



Stability guarantees for learning based effort control in rigid robotics manipulators

Foundations for Stable Effort Control in Rigid Robotics: Validation and Future Work

ME51035: ME-BMD MSc Thesis

Rick Staa

Stability guarantees for learning based effort control in rigid robotics manipulators

Foundations for Stable Effort Control in Rigid
Robotics: Validation and Future Work

by

Rick Staa

Student Name	Student Number
Rick Staa	4511328

Main Instructor: Prof.dr.ir. M. (Martijn) Wisse
Project Duration: May, 2023 - September, 2024
Faculty: Cognitive Robotics, Delft

Cover: Picture of Franka Research 3 robot from <https://www.franka.de/>
Style: TU Delft Report Style, with modifications by Daan Zwaneveld

Abstract

This thesis investigates the stability and robustness of the Lyapunov Actor-Critic (LAC) algorithm in comparison to the widely-used Soft Actor-Critic (SAC) algorithm. Motivated by the need for reliable and robust control systems capable of operating in dynamic and unpredictable environments, the research initially aimed to explore LAC's performance in an effort-controlled robotic manipulation task. However, the focus shifted to a reproduction study of Han et al.[9] after identifying key discrepancies in the implementation details, hyperparameters, and undocumented aspects of their study.

The primary goal was to validate the reproducibility of Han et al.[9]'s results and assess the role of critical parameters, such as the α_3 stability constraint, in the performance and stability of the LAC algorithm across various simulated environments. By reimplementing and rigorously testing the algorithms, our study confirmed that the α_3 parameter significantly impacts the performance and stability of the LAC algorithm, particularly in complex environments where maintaining high stability is essential for successful operation. The results demonstrated that a more conservative α_3 value, which tightens the stability constraint, improves performance and stability but also increases the time needed for the agent to adapt and stabilize its policy, highlighting the need to balance stability and adaptability in scenarios demanding both high performance and strict stability. Despite some small differences, the study confirmed the reproducibility of Han et al.[9]'s findings, demonstrating the potential benefits of the LAC algorithm compared to SAC in terms of stability and robustness. Furthermore, the study showed that other hyperparameters, such as network architecture, training length, and horizon length, greatly influence the performance and stability of the LAC and SAC algorithms, underscoring the importance of precise hyperparameter tuning, transparent reporting, and consistent experimental configurations in reproducing and validating results in reinforcement learning research.

By validating the stability of the LAC algorithm across different simulated environments, and through the development of detailed guidelines for key hyperparameters and a robust codebase, this research provides a solid foundation for extending the algorithm's application to more complex tasks, such as effort-controlled environments. These contributions facilitate reproducibility and pave the way for future research into the LAC algorithm's ability to operate in dynamic and unpredictable environments, ultimately supporting the safe deployment of learning-based controllers in real-world systems where stability and reliability are paramount.

Keywords: Reinforcement Learning, Actor-Critic algorithm, Soft Actor-Critic, Lyapunov Actor-Critic, Robustness, Stability, Effort Control, Robotic Manipulators, Hyperparameter Tuning, Real-World Robotics, Panda Emika Franka.

Acknowledgements

This report signifies the completion of my Master's program in Biomechanical Design at TU Delft. Balancing this project with my other research and professional endeavours was challenging, but it has been a rewarding journey filled with invaluable lessons, both professionally and personally. Although the outcome is not exactly as I envisioned due to time constraints, I am proud of the result and even more so for submitting it as is, overcoming my perfectionism. Reflecting on this journey, I am proud of the progress I have made and the obstacles I have overcome.

First and foremost, I extend my deepest gratitude to my wife, family and friends for their unwavering support throughout this demanding journey. Their love, encouragement, and understanding have kept me going to the finish line. I also thank my initial supervisor, Wei Pan, for introducing me to this fascinating field and enhancing my research skills through various projects alongside my thesis. Lastly, I am profoundly grateful to my supervisor, Dr. Ir. M. (Martijn) Wisse, for his guidance, patience, and support. His motivation and encouragement during the most challenging times have been invaluable. I am thankful for the opportunity to have worked with and learned from him.

*Rick Staa
Haarlem, September 2024*

Contents

Abstract	i
Acknowledgements	ii
Nomenclature	x
1 Introduction	1
2 Preliminaries	3
2.1 Lyapunov Stability Theory	3
2.2 Reinforcement Learning	5
2.2.1 Introduction to Reinforcement Learning	5
2.2.2 Actor Critic Algorithm	8
2.2.3 Soft-Actor Critic Algorithm	9
2.2.4 Lyapunov Actor-Critic Algorithm	11
3 Method	13
3.1 Reproduction Study	13
3.1.1 Replication of Research Environments	13
3.1.2 Translation of the Algorithms	14
3.1.3 Alpha3 Hyperparameter Tuning	15
3.1.4 Convergence and Performance Evaluation	16
3.1.5 Additional LAC Critic Layer Experiment	16
4 Results	18
4.1 Reproduction Study	18
4.1.1 Alpha3 Hyperparameter Tuning	18
4.1.1.1 Performance and Convergence Analysis	18
4.1.1.2 Stability Assessment via Lambda Convergence	20
4.1.2 Comparison of LAC and SAC Performance and Convergence	22
4.1.3 Impact of Additional Critic Layer on LAC Performance	24
5 Discussion	25
6 Conclusion	28
References	29
A Additional Reproduction Study Remarks	31
A.1 Code Translation Validation Process	31
A.1.1 Environments Validation	31
A.1.1.1 Comparison with Foundational Environment Sources	32
A.1.1.2 Analysis of Han et al.'s Adaptations	32
A.1.2 Algorithms Validation	32
A.2 Code Inconsistencies and Discrepancies	33
A.2.1 Environment Inconsistencies and Discrepancies	33
A.2.1.1 CartPole Environment	33
A.2.1.2 GRN Environments	34
A.2.1.3 FetchReach Environment	35
A.2.2 Algorithm Implementation Discrepancies	35
A.2.2.1 Hyperparameter Values	35
A.2.2.2 Learning Rate Configurations	36
A.2.2.3 Network Architecture Configurations	36
A.3 Enhancements for Algorithmic Numerical Stability	36

B Panda Effort Control Experiment	38
B.1 Experimental Setup	38
B.1.1 PandaReach Task	39
B.1.1.1 PandaTrack Task	39
B.2 Hyperparameter Tuning	39
B.3 Algorithm Training	40
B.4 Convergence and Performance Evaluation	40
B.4.1 Performance Evaluation	41
B.4.2 Convergence Evaluation	41
B.5 Stability Analysis	41
B.5.1 Reach Task Stability	41
B.5.2 Tracking Task Stability	41
B.6 Robustness and Generalization Experiments	42
B.6.1 Robustness Evaluation	42
B.6.2 Generalization Evaluation	42
B.6.3 Conclusion on Robustness and Generalization	42
C Codebase Structure and Overview	44
C.1 Codebase Overview	44
C.1.1 Stable Learning Control	45
C.1.2 Stable Gym	45
C.1.3 ROS Gazebo Gym	46
C.1.4 ROS Gazebo Gym Examples	47
C.1.5 ROS Gazebo Gym Workspace	47
C.1.6 Panda Gazebo	47
D Additional Contributions and Byproducts of Research	49
D.1 Article: Deep reinforcement learning control approach to mitigating actuator attacks	49
D.2 Open Source Contributions	49
D.2.1 Franka Emika Panda Robot Gazebo Simulation	49
D.2.2 Panda MoveIt Config	50
D.2.3 Minor Contributions	50
E Supplementary Figures and Tables	51
E.1 Figures	51
E.1.1 Reproduction Study	51
E.1.1.1 Additional Alpha3 Tuning Experiments	51
E.1.1.1.1 Extended Seed Analysis for GRN and CompGRN Environments	51
E.1.1.1.2 Impact of Prolonged Training on GRN Environment	53
E.1.1.1.3 Impact of Lambda Learning Rate	54
E.1.1.1.4 Impact of Actor Network Structure	59
E.1.1.1.5 Impact of Shoter Finite Horizon Length	64
E.1.1.1.6 Effect of Critic Network Structure	68
E.1.1.1.7 Analysis of Alpha3 in Han et al.'s Original Codebase	69
E.1.1.2 Extended Alpha3 Tuning Test Performance and Convergence Results	71
E.1.1.3 Extended Alpha3 Tuning Lambda Convergence Results	72
E.2 Tables	73
E.2.1 Reproduction Study	73
E.2.1.1 Alpha3 Tuning Hyperparameters	73
E.2.1.2 Additional Alpha3 Tuning Experiments	73
E.2.1.2.1 Extended Seed Analysis for GRN and CompGRN Environments	73
E.2.1.3 Alpha3 Tuning Performance and Convergence Statistics	75
E.2.1.4 Alpha3 Tuning Lambda Convergence Statistics	77
E.2.1.5 LAC SAC Comparison Performance and Convergence Statistics	80
E.2.1.6 Extra SAC Critic Network Experiment	82

List of Figures

2.1	Several notions of stability.	3
2.2	Interaction between an agent and its environment in a Markov decision process.	5
2.3	The actor-critic architecture.	8
4.1	Average test performance and convergence of the LAC algorithm for select α_3 values in different environments.	19
4.2	Convergence of λ values during LAC algorithm training across various environments for select α_3 values.	21
4.3	Average test performance and convergence of the LAC and SAC algorithm for select α_3 values in different environments.	23
4.4	Average test performance and convergence of the SAC algorithm between <i>two different critic network architectures</i> in various benchmark environments.	24
C.1	A diagrammatic representation of the codebase.	44
C.2	The structure of ROS Gazebo Gym package.	46
C.3	A selection of simulated environments within the Panda Gazebo package.	47
E.1	Average test performance and convergence of the LAC algorithm for select α_3 values in the GRN environment between <i>five</i> and <i>ten</i> runs with different seeds.	51
E.2	Convergence of λ values during LAC algorithm training in the GRN environment between <i>five</i> and <i>ten</i> runs with different seeds for select α_3 values.	52
E.3	Average test performance and convergence of the LAC algorithm for select α_3 values in the CompGRN environment between <i>five</i> and <i>ten</i> runs with different seeds.	52
E.4	Convergence of λ values during LAC algorithm training in the CompGRN environment between <i>five</i> and <i>ten</i> runs with different seeds for select α_3 values.	53
E.5	Average Test Performance and Lambda convergence of the LAC algorithm for select α_3 values in the GRN environment after <i>prolonged training</i>	53
E.6	Average test performance and convergence of the LAC algorithm for select α_3 values between <i>two different λ learning rates</i> in the CartPole environment.	54
E.7	Convergence of λ values for the LAC algorithm for select α_3 values between between <i>two different λ learning rates</i> in the CartPole environment.	54
E.8	Average test performance and convergence of the LAC algorithm for select α_3 values between <i>two different λ learning rates</i> in the GRN environment.	55
E.9	Convergence of λ values for the LAC algorithm for select α_3 values between between <i>two different λ learning rates</i> in the GRN environment.	55
E.10	Average test performance and convergence of the LAC algorithm for select α_3 values between <i>two different λ learning rates</i> in the CompGRN environment.	56
E.11	Convergence of λ values for the LAC algorithm for select α_3 values between between <i>two different λ learning rates</i> in the CompGRN environment.	56
E.12	Average test performance and convergence of the LAC algorithm for select α_3 values between <i>two different λ learning rates</i> in the FetchReach environment.	57
E.13	Convergence of λ values for the LAC algorithm for select α_3 values between between <i>two different λ learning rates</i> in the FetchReach environment.	57
E.14	Average test performance and convergence of the LAC algorithm for select α_3 values between <i>two different λ learning rates</i> in the FetchReach (infinite horizon) environment.	58
E.15	Convergence of λ values for the LAC algorithm for select α_3 values between between <i>two different λ learning rates</i> in the FetchReach (infinite horizon) environment.	58
E.16	Average test performance and convergence of the LAC algorithm for select α_3 values between <i>two different actor network structures</i> in the CartPole environment.	59

E.17 Convergence of λ values for the LAC algorithm for select α_3 values between between <i>two different actor network structures</i> in the CartPole environment.	59
E.18 Average test performance and convergence of the LAC algorithm for select α_3 values between <i>two different actor network structures</i> in the GRN environment.	60
E.19 Convergence of λ values for the LAC algorithm for select α_3 values between between <i>two different actor network structures</i> in the GRN environment.	60
E.20 Average test performance and convergence of the LAC algorithm for select α_3 values between <i>two different actor network structures</i> in the CompGRN environment.	61
E.21 Convergence of λ values for the LAC algorithm for select α_3 values between between <i>two different actor network structures</i> in the CompGRN environment.	61
E.22 Average test performance and convergence of the LAC algorithm for select α_3 values between <i>two different actor network structures</i> in the FetchReach environment.	62
E.23 Convergence of λ values for the LAC algorithm for select α_3 values between between <i>two different actor network structures</i> in the FetchReach environment.	62
E.24 Average test performance and convergence of the LAC algorithm for select α_3 values between <i>two different actor network structures</i> in the FetchReach (infinite horizon) environment.	63
E.25 Convergence of λ values for the LAC algorithm for select α_3 values between between <i>two different actor network structures</i> in the FetchReach (infinite horizon) environment.	63
E.26 Average test performance and convergence of the LAC algorithm for select α_3 values between two different finite horizon lengths in the CartPole environment.	64
E.27 Convergence of λ values for the LAC algorithm for select α_3 values between between two different finite horizon lengths in the CartPole environment.	64
E.28 Average test performance and convergence of the LAC algorithm for select α_3 values between two different finite horizon lengths in the GRN environment.	65
E.29 Convergence of λ values for the LAC algorithm for select α_3 values between between two different finite horizon lengths in the GRN environment.	65
E.30 Average test performance and convergence of the LAC algorithm for select α_3 values between two different finite horizon lengths in the CompGRN environment.	66
E.31 Convergence of λ values for the LAC algorithm for select α_3 values between between two different finite horizon lengths in the CompGRN environment.	66
E.32 Average test performance and convergence of the LAC algorithm for select α_3 values between two different finite horizon lengths in the FetchReach environment.	67
E.33 Convergence of λ values for the LAC algorithm for select α_3 values between between two different finite horizon lengths in the FetchReach environment.	67
E.34 Average test performance and convergence of the LAC algorithm for select α_3 values between <i>two different critic network architectures</i> in the CompGRN environment.	68
E.35 Convergence of λ values for the LAC algorithm for select α_3 values between <i>two different critic network architectures</i> in the CompGRN environment.	68
E.36 Average test performance and convergence of the LAC algorithm for select α_3 values in different environments in the original codebase of Han et al.[9].	69
E.37 Convergence of λ values during LAC algorithm training across various environments for select α_3 values in different environments in the original codebase of Han et al.[9].	70
E.38 Average test performance and convergence of the LAC algorithm for select α_3 values in different environments.	71
E.39 Convergence of λ values during LAC algorithm training across various environments for select α_3 values.	72

