]
- [105] “Gymnasium Documentation,” <https://gymnasium.farama.org/index.html>. [Cited on page 55]
- [106] “Using Custom Environments — Stable Baselines3 2.7.1a0 documentation,” https://stable-baselines3.readthedocs.io/en/master/guide/custom_env.html. [Cited on page 55]
- [107] “Mavlink/mavros,” MAVLink Micro Air Vehicle Protocol, Sep. 2025. [Cited on page 57]
- [108] “Using Pymavlink Libraries (mavgen) | MAVLink Guide,” https://mavlink.io/en/mavgen_python/. [Cited on page 58]
- [109] “Gimbal Protocol (v2) | MAVLink Guide,” https://mavlink.io/en/services/gimbal_v2.html. [Cited on page 58]
- [110] “Albumentations: Fast and flexible image augmentations,” <https://albumentations.ai/>. [Cited on page 60]
- [111] A. Ilyas, L. Engstrom, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry, “A Closer Look at Deep Policy Gradients,” May 2020. [Cited on page 69]
- [112] T. Salimans and D. P. Kingma, “Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks,” Jun. 2016. [Cited on page 92]
- [113] Y. Wu and K. He, “Group Normalization,” Jun. 2018. [Cited on page 92]
- [114] C. Lyle, Z. Zheng, K. Khetarpal, J. Martens, H. van Hasselt, R. Pascanu, and W. Dabney, “Normalization and effective learning rates in reinforcement learning,” in *Advances in Neural Information Processing Systems (NeurIPS 2024) – Main Conference Track*, 2024. [Cited on page 92]

Appendix A

Appendix

A.1 Feature Extractor Architectures in Python/Pytorch

```
1  class NatureCNN(BaseFeaturesExtractor):
2      """
3      CNN from DQN Nature paper:
4      Mnih, Volodymyr, et al.
5      "Human-level control through deep reinforcement learning."
6      Nature 518.7540 (2015): 529-533.
7      """
8      def __init__(
9          self,
10         observation_space: gym.Space,
11         features_dim: int = 512,
12         normalized_image: bool = False,
13     ) -> None:
14         assert isinstance(observation_space, spaces.Box)
15         super().__init__(observation_space, features_dim)
16         # We assume CxHxW images (channels first)
17         # Re-ordering will be done by pre-preprocessing or wrapper
18         assert is_image_space(observation_space, check_channels=
19                               False, normalized_image
20                               =normalized_image)
21
22         n_input_channels = observation_space.shape[0]
23         self.cnn = nn.Sequential(
```

```

21         nn.Conv2d(n_input_channels, 32, kernel_size=8, stride=
22                     4, padding=0),
23         nn.ReLU(),
24         nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=0),
25         nn.ReLU(),
26         nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=0),
27         nn.ReLU(),
28         nn.Flatten(),
29     )
30
31     # Compute shape by doing one forward pass
32     with th.no_grad():
33         n_flatten = self.cnn(th.as_tensor(observation_space.
34                                         sample()[None]).
35                               float()).shape[1]
36
37     self.linear = nn.Sequential(nn.Linear(n_flatten,
38                                         features_dim), nn.ReLU
39                                 ())
40
41     def forward(self, observations: th.Tensor) -> th.Tensor:
42         return self.linear(self.cnn(observations))

```

Code Listing A.1: Default CNN feature extractor.

```

1  class ImagePlusVelocityStackedExtractor(BaseFeaturesExtractor):
2      def __init__(
3          self,
4          observation_space: spaces.Dict,
5          img_dim: int = 512,
6          vel_dim: int = 64,
7          vel_hidden: int = 128,      # size of the MLP hidden layer
8      ):
9          # total feature size = img_dim + vel_dim
10         super().__init__(observation_space, features_dim=img_dim +
11                           vel_dim)
12
13         # --- Image branch (unchanged) ---
14         img_space = observation_space.spaces["image"]
15         n_channels = img_space.shape[0]
16
17         self.cnn = nn.Sequential(
18             nn.Conv2d(n_channels, 32, 8, stride=4), nn.ReLU(),
19             nn.ReLU(),
20             nn.Conv2d(32, 64, 4, stride=2), nn.ReLU(),
21             nn.ReLU(),
22             nn.Conv2d(64, 64, 3, stride=1), nn.ReLU(),
23             nn.ReLU(),

```

```

23         nn.Flatten(),
24     )
25
26     with torch.no_grad():
27         dummy = torch.as_tensor(img_space.sample()[None]).
28                                     float()
29         n_flatten = self.cnn(dummy).shape[1]
30     self.img_proj = nn.Sequential(
31         nn.Linear(n_flatten, img_dim),
32         nn.ReLU(),
33     )
34
35     # --- Velocity branch: MLP on stacked velocities ---
36     vel_space = observation_space.spaces["stacked_velocities"]
37
38     vel_stack = vel_space.shape[0] # number of time steps
39     vel_dim_per_step = vel_space.shape[1] # 2 (velocity
40                                         components per step)
41     stacked_dim = vel_stack * vel_dim_per_step
42
43     self.vel_mlp = nn.Sequential(
44         nn.Linear(stacked_dim, vel_hidden),
45         nn.ReLU(),
46         nn.Linear(vel_hidden, vel_dim),
47         nn.ReLU(),
48     )
49
50     def forward(self, obs_dict):
51         # Image features
52         img = obs_dict["image"].float() # (N, C, H, W)
53         img_feat = self.img_proj(self.cnn(img)) # (N, img_dim)
54
55         # Velocity features
56         sv = obs_dict["stacked_velocities"].float()
57         sv = sv.view(sv.shape[0], -1)
58         vel_feat = self.vel_mlp(sv) # (N, vel_dim)
59
60         # concatenate and return
61         return torch.cat([img_feat, vel_feat], dim=1)

```

Code Listing A.2: Costum feature extractor with stacked actions/velocities.

```

1 class ImagePlusVelocityStackedExtractor(BaseFeaturesExtractor):
2     def __init__(
3         self,
4         observation_space: spaces.Dict,
5         img_dim: int = 512,

```

```

6         vel_dim: int = 64,
7         vel_hidden: int = 128,          # size of the MLP hidden layer
8     ):
9         # total feature size = img_dim + vel_dim
10        super().__init__(observation_space, features_dim=img_dim +
11                           vel_dim)
12
13        # --- Image branch (now with group and layer norm) ---
14        img_space = observation_space.spaces["image"]
15        n_channels = img_space.shape[0]
16
17        self.cnn = nn.Sequential(
18            nn.Conv2d(n_channels, 32, 8, stride=4), nn.ReLU(),
19            nn.GroupNorm(num_groups=8, num_channels=32),
20            nn.ReLU(),
21
22            nn.Conv2d(32, 64, 4, stride=2), nn.ReLU(),
23            nn.GroupNorm(num_groups=8, num_channels=64),
24            nn.ReLU(),
25
26            nn.Conv2d(64, 64, 3, stride=1), nn.ReLU(),
27            nn.GroupNorm(num_groups=8, num_channels=64),
28            nn.ReLU(),
29
30            nn.Flatten(),
31        )
32
33        with torch.no_grad():
34            dummy = torch.as_tensor(img_space.sample()[None]).
35                           float()
36
37            n_flatten = self.cnn(dummy).shape[1]
38        self.img_proj = nn.Sequential(
39            nn.Linear(n_flatten, img_dim),
40            nn.LayerNorm(img_dim),
41            nn.ReLU(),
42        )
43
44        # --- Velocity branch: MLP on stacked velocities ---
45        vel_space = observation_space.spaces["stacked_velocities"]
46        vel_stack = vel_space.shape[0] # number of time steps
47        vel_dim_per_step = vel_space.shape[1] # 2 (velocity
48                                           components per step)
49
50        stacked_dim = vel_stack * vel_dim_per_step
51
52        self.vel_mlp = nn.Sequential(
53            nn.Linear(stacked_dim, vel_hidden),
54            nn.LayerNorm(vel_hidden),
55            nn.ReLU(),
56            nn.Linear(vel_hidden, vel_dim),

```

```

52         nn.LayerNorm(vel_dim),
53         nn.ReLU(),
54     )
55
56     def forward(self, obs_dict):
57         # Image features
58         img = obs_dict["image"].float()           # (N, C, H, W)
59         img_feat = self.img_proj(self.cnn(img))    # (N, img_dim)
60
61         # Velocity features
62         sv = obs_dict["stacked_velocities"].float()
63         sv = sv.view(sv.shape[0], -1)
64         vel_feat = self.vel_mlp(sv)                # (N, vel_dim)
65
66         # concatenate and return
67         return torch.cat([img_feat, vel_feat], dim=1)

```

Code Listing A.3: Costum feature extractor with stacked actions/velocities and improved regularization.