List of Tables

3.1	Summary of key environments selected for our <i>Reproduction Study</i>	14
B.1	The hyperparameters tuned in the <i>Panda Effort Control Experiment</i>	40
E.1	Hyperparameters employed in the α_3 tuning process for our <i>Reproduction Study</i>	73
E.2	Test performance and convergence statistics for the GRN environment with varying α_3 values between five and ten runs with different seeds.	73
E.3	Lambda convergence statistics for the GRN environment with varying α_3 values between five and ten runs with different seeds.	74
E.4	Test performance and convergence statistics for the CompGRN environment with varying α_3 values between five and ten runs with different seeds.	74
E.5	Lambda convergence statistics for the CompGRN environment with varying α_3 values between five and ten runs with different seeds.	74
E.6	Test performance comparison for each seed in the GRN and CompGRN environments with $\alpha_3 = 0.1$	74
E.7	Test performance and convergence metrics for the CartPole environment with varying α_3 values, averaged over 5 randomly seeded runs.	75
E.8	Test performance and convergence metrics for the GRN environment with varying α_3 values, averaged over 5 randomly seeded runs.	75
E.9	Test performance and convergence metrics for the CompGRN environment with varying α_3 values, averaged over 5 randomly seeded runs.	76
E.10	Test performance and convergence metrics for the FetchReach environment with varying α_3 values, averaged over 5 randomly seeded runs.	76
E.11	Test performance and convergence metrics for the FetchReach (infinite horizon) environment with varying α_3 values, averaged over 5 randomly seeded runs.	77
E.12	Lambda convergence statistics for the CartPole environment with varying α_3 values, averaged over 5 randomly seeded runs.	77
E.13	Lambda convergence statistics for the GRN environment with varying α_3 values, averaged over 5 randomly seeded runs.	78
E.14	Lambda convergence statistics for the CompGRN environment with varying α_3 values, averaged over 5 randomly seeded runs.	78
E.15	Lambda convergence statistics for the FetchReach environment with varying α_3 values, averaged over 5 randomly seeded runs.	79
E.16	Lambda convergence statistics for the FetchReach (infinite horizon) environment with varying Alpha3 values, averaged over 5 randomly seeded runs.	79
E.17	Test performance and convergence statistics for the CartPole environment with different conditions.	80
E.18	Test performance and convergence statistics for the GRN environment with different conditions.	80
E.19	Test performance and convergence statistics for the CompGRN environment with different conditions.	81
E.20	Test performance and convergence statistics for the FetchReach environment with different conditions.	81
E.21	Test performance and convergence statistics for the FetchReach (infinite horizon) environment with different conditions.	82
E.22	Test performance and convergence statistics for the CartPole environment with different critic network architectures.	82
E.23	Test performance and convergence statistics for the GRN environment with different critic network architectures.	82

E.24 Test performance and convergence statistics for the CompGRN environment with different critic network architectures.	82
E.25 Test performance and convergence statistics for the FetchReach environment with different critic network architectures.	83

List of Theorems

2.1 Sufficient conditions for stability of autonomous systems [22]	4
--	---

Nomenclature

This chapter contains abbreviations and symbols used throughout this report. It only includes abbreviations and symbols specific to this report or ones with ambiguous definitions. Please check out [18] for a list of standardized math symbols.

Abbreviations

Abbreviation	Definition
A3C	Asynchronous Advantage Actor-Critic
AS	Asymptotically stable
CompGRN	Complex gene regulatory networks
DDPG	Deep Deterministic Policy Gradient
DP	Dynamic programming
ES	Exponential Stability
GPU	Graphics processing unit
GRN	Gene regulatory networks
LAC	Lyapunov Actor-Critic
LATC	Lyapunov Actor Twin-Critic
MC	Monte Carlo
MDP	Markov Decision Process
PPO	Proximal Policy Optimization
RL	Reinforcement learning
SAC	Soft actor-critic
SARSA	State-Action-Reward-State-Action
SGA	Stochastic Gradient Ascent
SGD	Stochastic Gradient Descent
SISL	Stability in the Sense of Lyapunov
TD	Temporal Difference
TD3	Twin Delayed Deep Deterministic Policy Gradient

Symbols

Symbol	Definition	Unit
L	Lyapunov function	
$J(\cdot)$	Objective function, usually maximized	
$\mathcal{L}(\cdot)$	Loss function, usually minimized	
$V(s)$	Value function at state s	
$Q(s, a)$	Action-value function for state-action pair (s, a)	
c	Cost function	
α	Learning rate	
β	Entropy regularization coefficient	
σ	Critic learning rate	
λ	Stability coefficient	
γ	Discount factor for future rewards	
τ	Soft update coefficient for target networks	
ζ	Trajectory	
π	Policy function (mapping from state to action)	

Symbol	Definition	Unit
T	Total number of timesteps	
t	Time step	
δ	Error threshold for termination	
θ	Policy parameters	
ψ	Q-Critic parameters	
ϕ	Value function parameters	
R	Reward	
G	Return	
S	State	
A	Action	
\mathcal{P}	Transition probabilities	

1

Introduction

As robots increasingly transition from controlled factory settings to complex real-world environments, there is a growing demand for robots capable of working safely alongside humans and other robots [25, 24, 29]. An essential prerequisite for achieving this lies in ensuring the stability of the robot's controller. Ensuring stability is paramount for the secure operation of robotic systems in real-world scenarios, as instability may result in unpredictable and hazardous behaviours. While the stability of robotic systems is well understood for classical control methods, which typically provide rigorous stability guarantees, these methods usually require a comprehensive system model [4, 12, 1]. This reliance on a detailed model makes them less suitable for complex tasks where the system model is incomplete or poorly understood.

To overcome the limitations of classical control methods, researchers have increasingly turned to learning-based controllers, which can acquire complex behaviours without requiring a full system model or explicit programming. This makes them more suitable for dynamic and unpredictable environments. Two prominent learning-based control methods gaining traction are imitation learning and reinforcement learning [27, 21, 10]. Imitation learning, which acquires a policy by mimicking an expert's behaviour, is often preferred for tasks that are easily demonstrable, given its sample efficiency and faster learning [11]. However, it may introduce human biases, resulting in suboptimal policies. In contrast, reinforcement learning excels in scenarios where the desired behaviour is difficult to demonstrate or unknown. Although reinforcement learning can yield superior policies compared to imitation learning, it often suffers from reduced sample efficiency and longer training times [23, 21]. Despite the potential of these methods to learn complex behaviours [23], their inherent instability has limited their safe deployment in real-world applications [28].

Recent research has increasingly leveraged Lyapunov stability theory to provide stability guarantees for learning-based controllers, particularly in imitation learning applications [22]. However, providing stability guarantees for model-free reinforcement learning methods remains a significant challenge due to the inherent trial-and-error nature of reinforcement learning [9]. Early work in this area, such as the application of Lyapunov functions in constrained Markov Decision Processes, primarily addressed safety but did not fully address system stability. Other approaches have conducted stability analysis, but they often rely on assumptions about system dynamics that are difficult to meet in practice or introduce constraints that are impractical for high-dimensional systems [9]. To address these shortcomings, Han et al.[9] introduced the Lyapunov Actor-Critic (LAC) algorithm, an extension of the widely used Soft Actor-Critic (SAC) algorithm. LAC integrates stability constraints directly into the policy optimization process, ensuring that learned policies remain stable even in the presence of disturbances and uncertainties. LAC has demonstrated comparable or superior performance to state-of-the-art algorithms, with particular advantages in stability and robustness across several simulated environments. However, in the Fetch manipulator environment, which is position-controlled, no additional stability benefits were observed compared to the SAC algorithm. This is likely because the underlying position PID controller already effectively manages disturbances and instabilities, rendering LAC's stability guarantees unnecessary in such a context.

The original goal of this research was to extend Han et al.[9]'s work by investigating how the LAC algorithm performs compared to the widely-used SAC algorithm in an effort-controlled robotic manip-

ulation task. Specifically, the research was aimed at evaluating the stability and robustness benefits of the LAC algorithm in an effort-controlled environment, such as the Panda manipulator, where disturbances are not automatically managed by a position controller. This environment would have allowed for a more comprehensive evaluation of LAC's potential to improve the robustness and generalizability of learning-based controllers in real-world robotics tasks. However, due to time and computational constraints, the *Panda Effort Control Experiment* could not be carried out within the scope of this thesis. As a result, this research shifted focus towards ensuring the reproducibility of the results reported by Han et al.[9] through a comprehensive *reproduction study*. This study serves as a critical foundational step, verifying the validity of the LAC algorithm under different simulated environments and preparing the groundwork for future experiments.

The primary focus of this thesis is the *reproduction study*, which aims to validate the results of Han et al.[9]. This study is crucial for ensuring the reliability of the LAC algorithm across different simulated environments, particularly in validating the effect of the α_3 parameter, which specifies the strictness of the stability constraint. By reimplementing the LAC algorithm, this study not only verifies Han et al.[9]'s results but also provides a strong foundation for future research, including the *Panda Effort Control Experiment*. The main research question guiding this *Reproduction Study* is:

How reproducible are the results of Han et al.[9] in new simulated environments, and what role do key parameters (such as the α_3 parameter) play in the performance and stability of the LAC algorithm compared to SAC?

To address this question, the study is divided into the following subquestions:

- *What key discrepancies in environment configurations, hyperparameters, and network architectures were identified in Han et al.[9]'s research, and how did they affect the reproducibility of the LAC algorithm?*
- *How sensitive is the performance and stability of the LAC algorithm to changes in key hyperparameters such as network architecture, horizon length, and the λ learning rate during the reproduction study?*
- *How does the α_3 parameter influence the performance and stability of the LAC algorithm compared to the SAC algorithm across various environments?*
- *How do different simulated environments affect the convergence and stability of the LAC algorithm, and what role do environment-specific factors play in reproducibility?*

Although the original goal of conducting the *Panda Effort Control Experiment* could not be realized within the timeframe of this research, this thesis lays a crucial foundation by ensuring that the LAC algorithm is validated through the *Reproduction Study*. By successfully reproducing Han et al.[9]'s results, this study ensures that future experiments can proceed with a solid methodological basis and a validated algorithm.

This report is structured as follows: the next chapter provides the theoretical background necessary to understand the LAC algorithm and its relevance to learning-based controllers. Following this, the methodology for the *Reproduction Study* is presented, detailing the experimental setup, environments, and algorithms used. The results of the *Reproduction Study* are then discussed, including the impact of the α_3 parameter on stability and performance. The report concludes with a summary of findings and recommendations for future research, particularly regarding the *Panda Effort Control Experiment*.

2

Preliminaries

This chapter discusses the key theoretical concepts foundational to the research presented in this report. We start with a brief review of the *Lyapunov stability theory*, which provides the theorems and principles used in *Lyapunov stability analysis*, instrumental to our methodological framework. Subsequently, our focus shifts to *model-free reinforcement learning*, with an emphasis on the *soft actor-critic* (SAC) algorithm as proposed by Haarnoja et al.[7]. Building on this, we delve into the *Lyapunov Actor-Critic* (LAC) algorithm introduced by Han et al.[9], emphasizing its distinctive features that guarantee stability. Our goal is to provide a concise, targeted overview aligned with the thematic scope of our research rather than an exhaustive analysis. For more detailed discussions, readers are encouraged to consult the cited references.

2.1. Lyapunov Stability Theory

As highlighted in the introduction, stability is a crucial aspect of any control system, ensuring that the system's behaviour remains predictable and safe. Both the concept of stability and the theorems provided by Lyapunov stability theory, fundamental tools in (nonlinear) stability analysis, have been discussed in detail in a previous literature study by the author [22]. This section provides a brief overview of the key concepts and definitions to aid understanding of our research. Select portions of the original text, figures, and definitions are directly quoted for clarity. For a deeper exploration, including mathematical definitions and detailed examples, readers should refer to the previously mentioned literature.

In control theory, stability means the control system is guaranteed to always converge to a given setpoint while remaining bounded in state space. Unlike linear systems, where stability can be determined using general closed-loop solutions and eigenvalue analysis, determining stability in nonlinear dynamical systems is nontrivial since a general closed-loop solution is often unavailable. Lyapunov's

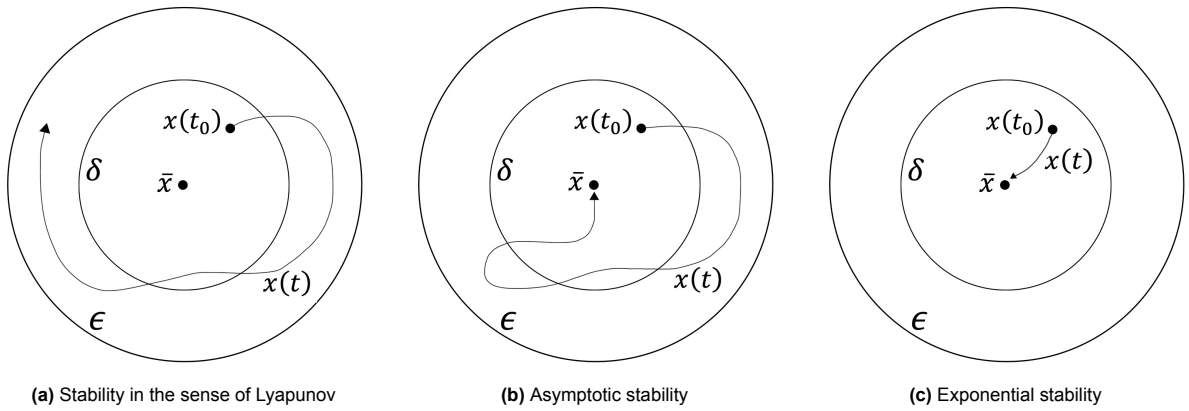


Figure 2.1: Several notions of stability. In these figures, δ , ϵ represent the start and end stability bounds, \bar{x} the equilibrium point, and $x(t_0)$ and $x(t)$ the initial point and trajectory, respectively [22].

stability theory, first introduced in 1892, provides a framework for reasoning about the stability of equilibrium points in these nonlinear systems without solving the complete system behaviour. It encompasses several notions of stability, fundamental theorems, and methods for investigating the system's stability. The definitions of stability in Lyapunov's stability theory focus on the behaviour of trajectories rather than just the stability of the equilibrium point. This trajectory-focused approach is beneficial because noise or disturbances can always perturb a physical system from its equilibrium. The most relevant notions of stability within Lyapunov's stability theory for our research are:

1. **Stability in the Sense of Lyapunov (SISL):** This form of stability means that the system's response remains within a bounded range around the equilibrium for any small perturbation. It is a local notion, indicating that if a system starts close enough to an equilibrium, it will stay close.
2. **Asymptotic Stability (AS):** A stronger form of stability where, in addition to being SISL, the system eventually converges to the equilibrium point over time.
3. **Exponential Stability (ES):** An even stronger form, where the system not only converges to the equilibrium point but does so at an exponential rate.

Figure 2.1 shows a visual representation of these stability notions. Lyapunov stability theory provides two main methods for assessing different types of stability in nonlinear systems: the *indirect stability method* and the *direct stability method*. The indirect stability method uses a first-order Taylor series to linearise the system around the equilibrium point. This method, however, only proves local stability and cannot provide insights into the global stability of the system or the region of attraction. In contrast, the direct stability method investigates the stability of a system using an energy-based approach. According to this method, a general (non)linear system is stable if a **positive definite** (energy) function of the state variables $L(x)$ ¹, called a *Lyapunov function*, exists that decreases over time, meaning $\dot{L}(x)$ is **negative (semi-)definite**. When found, this Lyapunov function can demonstrate several notions of stability when certain Lyapunov conditions are met. The Lyapunov conditions for the earlier defined stability notions (i.e., SISL, AS, ES) are shown without proof in Theorem 2.1. This theorem takes the origin (i.e., $x = 0$) as the equilibrium point. However, since any point can be shifted to the origin using a change of variables, this does not lead to a loss of generality.