A.2 YAML Configuration Examples

```

1  # config/training_config.yaml
2  training:
3
4      agent: "SAC"
5
6      total_timesteps: 300000
7      eval_freq: 10000    #eval at each x steps
8      n_eval_episodes: 3
9      eval_log: "/my/location/for/rl_eval_logs/"
10     train_log: "/my/location/for/rl_train_logs/"
11     save_freq: 25000    #save the training model each x steps
12     save_path: "/my/location/for/saving_model/"
13     save_best_path: "/my/location/for/saving_best_model/"
14     tensorboard_log: "/my/location/for/tensorboard_logs/"
15
16     SAC:
17         learning_rate: 0.0001
18         buffer_size: 800000 #100000, best 500000 with random light
19         batch_size: 256
20         tau: 0.01          #target network soft update
21         gamma: 0.99        #discount-factor
22         learning_starts: 5000 #2000
23         train_freq: 1       #train the agent at each update
24         gradient_steps: 1   #number of mini-batches updates

```

```

25     entropy_fixed: False
26     entropy_start_value: 1.0 #start higher at start for
                                   exploration
27
28 environment:
29     render: True
30     save_trajectory: False
31     info_file: "/home/joao/test_ws/src/rl_agent/config/
                                   episode_info_temp.json"
32     kill_sim_script: "/home/joao/test_ws/src/rl_agent/script/end_sim
                                   .sh"
33     enable_sim_script: "/home/joao/test_ws/src/rl_agent/script/
                                   start_sim.sh"
34
35     max_episode_steps: 300
36     step_timeout: 0.1
37
38     alternated: False
39     image_augmentation: False
40     display_augmentation: False
41
42     agent_info:
43         pose_ros2_topic: "/odometry"
44         ready_ros2_topic: "/uav_ready"
45
46     observations:
47         type: "combined" #image or combined
48
49     image:
50         display: True
51         ros2_topic: "camera/image"
52         frame_read_buffer: 20 #30 ou 12 #20
53         frame_skipping: 4 #4 ou 3 #Frames that jumps the frame
                                   stack
54
55         grayscale: True
56         blur_intensity: [5, 5] #introduce a bias for the CNN
57         resize_dim: [96, 72] #4:3
58
59     combined:
60         image:
61             display: True
62             ros2_topic: "camera/image"
63             frame_read_buffer: 30
64             frame_skipping: 4 #Frames that jump the frame stack
65             grayscale: True
66             blur_intensity: [7, 7] #introduce a bias for the CNN
67             resize_dim: [96, 72] #4:3
68         velocity:
69             linear_enabled: True

```

```

69         angular_enabled: True
70         stacked: True
71         embedding_dim: 32
72     pos_error:
73         enabled: False
74         embedding_dim: 16
75
76     actions:
77         #On body frame
78         ros2_topic: "/mavros/setpoint_raw/local"
79         #vy, w
80         size: 2
81         lower_bound: [-0.5, -0.3]
82         upper_bound: [0.5, 0.3]
83
84     yaw_max_error: 25.0 #degrees
85     pos_max_error: 1.0 #meters
86
87     reward: #sum of 'weight' should be 1
88     yaw:
89         std_dev: 0.10
90         weight: 0.35
91
92     position:
93         std_dev: 0.25
94         weight: 0.40
95
96     smoothness:
97         weight: 0.15
98         action_importance: [0.5, 0.5] #importance of smoothness for
                                         each action
99         max_diff: [2.0, 0.6] #for normalization
100
101     end_reaching:
102         reward: 10
103         threshold_pos: 0.25 #meters
104         threshold_orient: 3 #degrees
105
106     limit_penalty: -10