The stability theorem presented in Theorem 2.1 applies to a nonlinear time-invariant autonomous system of the form $\dot{x} = f(x)$. However, Lyapunov's stability theory also includes similar theorems for non-autonomous and time-varying systems [22]. It is important to note that the conditions of Theorem 2.1 used in Lyapunov's direct method are **sufficient conditions** for stability. These conditions only prove stability if a Lyapunov function L is found. If these conditions are violated for a given Lyapunov candidate, it does not prove that the equilibrium is unstable but rather that it is not a proper Lyapunov function. From this, we can directly see the main weakness of this method, namely that finding these functions is not trivial. Since no general method exists for finding Lyapunov functions in nonlinear systems, they must be found by trial and error. However, in practice, the situation is not as challenging as it seems due to the extensive research on this topic. As a result, when trying to find a Lyapunov function for a new system, Lyapunov functions from prior research on similar systems can be used as candidates [8]. Additionally, in recent years, several data-driven approaches, such as optimization and learning-based methods, have been designed for specific types of systems that can iteratively find Lyapunov functions [13, 6, 20, 9]. These methods are instrumental in the context of reinforcement learning, where the system's dynamics are unknown and must be learned from data.

Theorem 2.1: Sufficient conditions for stability of autonomous systems [22]

Let $x = 0$ be an equilibrium point of $\dot{x} = f(x)$ and $D \subset \mathbb{R}^n$ be a domain containing $x = 0$. Let $L : D \rightarrow \mathbb{R}$ be a continuously differentiable function such that

$$L(0) = 0 \quad \text{and} \quad L(x) > 0 \quad \text{in} \quad D - \{0\} \quad (2.1)$$

$$\dot{L}(x) = \frac{\partial L}{\partial x} f(x) \leq 0 \quad \text{in} \quad D \quad (2.2)$$

¹In the literature, the symbol V is commonly used to denote Lyapunov functions. However, to avoid confusion with the value functions discussed in this chapter, we have replaced V with L .

then, $x = 0$ is (locally) **SISL**. Moreover, if

$$\dot{L}(x) = \frac{\partial L}{\partial x} f(x) < 0 \quad \text{in } D - \{0\} \quad (2.3)$$

then $x = 0$ is (locally) **AS**. Furthermore, if we have

$$\dot{L}(x) = \frac{\partial L}{\partial x} f(x) \leq -\alpha L(x) \quad \text{in } D - \{0\} \quad (2.4)$$

then $x = 0$ is (locally) **ES**. Finally, if $D = \mathbb{R}^n$, and (2.1) holds for all $x \neq 0$, and x is

$$\|x\| \rightarrow \infty \Rightarrow L(x) \rightarrow \infty \quad (2.5)$$

then x is said to be **radially unbounded**, meaning that trajectories cannot diverge to infinity even as L decreases. Consequently, $x = 0$ is **GAS** if (2.3) holds and **GES** if (2.4) holds.

2.2. Reinforcement Learning

2.2.1. Introduction to Reinforcement Learning

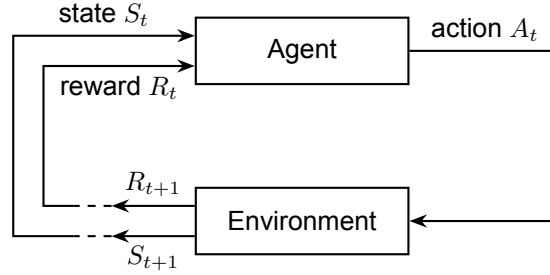


Figure 2.2: Interaction between an agent and its environment in a Markov decision process.

Reinforcement learning is a machine learning method where an agent learns to make decisions through direct interaction with an environment, receiving feedback in the form of rewards or penalties. This approach is similar to how humans and animals learn from their interactions with the world through trial and error. Unlike supervised learning, which involves learning from labelled data, and unsupervised learning, which involves finding patterns in unlabelled data, reinforcement learning emphasizes learning from interaction to achieve a specific goal. This interactive decision-making process, illustrated in Figure 2.2, is often modelled using a (finite) *Markov Decision Process* (MDP). An MDP provides a mathematical framework for modelling decision-making problems where both the environment's dynamics, which may include random factors, and the actions of the decision-maker influence outcomes. In an MDP, the interaction between the agent and the environment occurs at discrete time steps². At each time step $t \in \mathbb{N}_0$, the agent observes the current state $S_t \in \mathcal{S}$ of the environment and chooses an action $A_t \in \mathcal{A}(S_t)$ based on that state. Following this action, the environment provides a numerical reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and transitions to a new state $S_{t+1} \in \mathcal{S}$. This continuous process gives rise to a sequence of states, actions, and rewards $(\mathcal{S}, \mathcal{A} \text{ and } \mathcal{R})$ over time, called a trajectory. One important property of MDPs is the Markov property, which asserts that the future state S_{t+1} and reward R_{t+1} depend only on the current state S_t and action A_t , and not on the sequence of events that preceded it. This property simplifies the modelling of the environment's dynamics as the transition probabilities $\mathcal{P}(S_{t+1}, R_{t+1} | S_t, A_t)$, which defines the likelihood of transitioning to state S_{t+1} and receiving reward R_{t+1} given the current state S_t and action A_t , fully characterize the dynamics of the environment.

The goal of reinforcement learning is to find a policy π , a mapping from states to probabilities of selecting possible actions $\pi(a|s)$, that maximizes the expected cumulative reward, or return, over time.

²For simplicity, we focus on discrete-time dynamics. For continuous-time dynamics, see Sutton and Barto [27].

The return G_t at time step t is defined as the total accumulated reward, which can be expressed as

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

where $\gamma \in [0, 1)$ is the discount factor that determines the weight of future rewards relative to immediate rewards. For problems with a finite horizon or episodic tasks, γ is typically set to 1, and the sum may terminate after a finite number of steps when the episode ends. In contrast, for infinite-horizon tasks, the sum continues indefinitely, with the discount factor γ ensuring that distant future rewards are weighted less heavily. To assess the quality of a given policy π in achieving these returns, many reinforcement learning algorithms involve estimating *value functions* and *action-value functions*. The value function of a state s under policy π , denoted as $V_\pi(s)$, represents the expected return starting from that state and following policy π . Defined as

$$V_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right], \text{ for all } s \in \mathcal{S}.$$

Similarly, the action-value function $Q_\pi(s, a)$ extends this concept by considering both the state s and action a . It represents the expected return when starting in state s , taking action a , and then following policy π . Defined as

$$Q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right], \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}.$$

These two functions are central to reinforcement learning as they can be estimated from experience to evaluate and compare different states and actions in terms of their expected cumulative rewards. One fundamental property of these functions is that they satisfy a recursive relationship known as the *Bellman equation*. The Bellman equation for the value function $V_\pi(s)$ expresses a relationship between the value of a state and the values of its successor states. Formally, it is given by

$$\begin{aligned} V_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s], \quad \text{for all } s \in \mathcal{S}, \end{aligned}$$

where the value of the current state must equal the immediate reward plus the discounted value of the expected next state. This same recursive relationship holds for the action-value function Q_π , where the Bellman equation is

$$\begin{aligned} Q_\pi(s, a) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \\ &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma Q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a], \quad \text{for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}. \end{aligned}$$

These equations provide a recursive decomposition of the value and action-value functions, which is crucial for determining the *optimal policy* π_* that maximizes the expected return across all states. Solving the Bellman optimality equation yields the *optimal value function* $V_*(s)$ and the *optimal action-value function* Q_* , which represent the maximum expected return over all policies. The optimal value function $V_*(s)$ is defined as

$$V_*(s) \doteq \max_{\pi} V_\pi(s), \quad \text{for all } s \in \mathcal{S},$$

while the optimal action-value function Q_* is given by

$$Q_*(s, a) \doteq \max_{\pi} Q_\pi(s, a), \quad \text{for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}.$$

Although the Bellman optimality equation has a unique solution for finite MDPs and theoretically provides a solid foundation for identifying the optimal value and action-value functions—and thus the optimal policy—explicitly solving these equations is often infeasible. This infeasibility arises because the following key assumptions underlying the MDP framework are frequently difficult to meet: (1) the dynamics of the environment may not be accurately known, (2) ensuring that the states strictly adhere to

the Markov property can be challenging, and (3) the computational resources are often insufficient to complete the necessary calculations.

Dynamic programming (DP) offers a more computationally efficient way to compute the optimal value functions iteratively rather than attempting to solve the Bellman equation directly. DP methods, such as *value iteration* and *policy iteration*, iteratively update the value functions based on the Bellman equation until they converge to the optimal values. Value iteration involves repeatedly applying the Bellman optimality equation to update the value of each state until the values stabilize at their optimal levels, $V_*(s)$. Once $V_*(s)$ is determined, the optimal policy π_* can be derived by choosing the action that maximizes the expected return at each state. Policy iteration alternates between policy evaluation and policy improvement: policy evaluation computes the value function $V_\pi(s)$ for the current policy π , and policy improvement updates the policy to be greedy with respect to the current value function. This iterative process continues until the policy converges to the optimal policy π_* .

DP methods, such as value iteration and policy iteration, are powerful and exact tools for solving Markov Decision Processes (MDPs) when the environment model is fully known and the state and action spaces are finite and manageable. By iteratively applying the Bellman equation, these methods guarantee convergence to the optimal value function and policy. However, DP methods are often impractical in real-world scenarios due to two primary challenges: the requirement for a precise model of the environment and the curse of dimensionality. When the dynamics of the environment are unknown or too complex to model explicitly, DP methods cannot be applied directly. Additionally, even if the model is available, the sheer size of the state and action spaces in many practical applications makes it computationally infeasible to store and update value functions for all states.

To address the challenge of operating without a complete environment model, *model-free* sample-based methods like *Monte Carlo* (MC) and *Temporal-Difference* (TD) learning are utilized. These methods learn directly from experience, offering greater flexibility in unknown or complex environments. MC methods estimate value functions by averaging returns from sampled episodes. The value function is updated after each episode using the formula

$$V(s_t) \leftarrow V(s_t) + \alpha [G_t - V(s_t)],$$

where G_t is the return (total accumulated reward) following state S_t , and α is the learning rate. While MC methods provide unbiased estimates, they are typically limited to episodic tasks, cannot update value estimates incrementally during an episode, and often suffer from high variance, which can lead to slow convergence and, thus, inefficient learning. TD methods, such as State-Action-Reward-State-Action (SARSA) and Q-learning, address some of the limitations of MC methods by updating value estimates incrementally at each time step using the update rule

$$V(s_t) \leftarrow V(s_t) + \alpha [R_{t+1} + \gamma V(s_{t+1}) - V(s_t)],$$

where R_{t+1} is the immediate reward, γ is the discount factor, and $V(s_{t+1})$ is the estimated value of the next state. This incremental approach makes TD methods suitable for both episodic and continuous tasks. TD methods generally learn more efficiently due to their ability to perform step-by-step updates, but they can introduce bias into the learning process, particularly when estimates are based on limited or inaccurate initial values. Although this bias can sometimes lead to instability, such as oscillations during training or divergence from the true value function, techniques like experience replay have been developed to mitigate these issues, making TD methods more practical in complex environments.

Despite their advantages, both Monte Carlo and TD methods in their traditional tabular forms struggle with large state and action spaces, where maintaining a table of values for every possible state or state-action pair becomes infeasible. To address this, parametric function approximators like neural networks, decision trees, and linear models are employed to generalize from limited data to broader state-action spaces. These approximators compress the representation of value functions, allowing the agent to learn effectively in high-dimensional environments. While non-parametric methods also exist, parametric methods—especially neural networks—are more widely used in modern reinforcement learning due to their ability to approximate complex, non-linear functions in large-scale problems. Typically, these parametric approximators are trained using optimization techniques such as stochastic gradient descent (SGD), which iteratively adjusts the model parameters to minimize the error in value function estimation. This error, or loss, is calculated based on the difference between the predicted and actual values, guiding the learning process and optimizing policy performance.

One of the primary disadvantages of traditional *value-based* methods, even when combined with function approximators, is their difficulty in handling environments with high-dimensional or continuous action spaces. Additionally, these methods often struggle with convergence issues in complex environments where the dynamics are highly non-linear or involve many interacting variables. To address these challenges, reinforcement learning also includes *policy-based* methods, which optimize the policy directly rather than relying solely on value function estimation. Policy-based methods are particularly effective in environments with continuous action spaces or where the policy needs to be stochastic to explore effectively. These methods adjust the policy parameters θ to maximize a scalar performance measure $J(\theta)$, defined as the expected return

$$J(\theta) = \mathbb{E}_{\zeta \sim \pi_\theta} [R(\zeta)],$$

where $R(\zeta)$ is a general reward function representing the return for a trajectory ζ , which varies depending on the reinforcement learning algorithm used—for example, G_t in Monte Carlo methods, Q_t in Q-learning, and so on. The policy parameters are updated to maximize this objective using the stochastic gradient ascent (SGA) rule

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_{\theta} J(\pi_{\theta}),$$

where $\nabla_{\theta} J(\pi_{\theta})$ represents the gradient of the expected return with respect to the policy parameters, and α is the learning rate. The gradient in the Monte Carlo case can be expressed as

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\pi} \left[\sum_{t=0}^{\infty} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) G_t \right],$$

which highlights how the policy parameters are adjusted by following the gradient of the expected return, weighted by the returns observed in the environment. The exact computation of this gradient depends on the specific parameterization of the policy π , which is influenced by the chosen function approximation technique. While policy-based methods offer greater flexibility and are particularly well-suited for complex, high-dimensional environments, they can suffer from high variance in gradient estimates and may converge more slowly than value-based methods.

To leverage the strengths of both approaches, *actor-critic* methods combine elements of both value-based and policy-based methods, offering a balanced solution that mitigates the individual weaknesses of each. These methods will be discussed in detail in the next section. For a deeper dive into reinforcement learning, including its theoretical foundations, practical applications, and a comprehensive exploration of both value-based and policy-based methods, readers are encouraged to consult Sutton and Barto's seminal work on the subject [27].

2.2.2. Actor Critic Algorithm

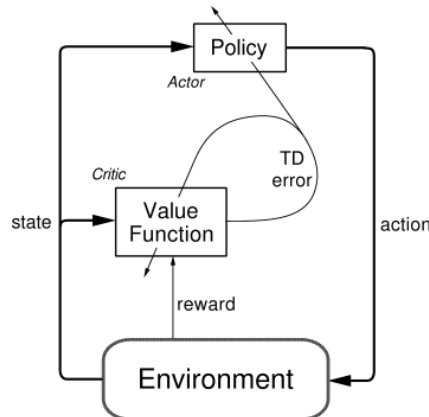


Figure 2.3: The actor-critic architecture [26].

Actor-Critic methods [14] play a pivotal role in model-free reinforcement learning, addressing the limitations of pure value-based methods, which often struggle with high-dimensional action spaces, and

policy-based methods, which can suffer from high variance and slower convergence. As illustrated in Figure 2.3, this framework comprises two primary components: the *actor* and the *critic*. The actor, or policy network, selects actions based on the current policy $\pi(a|s)$, while the critic, or value network, evaluates these actions by estimating a value function. In Actor-Critic methods, the actor's objective is to maximize the expected return, which aligns with the objective in policy-based methods, expressed as $J(\pi_\theta) = \mathbb{E}_\pi[G_t]$. However, rather than directly using the return G_t or the action-value Q_t to update the policy, Actor-Critic methods rely on the Temporal Difference (TD) error as feedback, providing a more stable and effective learning signal. At each time step t , the actor selects an action A_t based on the current state S_t and the policy $\pi(a|s)$. The critic then assesses the value of the resulting state using the TD error, defined as

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t),$$

where $V(S_t)$ is the critic's estimate of the value function at state S_t . The TD error quantifies the difference between the expected reward (as predicted by the critic's current value function) and the actual reward received. This error informs the critic whether the action taken by the actor resulted in a better or worse outcome than anticipated. The actor's policy is then updated based on this TD error, where actions that lead to higher-than-expected rewards are reinforced, and actions that result in lower-than-expected rewards are discouraged. This policy adjustment is formalized as

$$\theta_{t+1} \doteq \theta_t + \alpha \delta_t \nabla_\theta \log \pi_\theta(a_t|s_t),$$

where θ_t represents the parameters of the actor's policy at time step t , α is the learning rate, and $\nabla_\theta \log \pi_\theta(a_t|s_t)$ is the gradient of the log-probability of the action with respect to the policy parameters. Simultaneously, the critic's value function is updated to better estimate the value of states by minimizing the mean squared TD error. The critic's objective is to reduce the difference between the predicted value and the actual observed reward plus the estimated value of the next state. This objective can be formalized as the loss function

$$\mathcal{L}(\phi) = \frac{1}{2} \mathbb{E}_{s_t \sim \pi} [(\delta_t)^2] = \frac{1}{2} \mathbb{E}_{s_t \sim \pi} [(R_{t+1} + \gamma V_\phi(S_{t+1}) - V_\phi(S_t))^2],$$

where ϕ represents the parameters of the critic's value function. The gradient of this loss function with respect to ϕ is then used to update the critic's parameters, typically following a gradient descent rule

$$\phi_{t+1} \leftarrow \phi_t - \sigma \nabla_\phi \mathcal{L}(\phi),$$

where σ is the learning rate for the critic. Minimizing this loss function with respect to ϕ enables the critic to provide more accurate evaluations of the state values, which in turn helps the actor make better decisions. This dual approach allows the actor to leverage the critic's feedback, enabling more efficient and stable learning in complex environments, thereby addressing the shortcomings of using purely value-based or policy-based methods alone.

2.2.3. Soft-Actor Critic Algorithm

The standard Actor-Critic methods discussed above are effective across various problems [23], but they face significant challenges, particularly in complex, high-dimensional environments. On-policy algorithms like Asynchronous Advantage Actor-Critic (A3C) [19], which employ stochastic policies, suffer from high variance in policy gradients. This is largely due to their reliance on data collected from the current policy, which limits their ability to reuse past experiences, causing poor sample efficiency, inefficient exploration, and a tendency to get stuck in local optima. Entropy regularization, though employed to maintain exploration, often falls short of effectively balancing exploration and exploitation in complex environments. The fundamental challenge lies in enabling sufficient exploration while allowing the policy to become more deterministic as it learns—a balance that traditional methods like A3C struggle to achieve. Off-policy methods, on the other hand, such as Deep Deterministic Policy Gradient (DDPG) [17] and Twin Delayed Deep Deterministic Policy Gradient (TD3) [5], aim to improve sample efficiency by learning from a replay buffer of past experiences. However, their deterministic policies, where actions are chosen directly based on the policy without inherent randomness, can lead to instability and increased sensitivity to environmental changes because they limit exploration. While noise is introduced to promote exploration, this method requires careful hyperparameter tuning and does not always achieve stable learning, especially in high-dimensional spaces.

To address these issues, the SAC algorithm introduces several key innovations [7]. While methods like A3C employ stochastic policies and use entropy as a secondary regularizer to encourage exploration, SAC takes a more direct approach by maximizing policy entropy as a core optimization goal. By incorporating entropy directly into the objective function, SAC optimizes not only for the expected return but also for the expected entropy of the policy, thereby enhancing both stability and learning efficiency. The objective function $J(\pi)$ that SAC maximizes is given by³

$$J(\pi) = \sum_{t=0}^T \mathbb{E}_{(s_t, a_t) \sim \rho_\pi} [r(s_t, a_t) + \beta \mathcal{H}(\pi(\cdot | s_t))],$$

where β^4 is a temperature parameter controlling the trade-off between exploration (through entropy \mathcal{H}) and exploitation (through rewards $r(s_t, a_t)$). This formulation encourages the policy to remain exploratory, preventing premature convergence to suboptimal solutions. The temperature parameter β itself is dynamically adjusted to maintain the desired level of entropy. The objective function for the temperature parameter β is

$$J(\beta) = \mathbb{E}_{a_t \sim \pi_t} [-\beta \log \pi_t(a_t | s_t) - \beta \bar{\mathcal{H}}],$$

where $\bar{\mathcal{H}}$ is the target entropy. Dynamically optimizing this parameter allows the policy to adapt its exploration-exploitation trade-off depending on the state, encouraging exploration in uncertain regions and favouring more deterministic actions in well-understood states. This automatic adjustment ensures that the policy maintains an appropriate average level of entropy, contributing to a more stable learning process and reducing the need for meticulous hyperparameter tuning.

The critic in SAC is updated by minimizing the soft Bellman residual, which incorporates the entropy term. The loss function for the Q-function, $J_Q(\psi)$, is defined as

$$J_Q(\psi) = \mathbb{E}_{(s_t, a_t) \sim \mathcal{D}} \left[\frac{1}{2} (Q_\psi(s_t, a_t) - (r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim p} [V_{\bar{\psi}}(s_{t+1})]))^2 \right],$$

where the soft value function $V_{\bar{\psi}}(s_t)$ is computed as

$$V_{\bar{\psi}}(s_t) = \mathbb{E}_{a_t \sim \pi} [Q_\psi(s_t, a_t) - \beta \log \pi(a_t | s_t)].$$

This soft value function accounts for both the expected rewards and the entropy of the policy, ensuring a balanced approach between exploration and exploitation. The actor's policy is updated by minimizing the following objective

$$J_\pi(\theta) = \mathbb{E}_{s_t \sim \mathcal{D}} [\mathbb{E}_{a_t \sim \pi_\theta} [\beta \log \pi_\theta(a_t | s_t) - Q_\psi(s_t, a_t)]]$$

This update adjusts the policy parameters to balance between maximizing expected returns and maintaining higher entropy, which encourages exploration and prevents the policy from becoming overly deterministic too quickly.

Additionally, SAC tackles the overestimation bias commonly found in traditional value-based methods by employing a *twin Q-network architecture*. This approach mitigates overestimation by computing the target value for the Q-function using the minimum of the two Q-values, ensuring more conservative value estimates and improving policy learning stability. The loss function for this twin Q-network update is

$$\mathcal{L}_Q(\psi) = \frac{1}{2} \mathbb{E}_{(s_t, a_t) \sim \mathcal{D}} \left[\left(Q_\psi(s_t, a_t) - \left(r(s_t, a_t) + \gamma \min_{i=1,2} Q_{\bar{\psi}_i}(s_{t+1}, a_{t+1}) \right) \right)^2 \right]$$

By taking the minimum value from the two Q-networks, this approach reduces overestimation bias, leading to more accurate value estimates.

Lastly, SAC operates as an off-policy algorithm, which improves sample efficiency by allowing the use of a replay buffer to learn from past experiences rather than relying solely on the most recent data. This off-policy nature, combined with entropy regularization, twin Q-networks, and a stochastic policy,

³For simplicity, the finite horizon version of the maximum entropy reinforcement learning objective is given. For the infinite horizon version, refer to [7].

⁴Depicted as α in [7], but replaced here with β to avoid confusion with the learning rate β mentioned in previous sections.

enables SAC to maintain a more stable and robust learning process than traditional Actor-Critic methods, especially in complex, high-dimensional environments. SAC not only accelerates convergence and enhances final performance on challenging tasks but also increases robustness against hyperparameter sensitivity and random seed variability. These innovations allow SAC to stabilize training, making it easier to achieve consistent results across different environments and tasks, thereby solidifying its position as a popular choice for a wide range of reinforcement learning applications, from robotic control to game playing [23].

2.2.4. Lyapunov Actor-Critic Algorithm

As discussed in the previous sections, while reinforcement learning algorithms like SAC have demonstrated significant potential in learning complex behaviours, particularly in high-dimensional environments without an available system model [23], they fall short in guaranteeing control stability. SAC improves learning stability, convergence, sample efficiency, and overall performance, but it does not inherently ensure the stability of the learned policies. Stability is critical in control tasks where safety and reliability are paramount, especially in real-world applications. This issue is even more pronounced in reinforcement learning, where the learned model's behaviour can be challenging to predict, particularly in unseen situations or when subjected to new environments, potentially leading to unexpected and dangerous outcomes.

To address these limitations, Han et al.[9] developed the LAC algorithm, which extends SAC by integrating Lyapunov stability principles discussed in Section 2.1 directly into the reinforcement learning framework. While SAC focuses on optimizing for rewards and entropy to encourage exploration, LAC adds the objective of ensuring system stability through the minimization of a Lyapunov function, $L(s)$. This function acts as an "energy-like" measure, quantifying how far the current state is from equilibrium. In LAC, rewards are replaced with positive definite costs to prioritize stability, aligning with the goal of minimizing deviations from equilibrium, whether static or dynamic. The primary objective in LAC is to design a policy that not only optimizes rewards and maintains sufficient exploration but also minimizes this "energy", ensuring that the system's trajectories remain bounded and reliably converge to the desired equilibrium, even amidst uncertainties and disturbances.

LAC achieves this by introducing a Lyapunov critic, $L_c(s, a)$, a neural network designed to approximate the Lyapunov function $L(s)$. To satisfy the first Lyapunov condition, $L_c(s, a)$ is parameterized to be positive definite, using the form $L_c(s, a) = f_\phi(s, a)^\top f_\phi(s, a)$, where f_ϕ is the output of a fully connected neural network with parameters ϕ . The Lyapunov critic is then trained to satisfy sample-based versions of the Lyapunov conditions, analogous to those in Theorem 2.1, ensuring that the Lyapunov function decreases over time—a crucial requirement for guaranteeing stability. This data-driven approach allows LAC to enforce stability through empirical data rather than requiring a fully derived Lyapunov function or an explicit model of the system dynamics. Instead, the Lyapunov function is learned during the reinforcement learning process, leveraging a known Lyapunov candidate to guide the optimization. Valid Lyapunov candidates include the value function for infinite-horizon cases and the sum of cost over a finite time horizon, both of which have been proven effective in this context. The objective function for the Lyapunov critic is expressed as

$$J(L_c) = \mathbb{E}_{s_t, a_t \sim \mathcal{D}} \left[\frac{1}{2} (L_c(s_t, a_t) - L_{\text{target}}(s_t, a_t))^2 \right],$$

where L_{target} is derived from one of these Lyapunov candidate functions. In addition to the Lyapunov critic, the LAC algorithm incorporates a constraint on the policy update to enforce stability during learning. This constraint is implemented using a Lagrange multiplier method, where the policy is optimized by minimizing the following objective function

$$J(\theta) = \mathbb{E}_{(s, a, s', c) \sim \mathcal{D}} [\beta (\log \pi_\theta(f_\theta(\epsilon, s) \mid s) + H_t) + \lambda (L_c(s', f_\theta(\epsilon, s')) - L_c(s, a) + \alpha_3 c)],$$

where β is the temperature parameter, λ is the Lagrange multiplier associated with enforcing the stability constraint, and α_3 controls the strictness of the stability constraint. The policy parameters θ are updated using SGA by maximizing the **negative** of this objective function, similar to other policy-based methods, while ensuring adherence to the Lyapunov stability criterion throughout the learning process. By satisfying the Lyapunov conditions, the LAC framework actively steers the system towards stability, thereby producing policies that are stable in mean cost. Like the temperature parameter β in SAC, the

λ parameter is dynamically optimized during training, ensuring stability constraints are met while allowing for sufficient exploration and performance improvement. This update of λ is critical as it balances the trade-offs between performance, exploration, and adherence to stability constraints, dynamically tightening the stability guarantees as the policy learns.

This approach not only enhances the stability of the learned policies but also provides robustness against uncertainties and disturbances. By guiding both policy and value updates with a Lyapunov function, LAC ensures that the system's state trajectories remain bounded and reliably converge, making it a powerful tool for control tasks that require safety, reliability, and generalizability to unseen states and tasks. Empirical results presented by Han et al.[9] demonstrate LAC's effectiveness, showing superior performance in various control tasks such as CartPole, HalfCheetah, and synthetic biological gene networks (GRN), consistently outperforming SAC in terms of both stability and robustness. For a deeper understanding of the LAC algorithm, including its theoretical foundations and performance evaluations, refer to Han et al.[9]'s original work [9].

3

Method

This chapter outlines the methodologies employed in our research, each precisely tailored to address specific aspects of our central research questions. The original intention of this study was to conduct a novel *Panda Effort Control Experiment*, which sought to apply the LAC algorithm developed by Han et al.[9] and the SAC algorithm developed by Haarnoja et al.[7] to a simulated effort-controlled Panda Emika Franka robot. This experiment was designed to diverge significantly from prior work by focusing on effort control rather than the position control used in Han et al.[9]’s original study, allowing us to explore the algorithms’ stability, robustness, and generalization in scenarios devoid of inherent disturbance management. However, during the *Reproduction Study*, we encountered several discrepancies and inconsistencies between the published details of Han et al.[9]’s work and the actual codebase, particularly in algorithmic details and hyperparameters (as discussed in Section 3.1.2 and Appendix A). Resolving these issues consumed a substantial amount of time, which, combined with the limited availability of computational resources needed for training the LAC and SAC algorithms in the simulated Panda Emika Franka robot, ultimately prevented us from executing the *Panda Effort Control Experiment* within the scope of this research. Consequently, the *Panda Effort Control Experiment*, along with the proposed method and implemented codebase, is detailed in appendices B and C, providing a solid foundation for future exploration by other researchers. This report focuses primarily on the *Reproduction Study* and the challenges encountered therein, with the *Panda Effort Control Experiment* remaining an open task for future investigation.

3.1. Reproduction Study

In the *Reproduction Study*, our primary objective is to replicate and extend the findings of Han et al.[9]. This crucial phase aims to validate the original results and sets the foundation for the innovative methodologies introduced in the *Panda Effort Control Experiment*. The subsequent sections detail the essential steps undertaken in the *Reproduction Study*, beginning with the replication of the research environments described by Han et al.[9]. After that, the process of translating Han et al.[9]’s TensorFlow 1 (TF1) codebase to PyTorch is described. Next, we outline the core of our *Reproduction Study*, which involved an extensive hyperparameter tuning process to assess the impact of the α_3 hyperparameter on the LAC algorithm, along with a comparison of the translated LAC and SAC algorithms against the original results. The chapter concludes with a discussion of the additional hyperparameter experiment conducted to investigate the impact of adding an extra layer to the LAC critic network, which was undertaken in response to a discrepancy identified in Han et al.[9]’s published research.