```

Code Listing A.4: YAML configuration for the final SAC training.

```

1 possible_planes:
2   - "grass"
3   - "sand"
4   - "dirt"
5   - "gravel"
6   - "blue_plastic"

```

```

7
8 base_poses:
9   start_mission:
10     - [1.0, 0]
11     - [1.0, 2]
12     - [1.0, 4]
13
14   end_mission:
15     - [-11.8, 0]
16     - [-11.8, 2]
17     - [-11.8, 4]
18
19   panels:
20     - [0.0, 0.0, 0.1, 0, -0.3, 0.0]
21     - [-1.2, 0.0, 0.1, 0, -0.3, 3.1415]
22     - [0.0, 2.0, 0.1, 0, -0.3, 0.0]
23     - [-1.2, 2.0, 0.1, 0, -0.3, 3.1415]
24     - [0.0, 4.0, 0.1, 0, -0.3, 0.0]
25     - [-1.2, 4.0, 0.1, 0, -0.3, 3.1415]
26     # And so on ...

```

Code Listing A.5: YAML file example used for loading the Gazebo simulation.

A.3 Reward Values Computation

```

1 ##### FOR POSITION ERROR #####
2 "It is created a polyline between the panel centers."
3 "The error becomes the short distance between the agent and the
   polyline."
4
5 def point_segment_distance(P, A, B):
6     AP = P - A
7     AB = B - A
8     ab2 = AB.dot(AB)
9     # handle degenerate segment
10    if ab2 == 0:
11        return np.linalg.norm(AP), A, 0.0
12
13    # projection parameter of P onto the infinite line AB
14    t = AP.dot(AB) / ab2
15    # clamp to segment
16    t_clamped = np.clip(t, 0.0, 1.0)
17    closest = A + t_clamped * AB
18    dist = np.linalg.norm(P - closest)
19    return dist, closest, t_clamped

```



```

20
21 def compute_position_error(P, verts):
22     P = np.asarray(P, dtype=float)
23     verts = [np.asarray(v, dtype=float) for v in verts]
24     best = (np.inf, None, None) # (min_dist, closest_point,
                                # segment_index)
25
26     #Go trough the polyline
27     for i in range(len(verts)-1):
28         A, B = verts[i], verts[i+1]
29         dist, closest, t = point_segment_distance(P, A, B)
30         if dist < best[0]:
31             best = (dist, closest, i)
32
33     min_dist = best[0] #return only the distance
34
35     return min_dist
36
37 ##### FOR YAW ERROR #####
38 "It is computed the error by wrapping the angle difference"
39 "yaw = 0 if its perfectly aligned with X axis of the world"
40
41
42 def compute_yaw_error(panel_yaw, agent_yaw, rads = False):
43
44     diff = agent_yaw - panel_yaw
45
46     #wrap around [0, 360], abs for absolute error, which will be
47     #positive
48     wrapped_diff = abs((diff + 180) % 360 - 180)
49
50     #print(f"wrapped_diff: {wrapped_diff}")
51
52     if rads is True:
53         return float(np.deg2rad(wrapped_diff))
54     else:
55         return float(wrapped_diff)
56
57 def compute_errors(agent_yaw: float, agent_xy: np.ndarray,
58                   panels_yaw: float,
59                   panels_positions: np.ndarray):
60
61     #Compute errors by calling the main functions:
62     yaw_error_deg = compute_yaw_error(panels_yaw, agent_yaw, rads