3.1.1. Replication of Research Environments

Our *Reproduction Study* began by re-creating the research environments used by Han et al.[9]. These environments, which are modified versions of the renowned Gymnasium (originally OpenAI Gym) environments, were adapted by Han et al.[9] to support the LAC algorithm and their emphasis on stability-focused research. We chose the CartPole, GRN, CompGRN, and FetchReach environments from Han et al.[9]’s study. The first three environments were selected for their critical role in exploring the LAC algorithm’s stability, robustness, and generalizability. The FetchReach environment was chosen

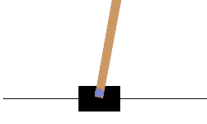
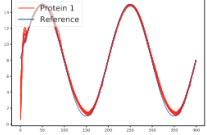
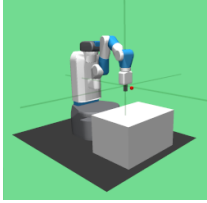
Environment	Description	Key Features
	The CartPole environment features a pole attached by an un-actuated joint to a cart that moves along a frictionless track. The goal is to keep the pole balanced at the screen's centre by applying left and right forces to the cart.	<ul style="list-style-type: none"> Action space: Continuous force $u \in [-20, 20]$ on the cart in \mathbb{R}^1. Observation space: Vector in \mathbb{R}^4 containing cart position x, velocity \dot{x}, pole angle θ, and angular velocity $\dot{\theta}$. Initial state: Uniformly sampled from $[-0.2, 0.2]^4$. Episode length: $t = 250$ steps. Cost function: $c = (\frac{x}{x_{threshold}})^2 + 20 * (\frac{\theta}{\theta_{threshold}})^2$. Termination condition: $\theta > \pm 20^\circ$, $x > \pm 10$, $t = 250$. Termination cost: $c = 100$.
	The GRN environment simulates a repressilator, a synthetic network with three (or four in the CompGRN variant) gene circuits that oscillate in behaviour. Agents adjust light signals to control gene expression, aiming to match a protein's concentration to a specific reference .	<ul style="list-style-type: none"> Action space: Light signal intensity $u \in [0, 1]$ in \mathbb{R}^3. Observation space: Vector in \mathbb{R}^8 containing protein concentrations m, p, reference signal r and error r_e. Initial state: Uniformly sampled from $[0.0, 5.0]^6$. Episode length: $t = 400$ steps. Cost function: Mean squared error between protein concentration and reference $c = (p_1 - r)^2$. Termination condition: Exceeding max cost or step limit. Termination cost: $c = 100$
	The FetchReach environment showcases a 7-DoF Fetch Mobile Manipulator simulation, complete with a two-fingered parallel gripper. Control is achieved through Cartesian displacements, with the primary goal being to navigate the end effector to a randomly assigned target position within the robot's operational workspace.	<ul style="list-style-type: none"> Action space: Cartesian displacement and gripper control $u \in [-1, 1]$ in \mathbb{R}^4. Observation space: Dictionary of the robot's end effector state, gripper state, and achieved/desired goals. Initial state: Predefined gripper position and orientation. Episode length: $t = 200$ steps. Cost function: Euclidean distance between achieved and desired goals $c = \ g_{desired} - g_{achieved}\ _2$. Termination condition: Exceeding step limit. Termination cost: Final cost function value.

Table 3.1: Summary of key environments selected for our *Reproduction Study*, originally adapted from OpenAI Gym (now evolved into *Gymnasium*) by Han et al.[9] for their research. These environments were chosen for their relevance in assessing the validity of our implementations, particularly within the context of the *Panda Effort Control Experiment*. Detailed information on these adaptations is available in the [stable-gym](#) documentation and in Han et al.[9]'s publication appendix.

because it closely resembles the PandaReach environment to be used in our *Panda Effort Control Experiment*, providing a valuable reference for its hyperparameter selection. To ensure our experimental setup accurately mirrors that of Han et al.[9], we rigorously validated our environments against their original implementations, as detailed in Section A.1.1 of Appendix A. This meticulous process confirmed our successful recreation of these environments and their fidelity to the original research. The key features of these environments, included in our `stable_gym`¹ package, are outlined in Table 3.1. A comprehensive description can be found in the appendix of Han et al.[9]'s paper or the documentation of the `stable_gym` package. An overview of our research codebase, including the `stable_gym` package, is provided in Appendix C.

3.1.2. Translation of the Algorithms

Following the successful replication of research the environments, our next step in the *Reproduction Study* was to translate Han et al.[9]'s TensorFlow 1 (TF1) codebase² into PyTorch, prompted by the deprecation of TF1. We chose PyTorch over TensorFlow 2 (TF2) due to its enhanced user-friendliness and comparable performance. Our translation effort aimed to faithfully reimplement the LAC and SAC algorithms, adhering as closely as possible to the mathematical formulations and hyperparameters documented in the original study.

¹The translated environments in the `stable_gym` package can be found at <https://github.com/rickstaa/stable-gym> (last accessed 21-01-2024).

²The original TensorFlow 1 codebase by Han et al.[9], used for our translation, is available at <https://github.com/hithmh/Actor-critic-with-stability-guarantee> (last accessed 21-01-2024).

During the translation process, we encountered several discrepancies and inconsistencies in algorithm details and hyperparameters between the published work of Han et al.[9] and their codebase. In cases of ambiguity, we gave precedence to the specifications outlined in the publication, considering them the authoritative source. Where the publication lacked sufficient detail, we consulted the codebase to bridge the gaps, striving to remain as faithful as possible to the original study. To validate our translation’s fidelity, extensive testing was conducted, comparing our implementation against the original results. This testing was critical for ensuring that our versions of the algorithms produced outcomes consistent with those reported by Han et al.[9], thereby affirming the accuracy of our translation efforts. A detailed account of the translation efforts, the challenges faced, the strategies employed to overcome them, and the validation steps taken to ensure accuracy is documented in Appendix A. Upon completion, the translated algorithms were incorporated into the `stable_learning_control` package³, showcasing our commitment to accurate replication and resolving observed discrepancies through rigorous validation and testing.

After completing the translation process, we initiated a preliminary assessment to ensure the translated LAC and SAC algorithms’ performance was consistent with the outcomes reported by Han et al.[9]. Each algorithm was trained ten times with a different random seed, involving one million interactions within the CartPole environment. We observed that the SAC algorithm demonstrated satisfactory performance and convergence, similar to the original study. For the LAC algorithm, however, while the performance and convergence met expectations, the average lambda value (λ), which is a critical indicator of adherence to the stability constraint, did not converge to zero. This contradicted the results reported by Han et al.[9] for the LAC algorithm, prompting a thorough investigation. We identified a potential typographical error in the original paper concerning the α_3 hyperparameter for the Lyapunov function. The original study stated $\alpha_3 = 1.0$, whereas the codebase used a value of 0.1. This inconsistency suggested that an α_3 value of 1.0 might be too conservative for the algorithm to satisfy within the training duration. Consequently, we recognized the need for a systematic hyperparameter tuning process for α_3 , detailed in the following subsection.

3.1.3. Alpha3 Hyperparameter Tuning

In response to the discrepancies identified during our preliminary assessments, we initiated a hyperparameter tuning process focused on the α_3 parameter, spanning across all key environments utilised in our *Reproduction Study*: CartPole, GRN, CompGRN, and FetchReach. Notably, we introduced a novel infinite horizon variant of the FetchReach environment, which was not present in Han et al.[9]’s original study, anticipating its relevance for the *Panda Effort Control Experiment*. We opted for the GridSearch method due to its straightforward approach, bypassing the complexity of more advanced alternatives like Ray Tuner [15] for the sake of a singular parameter’s tuning.

Our selection of α_3 values for the tuning process spanned from 0.1 to 1.5, incremented by 0.1. This range was strategically chosen to explore values around the 1.0 mark specified by Han et al.[9] while also considering that a value of 0 would negate the stability constraint. Given the constraints of training time and computational resources, we chose this incremental approach to balance the depth of evaluation and practicality. For each α_3 value, we performed five training iterations with different random seeds to account for variability, each comprising the total steps outlined in Han et al.[9]’s study for each environment. Although research of Colas et al.[3] suggests that a larger number of seeds might bolster statistical validity, our constraints limited us to five, a compromise deemed sufficient for validating Han et al.[9]’s findings within the bounds of our study’s feasibility. The entire tuning process was conducted in parallel on two local servers, each with three Nvidia 1070 TI GPUs, totalling approximately 60 hours.

Performance evaluation during the hyperparameter tuning process was comprehensive, involving assessments of the following metrics: *final average test performance*, *test performance convergence speed*, and λ *convergence*, with **lower costs indicating better performance**. The test performance was calculated by averaging the cost obtained over 10 independent trials in the environment, where the agent acted according to the policy learned after each training epoch. Each epoch involved 2048 environment interactions during training, but the performance evaluation itself was conducted separately from training to reflect the policy’s effectiveness in an independent test setting. The *final average test performance* was determined by averaging the test performance over the last N epochs, chosen to reduce noise and ensure stability in performance evaluation. This approach was applied differently across

³The translated algorithms in the `stable_learning_control` package can be found at <https://github.com/rickstaa/stable-learning-control> (last accessed 21-01-2024).

environments: $N = 25$ for CartPole to account for its extended training over 489 epochs, and $N = 10$ for FetchReach, GRN, and CompGRN, reflecting their shorter training spans of 147 and 49 epochs respectively. The determination of N values was guided by a preliminary analysis of convergence trends to optimally reduce noise while capturing a consistent portrayal of performance at convergence. The *test performance convergence speed* was established by pinpointing the initial epoch where the algorithm’s cost not only achieved but also consistently stayed beneath a predetermined convergence threshold. This threshold was established at 95% of the total convergence, which is calculated as the difference between the maximum observed cost and the final average test performance. To smooth performance variations and ensure a stable measure of achievement, a 5-epoch moving average on the cost data was employed, chosen for its optimal balance between sensitivity to genuine performance improvements and resilience against transient spikes. Lastly, the evaluation of λ *convergence* involved examining the convergence rate of the λ value towards zero, underscoring the algorithm’s alignment with the stability constraint. This evaluation drew upon the same principles applied to assess *final average test performance* and *test performance convergence speed*, examining the final average λ value across the last N epochs alongside its convergence rate. This dual analysis checks for the algorithm’s consistent compliance with stability prerequisites and gauges the efficiency with which it achieves this compliance.

Employing these metrics mentioned above, our analysis aimed at identifying an optimal α_3 value conducive to stability and practical learning. In adherence to Han et al.[9]’s methodology, we maintained consistency by altering only the α_3 hyperparameter, leaving all others unchanged as detailed in Table E.1 within Appendix E. The detailed outcomes of this tuning process, including the selected α_3 values for each environment and their consequential effects on algorithm performance, are elaborated in Section 4.1.1.

3.1.4. Convergence and Performance Evaluation

Following the hyperparameter tuning process, we evaluated the accuracy of our translated LAC and SAC algorithms. This evaluation aimed to compare the convergence rates and overall performance of both algorithms, focusing on how they perform relative to each other and to the original results presented by Han et al.[9]. To ensure a thorough comparison, we conducted tests across all selected α_3 values, with an emphasis on the key environments used in Han et al.[9]’s study. In each of these environments, we evaluated the results of the LAC algorithm obtained during the hyperparameter tuning, alongside those of the SAC algorithm. The metrics used to assess performance were the same as those employed during tuning: *test performance convergence speed* and *final average test performance*. Convergence was measured by the rate and consistency with which the algorithms stabilized to a steady average cost during training. Meanwhile, performance was evaluated based on the final average test performance, where a lower cost signified better outcomes. These metrics provided a clear basis for assessing both the efficiency of each algorithm in achieving stable solutions and their effectiveness in minimizing costs. To ensure the reliability of our comparison, we conducted 10 training sessions for the SAC algorithm, each initialized randomly. This approach maintained consistency with Han et al.[9]’s study, as we used the same number of environment interactions per session. By aligning our experimental design with the original study, we ensured that our results were directly comparable and offered meaningful insights into any potential improvements or discrepancies between the original and translated algorithms. An analysis of the results, along with a comparison of our findings with those of Han et al.[9], will be presented in Section 4.1.2 of the next chapter.

3.1.5. Additional LAC Critic Layer Experiment

Lastly, during our reproduction we identified a discrepancy in the critic network architecture between the SAC and LAC algorithms as implemented by Han et al.[9]. Specifically, while the SAC critic network adhered to the original [256, 256] architecture documented by Haarnoja et al.[7], the LAC critic network deviated by incorporating an additional output layer, resulting in configurations of either [256, 256, 16] or [64, 64, 16]. Given that this architectural variation could impact the algorithms’ performance and convergence, we conducted an additional experiment within our *Reproduction Study* to assess its effect. The goal was to ensure that the comparison between the LAC and SAC algorithms was fair and unbiased, thereby providing a solid foundation for the proposed *Panda Effort Control Experiment*. This experiment involved modifying the SAC critic network to include an additional output layer, thereby aligning it with the LAC architecture. The results of this experiment are presented in Section 4.1.3 of the Results

chapter, providing insights into the impact of this architectural variation on the algorithm's performance and convergence.

4

Results

This chapter presents the findings of the *Reproduction Study*, which replicated and validated the LAC algorithm introduced by Han et al.[9]. Our analysis focuses on the impact of the α_3 hyperparameter on the algorithm’s performance and convergence capabilities. Additionally, we compared the LAC algorithm’s results with those of the SAC algorithm to ensure the accuracy of our reimplementation and verify that the results reported by Han et al.[9] were reproducible. Due to time constraints, the experiments focusing on stability, robustness, and generalizability, as conducted by Han et al.[9], were not included in this study. The following sections provide detailed outcomes of the experiments that were conducted, offering key insights into the behaviour of both algorithms under various configurations. Additional experiments performed due to inconsistencies identified in Han et al.[9]’s work are presented in Appendix A. Together, the results discussed here and the lessons learned during the reimplementation provide a strong foundation for future work, including the proposed *Panda Effort Control Experiment*.

4.1. Reproduction Study

4.1.1. Alpha3 Hyperparameter Tuning

4.1.1.1. Performance and Convergence Analysis

Following the methods outlined in Section 3.1.3, our hyperparameter optimization experiment aimed to identify the optimal α_3 value for enhancing the LAC algorithm’s performance and convergence speed across key environments, including CartPole, GRN, CompGRN, and FetchReach. The results are presented in Figure 4.1, which shows the algorithm’s average performance, measured by mean cost, and its convergence behaviour across varying α_3 values. A more detailed breakdown of the *final average test performance* and *test performance convergence speed* can be found in Tables E.7 to E.10 in Appendix E.2.1.3. Additionally, extended figures for α_3 values from 0.1 to 1.5 are available in Appendix E.1.1.2, providing a broader view of the algorithm’s behaviour.

Across most environments, the LAC algorithm demonstrated robust performance and convergence across a wide range of α_3 values, as indicated by the narrow confidence intervals in Figure 4.1. This consistency highlights the algorithm’s ability to achieve strong performance and rapid convergence regardless of the α_3 value. For instance, in the CartPole environment, the *final average test performance* ranged from 29.14 to 33.82, with a low variance, as shown in Table E.7. Similarly, the *test performance convergence speed* remained stable, typically reaching 95% of total convergence within 110 to 239 epochs (i.e. $0.22e6 - 0.48e6$ steps). Similar trends were observed in the FetchReach environments, further indicating the algorithm’s robustness across different α_3 values. Moreover, the novel infinite horizon FetchReach variant showed performance and convergence similar to the finite horizon version, underscoring the LAC algorithm’s robustness in robotic tasks.

In contrast, performance in more complex environments such as GRN and CompGRN was more sensitive to the choice of α_3 . These environments involve tracking tasks, where the agent must follow a reference signal, and the stability constraint of the LAC algorithm plays a crucial role in guiding the agent toward the desired behaviour. In the GRN environment, a significant increase in mean cost was observed for both $\alpha_3 = 0.1$ and $\alpha_3 = 0.2$, with the *final average test performance* reaching 723.7 and 320.78, respectively (Table E.8). These values sharply contrast with the much lower mean cost

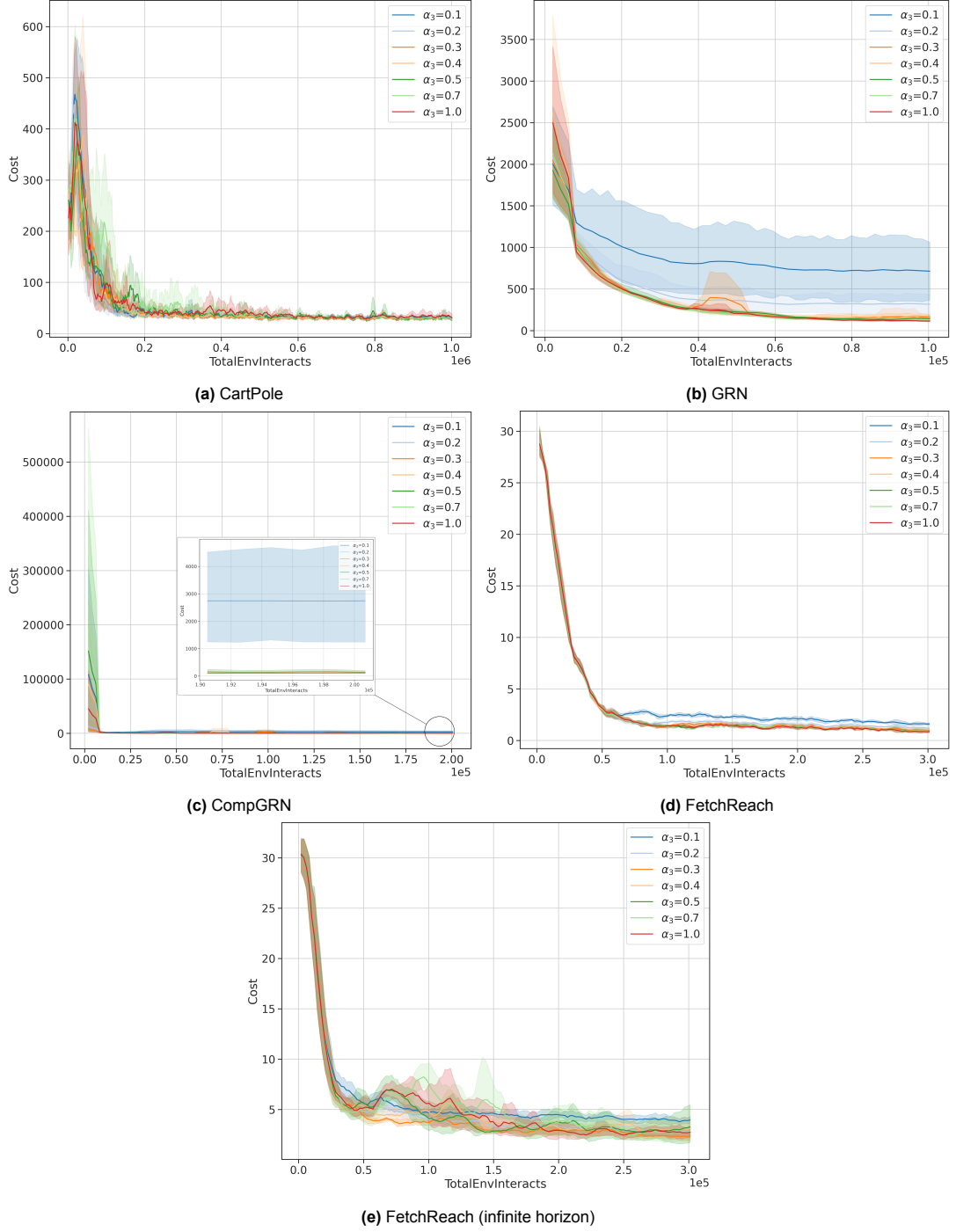


Figure 4.1: Average test performance, measured as mean cost, and convergence of the LAC algorithm for select α_3 values in different environments. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals. For clarity, only key α_3 values are shown. Detailed data for all α_3 values (0.1 to 1.5) and further statistical analysis are available in Sections E.1.1.2 and E.2.1.3 of Appendix E.

observed for other α_3 settings (e.g., 134.99 ± 17.15), though the impact at $\alpha_3 = 0.2$ was less severe. Additionally, the *test performance convergence speed* for these values was slower, and there was a more significant variance between random seeds, as reflected in the standard deviations, suggesting that the algorithm may have struggled to avoid local minima in some cases. This effect was especially pronounced for $\alpha_3 = 0.1$, while $\alpha_3 = 0.2$ exhibited similar but milder issues. A similar pattern was observed in the CompGRN environment, suggesting a strong sensitivity of the algorithm to lower α_3 values in tracking tasks that rely heavily on stability constraints. Further experiments with five additional seeds (Appendix E.1.1.1.1 and E.2.1.2.1) confirmed the persistent higher costs for both $\alpha_3 = 0.1$ and $\alpha_3 = 0.2$ in GRN and CompGRN, ruling out random variance. Although time constraints limited further testing, the available data suggest that lower α_3 values (particularly $\alpha_3 = 0.1$) are suboptimal for complex, dynamic environments where stability constraints are crucial.

To verify that the observed results were not due to translation errors or implementation issues, we conducted additional experiments using the original codebase from Han et al.[9]. As shown in Appendix E.1.1.1.7, the outcomes of these experiments were consistent with our findings, affirming that the observed performance and convergence behaviour were not due to implementation errors. This consistency strengthens our confidence that the effect of α_3 on the LAC algorithm does not significantly impact performance and convergence in most environments. However, in environments like GRN and CompGRN, where the stability constraint plays a pivotal role due to the stability-based cost function, lower α_3 values (e.g., 0.1-0.2) may hinder performance and convergence.

4.1.1.2. Stability Assessment via Lambda Convergence

Understanding the convergence behaviour of λ during training is crucial for assessing the stability of the LAC algorithm across various environments. In this context, λ acts as a slack variable that indicates how well the agent has learned a valid Lyapunov function, which in turn reflects the stability of the control policy. Convergence to lower λ values suggests that the stability constraint is being met, and the algorithm is operating within safe limits. Figure 4.2 illustrates the convergence of λ values for different α_3 settings across key environments, demonstrating the algorithm’s adherence to these stability constraints. As shown in the figure, the speed of convergence and the final λ value are strongly influenced by the choice of α_3 , with higher α_3 values leading to slower convergence and higher final λ values.

This effect is particularly prominent in the GRN and CompGRN environments, where α_3 plays a critical role in stability. In the GRN environment, only α_3 values of 0.1 and 0.2 demonstrated successful convergence, while higher values failed to reduce λ to zero. In the more complex CompGRN environment, only $\alpha_3 = 0.1$ managed to achieve any significant convergence. This suggests that lower α_3 values are more suitable for providing stable policies in highly dynamic environments, especially within the current training duration. In contrast, environments such as CartPole and FetchReach exhibit successful convergence for α_3 values between 0.1 and 0.3, showing that these settings are generally effective for simpler tasks. These results align with theoretical expectations, as higher α_3 values introduce more conservative stability constraints, complicating optimization and delaying convergence.

It is important to note that incomplete λ convergence for higher α_3 values does not inherently signify instability. Instead, λ serves as a slack variable that indicates the extent to which the stability constraint has been satisfied. A non-zero λ value suggests partial satisfaction of the stability constraint. Therefore, it is likely that the current parameter settings in our experiments did not allow the algorithm to train for a sufficient number of steps to fully satisfy the stability constraint. An additional experiment in the GRN environment demonstrated that λ converged to zero only after $2e5$ steps (see Figure E.1.1.1.2 in Appendix E), indicating that prolonged training may be necessary for higher α_3 values to fully converge.

These results conflict with the findings in Han et al.[9], where an α_3 value of 1.0 was reported. We strongly suspect this to be a typographical error, as our results show that $\alpha_3 = 1.0$ does not converge to a stable λ value in any environment except FetchReach. It is therefore unlikely that this value could reproduce the λ convergence plots in Figure S4 of their study. To further investigate, we tested the original codebase from Han et al.[9], which exhibited similar behaviour, with higher α_3 values struggling to achieve λ convergence (see Section E.1.1.1.7). This strengthens our hypothesis that a typographical error may have occurred in reporting α_3 values in their study. While non-convergence of λ is not inherently problematic, as λ serves as a safeguard for stability, it becomes critical when λ exceeds 1.0. In our implementation, we clip λ at 1.0 to prevent runaway growth due to early stability violations, which could lead to inappropriate policy updates. This issue is particularly evident in the CompGRN

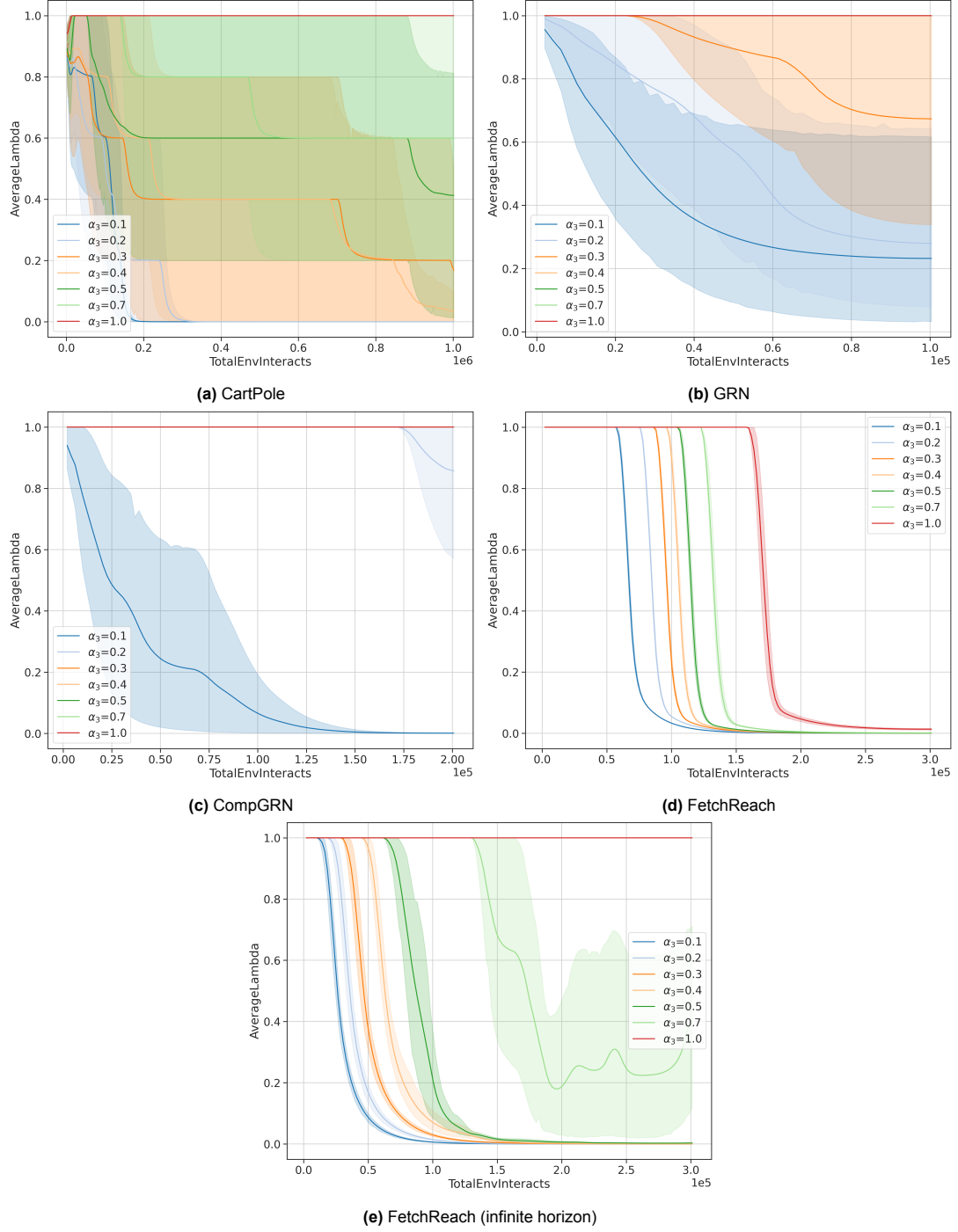


Figure 4.2: Convergence of λ values during LAC algorithm training across various environments for select α_3 values, illustrating adherence to the stability constraint. Each line represents the mean λ value across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals. For clarity, only key α_3 values are shown. Detailed data for all α_3 values (0.1 to 1.5) and further statistical analysis are available in Sections E.1.1.3 and E.1.1.3 of Appendix E.

environment, where only $\alpha_3 = 0.1$ achieved full convergence. This clipping behaviour may result in invalid policy updates, further risking instability. Therefore, accurately reporting α_3 values is crucial for ensuring the stability of the LAC algorithm.

4.1.2. Comparison of LAC and SAC Performance and Convergence

Having examined the appropriate α_3 values for the LAC algorithm, we can now directly compare its performance and convergence with the SAC algorithm. This comparison is crucial for verifying the accuracy of our reproduction of Han et al.[9] and confirming the reproducibility of their reported results. Figure 4.3 and the associated tables in Appendix E show the performance of LAC across selected α_3 values, in comparison with the SAC algorithm. As illustrated, both the LAC and SAC algorithms are able to converge to stable policies in all environments with similar performance. LAC, however, shows a slight performance edge in the GRN, CompGRN, and FetchReach environments, exhibiting lower *final average test performance* and faster *test performance convergence*. This was also noted by Han et al.[9], likely because the GRN and CompGRN environments involve tracking tasks, where the Lyapunov constraint helps steer the agent toward the desired behaviour. Additionally, we observed a clear performance advantage for LAC over SAC in the finite horizon case of the FetchReach environment, which was less pronounced in their study. Another notable divergence from Han et al.[9]’s findings occurred in the CartPole environment, where, in our research, SAC exhibited faster convergence than expected. These differences likely result from design choices we made due to inconsistencies or missing data in Han et al.[9]’s study, as well as different random seed selections. Despite these differences, our results validate the accuracy of our LAC implementation and affirm the general validity of Han et al.[9]’s conclusions. Overall, LAC and SAC perform similarly in terms of cost and convergence speed, with slight variations across specific environments. These findings provide confidence in the accuracy of our reimplementation and the reproducibility of Han et al.[9]’s results, laying a solid foundation for future work, including the proposed *Panda Effort Control Experiment*.