```

```
63     return yaw_error_deg, position_error_m
```

Code Listing A.6: Python functions to compute yaw and position errors.

```
1  def gaussian_function(x, std_dev, mu=0):
2      f = 1 * np.exp(-((x-mu)**2)/(2*std_dev**2))
3      return f
4
5  def compute_reward(yaw_error_deg: float, position_error_m: float,
6                     current_action: np.ndarray,
7                     previous_action: np.ndarray,
8                     agent_vel_state: np.ndarray,
9                     progress: float, cfg: Dict[str,
10                      Any]):
11
12      #yaw cfg
13      yaw_sigma = cfg['yaw']['std_dev']
14      yaw_weight = cfg['yaw']['weight']
15
16      #pos cfg
17      pos_sigma = cfg['position']['std_dev']
18      pos_weight = cfg['position']['weight']
19
20      #smooth cfg
21      smoothness_weight = float(cfg['smoothness']['weight'])
22      action_importance = np.array(cfg['smoothness']['
23          action_importance'])
24      action_importance_matrix = np.diag(action_importance)
25      max_action_diff = np.array(cfg['smoothness']['max_diff'])
26
27      action_diff = (current_action - previous_action)**2
28      normalized_diff = 1.0 - (action_diff / max_action_diff) #Where
29          1 is smoother, and 0 not
30      r_smoothness = smoothness_weight * (action_importance_matrix *
31          normalized_diff).sum()
32
33      #Pass trough the gaussian functions:
34      r_yaw = yaw_weight * gaussian_function(x=np.deg2rad(
35          yaw_error_deg), std_dev =
36          yaw_sigma)
37      r_pos = pos_weight * gaussian_function(x=position_error_m,
38          std_dev = pos_sigma)
39
40      total_reward = r_yaw + r_pos + r_progress + r_smoothness
41
42      return total_reward
```

Code Listing A.7: Compute reward based on the GT errors.

A.4 Training Loop Step and Reset Functions

```

1  class GazeboRLEnv(gym.Env):
2      """
3      Custom Gymnasium environment for RL training with Gazebo and
4          ROS2
5      """
6      def __init__(self, ros2_interface: ROS2Interface, config: Dict
7          [str, Any], policy_type:
8          str):
9
10         super().__init__()
11
12         #Initializations ...
13
14     def reset(self, seed=None, options=None) -> Tuple[np.ndarray,
15         Dict[str, Any]]:
16
17         """Reset the environment"""
18         super().reset(seed=seed)
19
20         # Reset Gazebo simulation
21         self._reset_simulation()
22
23         # Update the episode data
24         self.ros2_interface.get_episode_info(override_file_check =
25             False)
26
27         # Wait for stable state
28         while self.ros2_interface.agent_ready is False:
29             time.sleep(0.01)
30
31         #-----
32         # Get initial observation to prevent NaNs
33         obs = self._get_observation()
34
35         #-----
36         #Initialize trajectory files if needed
37         if self.save_trajectory:
38             self._initialize_files()
39
40         #-----
41
42         info = self._get_info()
43
44         return obs, {}

```

```

37 def step(self, action: np.ndarray) -> Tuple[np.ndarray, float,
38                                           bool, bool, Dict[str, Any]]:
39     """Execute one step in the environment"""
40
41     # Scale action to actual robot limits
42     self.action_buffer.append(action)
43     self.current_action = self._scale_action(action)
44
45     # Send action to robot
46     self.ros2_interface.send_action(self.current_action)
47
48     # Wait for action to take effect
49     time.sleep(self.step_timeout)
50
51     #-----
52     # Get the current observation
53     obs = self._get_observation()
54
55     # Calculate reward from observations
56     reward = self._calculate_reward()
57     #-----
58
59     # Check if episode is done
60     self.terminated, extra_final_reward=self._is_terminated()
61
62     #-----
63     #Save trajectory if needed
64     if self.save_trajectory:
65         self._save_trajectory_csv()
66
67         if self.terminated:
68             self._save_final_result()
69     #-----
70
71     # Add the next step
72     self.current_step += 1
73
74     truncated = self.current_step >= self.max_episode_steps
75
76     #Penalty is also applied if truncated
77     if truncated:
78         reward += self.limit_penalty
79         print(f"Maximum desired training steps reached.
80               Penalty: {self.
81                     limit_penalty}")
82     else:

```

```
81         #Reward extra if reaches the end, or penalize it if  
            wont  
82         reward += extra_final_reward  
83  
84         # Update reward and previous actions  
85         self.episode_reward += reward  
86         self.previous_action = self.current_action  
87  
88         info = self._get_info()  
89  
90         return obs, reward, self.terminated, truncated, info
```

Code Listing A.8: Simplified implementation of the step and reset function within the Gymnasium environment.