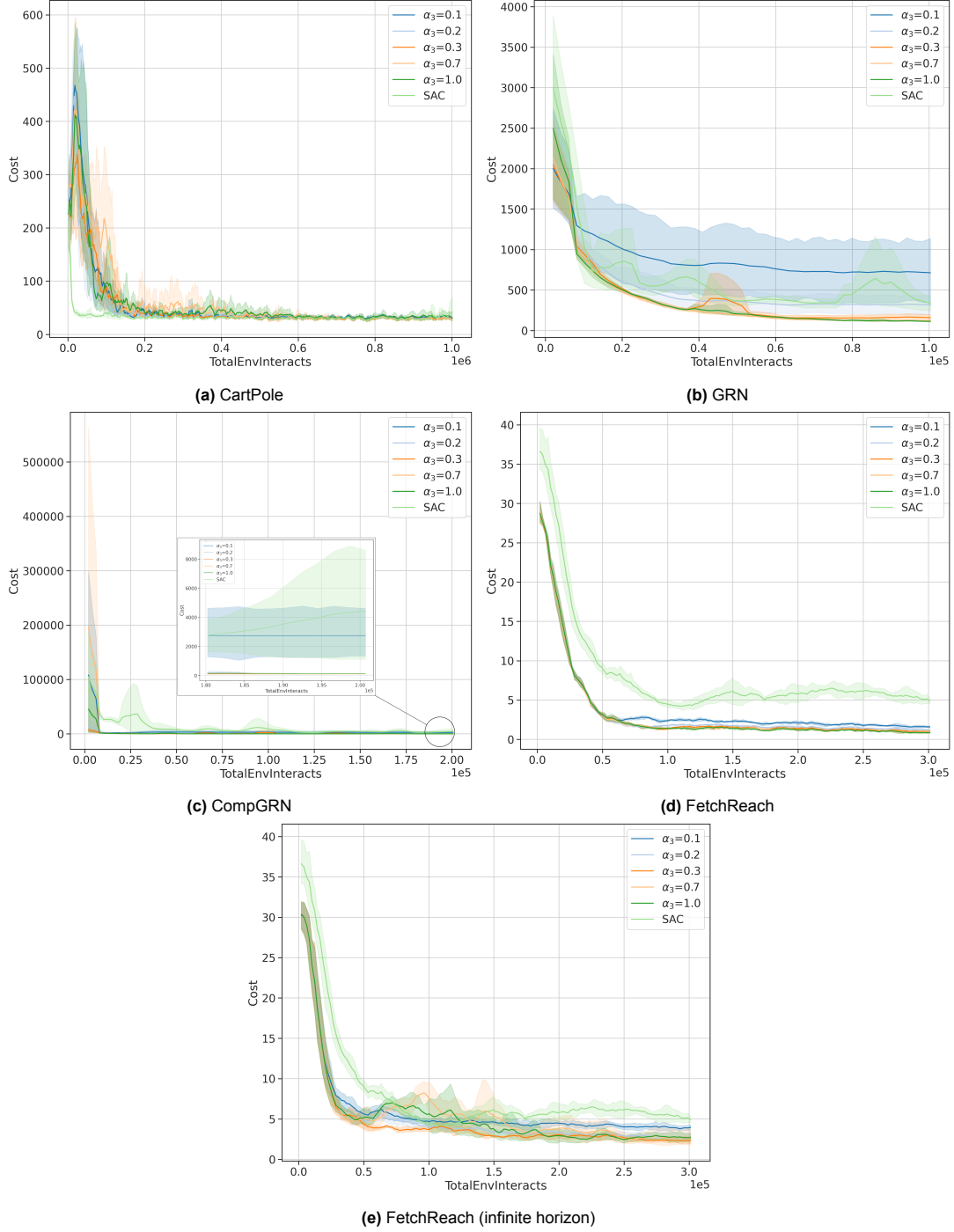


Figure 4.3: Average test performance, measured as mean cost, and convergence of the **LAC** and **SAC** algorithm for select α_3 values in different environments. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals. For clarity, only key α_3 values are shown. Detailed statistics about these figures are available in Sections E.1.1.2 and E.2.1.3 of Appendix E.

4.1.3. Impact of Additional Critic Layer on LAC Performance

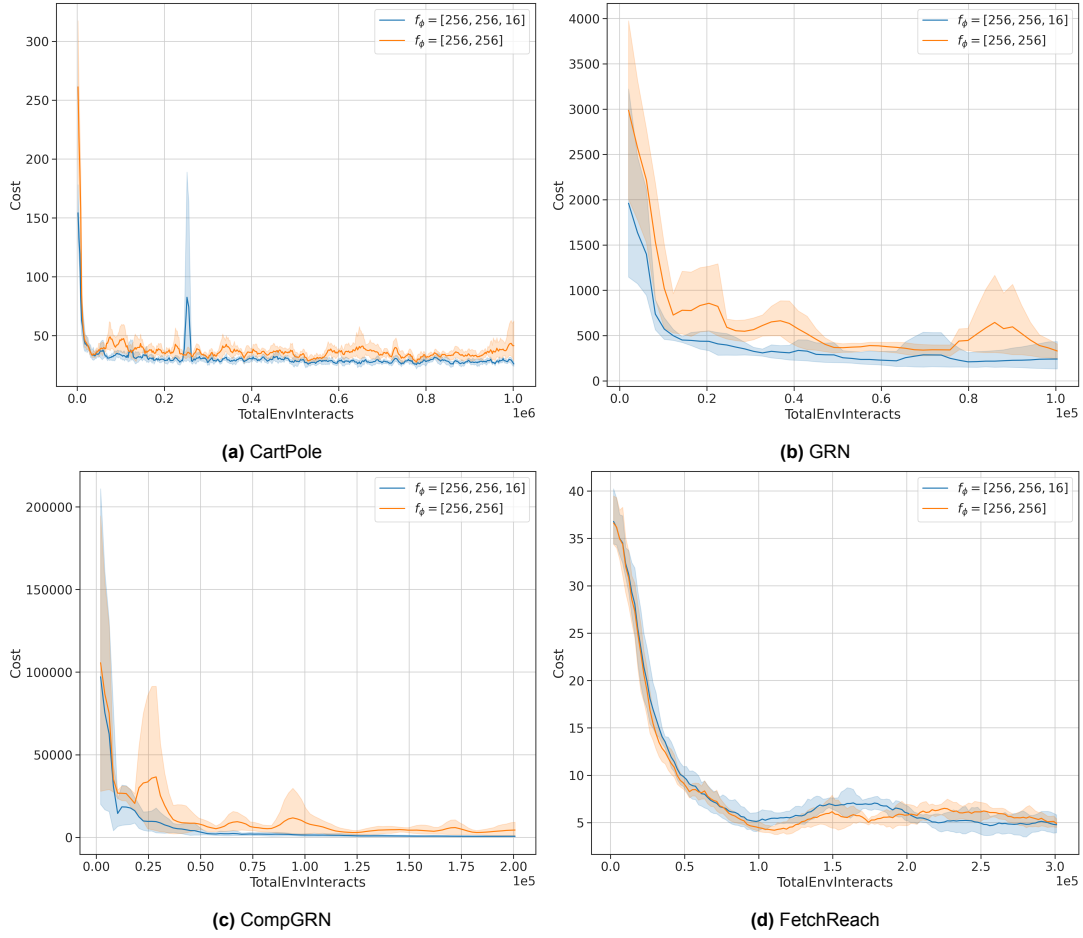


Figure 4.4: Average test performance, measured as average cost, and convergence of the **SAC** algorithm between *two different critic network architectures* in various benchmark environments. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals. Tables containing the statistical analysis of these figures are available in Sections E.2.1.6 of Appendix E.

Figure 4.4 and the associated Tables in Appendix E.2.1.6 illustrate the results of adding an extra layer to the SAC critic network. In the CartPole environment, the additional layer led to a slight improvement in performance, while more significant gains were observed in the GRN and CompGRN environments. For FetchReach, the performance difference was minor and within the expected range of variability across different seeds. These results suggest that the extra layer in the SAC critic network positively impacts performance in more complex environments like GRN and CompGRN, but provides only modest gains in simpler environments like CartPole. Overall, the additional critic layer enhanced SAC’s performance in complex environments but was insufficient to outperform LAC in any environment (see Figure 4.3). Despite the layer’s impact, SAC and LAC still performed comparably in terms of *final average test performance* and *test performance convergence speed*. However, a more direct comparison with LAC could have been made if the SAC architecture had also included the extra layer from the outset.

5

Discussion

This study aimed to evaluate the stability and robustness of the LAC algorithm in reinforcement learning by reproducing and validating the results originally reported by Han et al.[9]. Through this *Reproduction Study*, several key insights emerged, particularly concerning the algorithm’s performance across different settings. This study also highlights broader challenges related to reproducibility in reinforcement learning research, emphasizing the importance of precise hyperparameter tuning, transparent reporting, and consistent environment configurations.

The primary objective was to assess the reproducibility of Han et al.[9]’s results in new simulated environments, with a particular emphasis on comparing the performance and stability of LAC and SAC. Specifically, we focused on key parameters like α_3 and their influence on learning stability, convergence speed, and the convergence of the λ parameter, which enforces the stability constraint in LAC. During the reproduction process, we encountered notable discrepancies between the published details and the available codebase, particularly in algorithmic details and hyperparameters, such as network architectures and horizon lengths. These discrepancies necessitated adjustments and careful tuning to ensure that our study aligned with the experimental conditions and outcomes reported by Han et al.[9].

The results of this *Reproduction Study* highlighted several key findings, notably the critical influence of the α_3 parameter on the algorithm’s performance, its convergence speed, and the convergence behaviors of the λ parameter. The study successfully demonstrated that the LAC algorithm maintains robust performance across a variety of environments, particularly in simpler settings like CartPole and FetchReach, where variations in α_3 had minimal impact on performance and convergence speed. However, in more complex environments such as GRN and CompGRN, which require high stability for tracking tasks, the influence of α_3 became markedly pronounced. Lower α_3 values in these environments led to significantly worse performance and slower convergence speeds, underscoring the importance of tuning α_3 to enhance the stability of the learning process. Higher α_3 settings, while improving performance and adherence to stability constraints, necessitated extended periods to effectively learn the Lyapunov function. This illustrates a critical trade-off: enhancing stability with higher α_3 values improves performance but at the cost of increased training duration. This dynamic was also reflected in the convergence behaviors of the λ parameter. In environments where stability is paramount, like GRN and CompGRN, higher α_3 values led to slower λ convergence and higher final λ values within the fixed training steps. This indicates that more conservative α_3 settings, though beneficial for achieving desired stability, complicate the optimization process, extending the time required for the agent to adapt and stabilize its policy under the stricter Lyapunov constraints. These findings highlight the complexity of deploying reinforcement learning algorithms in scenarios that demand both high performance and stringent stability. They emphasize the necessity for careful hyperparameter tuning, particularly of α_3 , to balance efficiency and effectiveness, ensuring that the LAC algorithm can perform reliably across diverse operational environments.

In our reproduction study, when comparing SAC and LAC, our overall results aligned with those reported by Han et al.[9], reinforcing the consistency and reliability of the LAC algorithm across different environments. Interestingly, however, some discrepancies were observed. Notably, the SAC algorithm exhibited faster convergence than LAC in simpler environments like CartPole, a contrast to Han

et al.[9]’s findings. Additionally, our results highlighted a clear performance advantage for LAC over SAC in the FetchReach environment, a divergence that was less pronounced in their study. These discrepancies between our findings and those of Han et al.[9] may stem from necessary adjustments made due to inconsistencies or missing data in the original study. Our adaptations aimed to address these gaps and ensure that our experimental setup accurately reflected the conditions under which these algorithms were expected to perform. This underscores the importance of transparent reporting and precise experimental configuration in reproducing and validating results in reinforcement learning research, as also noted by Colas et al.[2]. Even minor discrepancies in experimental setups can lead to significant variations in outcomes, emphasizing the critical nature of meticulous methodological fidelity.

In addition to the main findings from our reproduction study, we conducted an exploratory analysis to examine the impact of network architecture differences on algorithm performance. We noted an inconsistency in the network architectures used by Han et al.[9]—specifically, an additional output layer in LAC’s critic network that was not present in SAC’s. To assess the effects of this discrepancy, we experimented with adding a similar layer to SAC’s architecture. This modification improved SAC’s performance, thereby underscoring the significant influence of such hyperparameters on comparative results. While this adjustment did not alter LAC’s overall superiority under the primary conditions of our study, it highlighted the critical importance of maintaining comparable architectures to ensure fairness in algorithm comparisons. This additional insight further reinforces the need for meticulous accounting of hyperparameter settings when evaluating the performance and outcomes of different reinforcement learning algorithms, amplifying the earlier point about the importance of methodological fidelity.

Having established the reproducibility of the LAC algorithm and elucidated the critical role of hyperparameters such as α_3 in different simulated environments, our study sets a crucial foundation for extending the work of Han et al.[9] to more complex applications. This research significantly contributes to the existing body of knowledge by addressing discrepancies and clarifying the conditions necessary for stability and performance, providing a detailed blueprint for future explorations in safe reinforcement learning. Specifically, it fills critical gaps in the original research by offering a comprehensive set of parameters and setup details essential for reproducibility in varied settings. The development of a robust and extensible codebase, coupled with our research, facilitates the extension of the LAC algorithm’s applications to new domains, such as the proposed *Panda Effort Control Experiment*. By providing these resources, our study not only supports theoretical exploration but also paves the way for further empirical studies, particularly in scenarios where real-world dynamics introduce modeling uncertainties and disturbances. This ensures that future applications of the LAC algorithm are grounded in a rigorously tested and stable framework, thereby enhancing the safety and effectiveness of reinforcement learning deployments. Our work contributes to advancing the field of reinforcement learning, enabling researchers to build upon solid, reproducible results and explore new frontiers in algorithm application.

Despite successfully reproducing many of Han et al.[9]’s results, our study has several limitations that could impact the broader applicability of our findings. One primary constraint was the reliance on a limited number of random seeds and a narrow range of hyperparameter settings, primarily due to computational constraints. Given the high sensitivity of reinforcement learning algorithms to these factors, future studies that explore a wider set of seeds and hyperparameters could provide a more thorough validation of the LAC algorithm’s performance across diverse settings. Our study primarily focused on the α_3 parameter and used the hyperparameters provided by Han et al.[9] as they were. While we assumed the original authors had optimized the settings for both SAC and LAC, the lack of detailed information about their hyperparameter tuning process leaves room for further investigation. Future work should explore additional hyperparameters and their potential influence on LAC’s performance and stability to form a more comprehensive understanding of the algorithm’s behaviour. Our research also shows that training time significantly affects the performance and stability of both LAC and SAC. While we adhered to the training time specified by Han et al.[9], extending it in future studies could provide deeper insights into the algorithms’ long-term stability and performance. Although we made modifications to account for missing details in the original methodology, these adjustments—though rigorously validated—may introduce variability that could affect the exact replicability of our results. Nonetheless, the clear relationships observed between the α_3 parameter and algorithm stability give us confidence in the overall validity of our findings, even if certain outcomes might be influenced by the design choices made during the reproduction process.

The most significant limitation for future research building upon our findings is the absence of experiments addressing control stability, robustness, and generalizability under model uncertainties and

external disturbances, as explored in Han et al.[9]’s original work. These experiments are crucial to ensure that the LAC algorithm not only replicates performance and convergence but also demonstrates the stability and robustness benefits emphasized in Han et al.[9]’s study. Robustness is a key factor for the real-world deployment of reinforcement learning algorithms in dynamic and unpredictable environments, particularly in robotics. The omission of these experiments limits the generalizability of our results and underscores a critical area for future research. Other areas for future exploration include potential improvements to the LAC algorithm itself. For instance, incorporating a second critic, as developed during our research but not included in this study (see Appendix C), could mitigate underestimation bias and further improve stability. Furthermore, the LAC algorithm proposed by Han et al.[9] uses a quadratic Lyapunov function that guarantees AS. However, exploring less conservative Lyapunov constraints [22] or those that ensure ES [28] could further enhance the algorithm’s performance without compromising safety. Such modifications could unlock even greater potential for LAC in complex, real-world tasks that demand both precision and robustness.

In conclusion, this study successfully reproduced the core findings of Han et al.[9], confirming that the LAC algorithm offers significant stability improvements over SAC, particularly in environments prone to instability. Our results underscore the importance of precise hyperparameter tuning, particularly the α_3 parameter, which plays a crucial role in balancing stability and performance. Although some discrepancies arose due to differences in environment configurations and hyperparameters, the overall trends aligned with the original study. Our *Reproduction Study* provides a solid foundation for future research, offering a detailed assessment of the LAC algorithm’s performance across various environments and hyperparameter settings. Extending these findings to more complex environments and real-world applications—such as the proposed *Panda Effort Control Experiment*, a promising opportunity for validating LAC’s robustness in dynamic conditions—could yield deeper insights into the algorithm’s stability, robustness, and performance. With these advancements, stable and robust reinforcement learning algorithms can become a practical reality for real-world systems, including robotics and other safety-critical domains.

6

Conclusion

This research sought to evaluate the stability guarantees of learning-based controllers in robotics manipulators through the Lyapunov Actor-Critic (LAC) algorithm, with a focus on effort-controlled environments. The study was motivated by the increasing demand for reliable and robust robotic systems capable of operating effectively in dynamic and unpredictable settings. The primary objective was to assess the reproducibility of Han et al.[9]’s results in new simulated environments, with a particular emphasis on comparing the performance and stability of LAC and SAC, providing a foundation for future research in real-world applications.

The research question guiding this study was: How reproducible are the results of Han et al.[9] in new simulated environments, and what role do key parameters (such as the α_3 parameter) play in the performance and stability of the LAC algorithm compared to SAC? Through a comprehensive *Reproduction Study*, this research successfully validated the findings of Han et al.[9], confirming the reproducibility of the LAC algorithm’s stability guarantees across various simulated environments. The study revealed that the α_3 parameter plays a critical role in enhancing the stability and performance of the LAC algorithm, particularly in complex environments and tasks where maintaining high stability is essential for successful operation.

The implications of these findings are significant for the fields of robotic control and reinforcement learning. By addressing discrepancies in Han et al.[9]’s original study and validating the LAC algorithm’s stability across different simulated environments, this research provides a strong foundation for extending the algorithm’s application to more complex tasks, such as effort-controlled environments, and ensuring the safe deployment of learning-based controllers in real-world settings where stability and reliability are critical. Additionally, the development of a robust codebase and detailed guidelines for key hyperparameters, such as α_3 , facilitates reproducibility and further empirical studies, making this work a valuable resource for future experimentation and practical application.

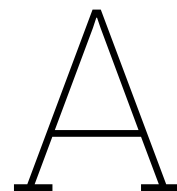
Looking ahead, extending these findings to more complex environments and real-world applications—such as the proposed *Panda Effort Control Experiment*—represents a promising opportunity to further validate the LAC algorithm’s robustness under dynamic conditions. Future experiments could yield deeper insights into the algorithm’s performance, stability, robustness, and applicability, further advancing its potential for deployment in real-world robotic systems.

In conclusion, this thesis has made a significant contribution to the validation of stability guarantees in learning-based robotic controllers, providing critical insights that will support future developments in the field. The validated methodology and findings from this research are expected to drive further exploration and innovation, ultimately advancing the deployment of stable and reliable robotic systems in real-world environments.

References

- [1] Andrea Bacciotti. *Stability and Control of Linear Systems*. Springer, 2019.
- [2] Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. *A Hitchhiker’s Guide to Statistical Comparisons of Reinforcement Learning Algorithms*. Aug. 29, 2022. arXiv: 1904.06979 [cs, stat]. URL: <http://arxiv.org/abs/1904.06979> (visited on 03/02/2024). Pre-published.
- [3] Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. “How Many Random Seeds? Statistical Power Analysis in Deep Reinforcement Learning Experiments”. In: *hal-01890154f* (July 5, 2018). DOI: 10.48550/arXiv.1806.08295. arXiv: 1806.08295 [cs, stat]. URL: <http://arxiv.org/abs/1806.08295> (visited on 08/18/2024).
- [4] Gene F. Franklin et al. *Feedback Control of Dynamic Systems*. Vol. 4. Prentice hall Upper Saddle River, 2002. URL: https://scsolutions.com/wp-content/uploads/TableofContents_FPE8e.pdf (visited on 09/02/2024).
- [5] Scott Fujimoto, Herke van Hoof, and David Meger. *Addressing Function Approximation Error in Actor-Critic Methods*. Oct. 22, 2018. DOI: 10.48550/arXiv.1802.09477. arXiv: 1802.09477 [cs, stat]. URL: <http://arxiv.org/abs/1802.09477> (visited on 08/30/2024). Pre-published.
- [6] Peter Giesl and Sigurdur Hafstein. “Review on Computational Methods for Lyapunov Functions”. In: *Discrete and Continuous Dynamical Systems - B* 20.8 (2015), p. 2291. DOI: 10.3934/dcdsb.2015.20.2291. URL: <https://www.aims sciences.org/article/doi/10.3934/dcdsb.2015.20.2291> (visited on 06/11/2022).
- [7] Tuomas Haarnoja et al. *Soft Actor-Critic Algorithms and Applications*. Jan. 29, 2019. DOI: 10.48550/arXiv.1812.05905. arXiv: 1812.05905 [cs, stat]. URL: <http://arxiv.org/abs/1812.05905> (visited on 07/05/2023). Pre-published.
- [8] Wassim M. Haddad and VijaySekhar Chellaboina. “Nonlinear Dynamical Systems and Control”. In: *Nonlinear Dynamical Systems and Control*. Princeton university press, 2011.
- [9] Minghao Han et al. “Actor-Critic Reinforcement Learning for Control With Stability Guarantee”. In: *IEEE Robotics and Automation Letters* 5.4 (Oct. 2020), pp. 6217–6224. ISSN: 2377-3766. DOI: 10.1109/LRA.2020.3011351.
- [10] Jiang Hua et al. “Learning for a Robot: Deep Reinforcement Learning, Imitation Learning, Transfer Learning”. In: *Sensors* 21.4 (4 Jan. 2021), p. 1278. ISSN: 1424-8220. DOI: 10.3390/s21041278. URL: <https://www.mdpi.com/1424-8220/21/4/1278> (visited on 09/09/2024).
- [11] Ahmed Hussein et al. “Imitation Learning: A Survey of Learning Methods”. In: *ACM Computing Surveys* 50.2 (Apr. 6, 2017), 21:1–21:35. ISSN: 0360-0300. DOI: 10.1145/3054912. URL: <http://doi.org/10.1145/3054912> (visited on 10/24/2022).
- [12] Hassan K. Khalil. *Nonlinear Control*. Vol. 406. Pearson New York, 2015.
- [13] Hassan K. Khalil. *Nonlinear Systems*. Prentice Hall, 2002. 750 pp. ISBN: 978-0-13-067389-3. Google Books: [t_d1QgAACAAJ](https://books.google.com/books?id=t_d1QgAACAAJ).
- [14] Vijay Konda and John Tsitsiklis. “Actor-Critic Algorithms”. In: *Advances in Neural Information Processing Systems*. Vol. 12. MIT Press, 1999. URL: <https://proceedings.neurips.cc/paper/1999/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html> (visited on 08/24/2024).
- [15] Richard Liaw et al. *Tune: A Research Platform for Distributed Model Selection and Training*. July 13, 2018. DOI: 10.48550/arXiv.1807.05118. arXiv: 1807.05118 [cs, stat]. URL: <http://arxiv.org/abs/1807.05118> (visited on 01/09/2024). Pre-published.
- [16] Richard Liaw et al. *Tune: A Research Platform for Distributed Model Selection and Training*. arXiv.org. July 13, 2018. URL: <https://arxiv.org/abs/1807.05118v1> (visited on 09/07/2024).

- [17] Timothy P. Lillicrap et al. *Continuous Control with Deep Reinforcement Learning*. July 5, 2019. DOI: [10.48550/arXiv.1509.02971](https://doi.org/10.48550/arXiv.1509.02971). arXiv: [1509.02971](https://arxiv.org/abs/1509.02971) [cs, stat]. URL: <http://arxiv.org/abs/1509.02971> (visited on 08/30/2024). Pre-published.
- [18] *List of Mathematical Symbols by Subject*. In: *Wikipedia*. June 3, 2022. URL: https://en.wikipedia.org/w/index.php?title=List_of_mathematical_symbols_by_subject&oldid=1091247735 (visited on 06/20/2022).
- [19] Volodymyr Mnih et al. *Asynchronous Methods for Deep Reinforcement Learning*. June 16, 2016. DOI: [10.48550/arXiv.1602.01783](https://doi.org/10.48550/arXiv.1602.01783). arXiv: [1602.01783](https://arxiv.org/abs/1602.01783) [cs]. URL: <http://arxiv.org/abs/1602.01783> (visited on 08/30/2024). Pre-published.
- [20] Hadi Ravanbakhsh and Sriram Sankaranarayanan. "Learning Control Lyapunov Functions from Counterexamples and Demonstrations". In: *Autonomous Robots* 43.2 (Feb. 1, 2019), pp. 275–307. ISSN: 1573-7527. DOI: [10.1007/s10514-018-9791-9](https://doi.org/10.1007/s10514-018-9791-9). URL: <https://doi.org/10.1007/s10514-018-9791-9> (visited on 09/11/2022).
- [21] Harish Ravichandar et al. "Recent Advances in Robot Learning from Demonstration". In: *Annual review of control, robotics, and autonomous systems* 3 (2020), pp. 297–330.
- [22] Staa Rick. *Stability Guarantees in Variable Impedance Control for Rigid Robotics Manipulators in Contact with (Semi)- Rigid Environments*. Apr. 29, 2023. URL: <https://github.com/rickstaa/thesis-stable-variable-impedance-learning/blob/main/report.pdf> (visited on 08/04/2024).
- [23] Ashish Kumar Shakya, Gopinatha Pillai, and Sohom Chakrabarty. "Reinforcement Learning Algorithms: A Brief Survey". In: *Expert Systems with Applications* 231 (Nov. 2023), p. 120495. ISSN: 09574174. DOI: [10.1016/j.eswa.2023.120495](https://doi.org/10.1016/j.eswa.2023.120495). URL: <https://linkinghub.elsevier.com/retrieve/pii/S0957417423009971> (visited on 08/27/2024).
- [24] Abdel-Nasser Sharkawy. "A Survey on Applications of Human-Robot Interaction". In: 251.4 (2021), p. 9.
- [25] Markku Suomalainen, Yiannis Karayiannidis, and Ville Kyrki. "A Survey of Robot Manipulation in Contact". Dec. 3, 2021. arXiv: [2112.01942](https://arxiv.org/abs/2112.01942) [cs]. URL: <http://arxiv.org/abs/2112.01942> (visited on 04/22/2022).
- [26] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. First. A Bradford Book. Cambridge, MA: MIT Press, 1998. URL: <http://incompleteideas.net/book/the-book-1st.html> (visited on 08/24/2024).
- [27] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second edition. Adaptive Computation and Machine Learning Series. Cambridge, Massachusetts: The MIT Press, 2018. 526 pp. ISBN: 978-0-262-03924-6.
- [28] Chengwei Wu et al. "Deep Reinforcement Learning Control Approach to Mitigating Actuator Attacks". In: *Automatica* 152 (June 1, 2023), p. 110999. ISSN: 0005-1098. DOI: [10.1016/j.automatica.2023.110999](https://doi.org/10.1016/j.automatica.2023.110999). URL: <https://www.sciencedirect.com/science/article/pii/S0005109823001528> (visited on 05/05/2023).
- [29] Angeliki Zacharaki et al. "Safety Bounds in Human Robot Interaction: A Survey". In: *Safety Science* 127 (July 1, 2020), p. 104667. ISSN: 0925-7535. DOI: [10.1016/j.ssci.2020.104667](https://doi.org/10.1016/j.ssci.2020.104667). URL: <https://www.sciencedirect.com/science/article/pii/S0925753520300643> (visited on 07/22/2022).



Additional Reproduction Study Remarks

In this appendix, we delve further into the nuanced aspects of our *Reproduction Study* of Han et al.[9]’s work, complementing the technical and methodological details already outlined in chapter 3. We aim to present comprehensive insights and observations that emerged during the *Reproduction Study*, with a special emphasis on detailing the validation process of our translated codebase, rigorously exploring discrepancies between Han et al.[9]’s original codebase and their published paper, and highlighting the enhancements we implemented to improve algorithmic numerical stability.

Structured in several key sections, the appendix first details the validation process applied to our translated algorithms and environments, ensuring fidelity to Han et al.[9]’s original work despite transitioning to a different programming framework. Following this, we provide comprehensive documentation of discrepancies between the original codebase and the published paper, including a detailed account of the inconsistencies encountered and the methodologies employed to resolve these issues. The narrative concludes with a discussion on the enhancements introduced to the algorithmic framework to improve numerical stability, a critical factor for ensuring the training stability and convergence of the algorithms.

By offering these detailed observations, analytical insights, and descriptions of enhancements, our goal is to enhance the transparency of our reproduction efforts and directly assist future researchers focused on this particular area of study. This appendix serves as a comprehensive resource designed to provide a clear, in-depth view of the methodologies we applied, the challenges we faced across all aspects of the *Reproduction Study*, and the proactive steps we took to ensure our findings’ accuracy, stability and reliability. We hope this extensive documentation will be invaluable for those aiming to replicate or extend the work of our study and that of Han et al.[9], facilitating a more straightforward path to understanding and validation in the stable reinforcement learning domain.

A.1. Code Translation Validation Process

A.1.1. Environments Validation

As we delve into the validation process of our translated environments, we must outline the specific challenges we encountered and the methodologies employed to ensure an accurate reproduction of Han et al.[9]’s original work. Our validation approach unfolded in two pivotal phases. Initially, we conducted unit tests to compare our translated environments against the foundational sources on which Han et al.[9] had based their adaptations. These sources specifically included environments within the Gymnasium, Gymnasium-robotics, and Bullet3 frameworks. This step ensured that our translations maintained consistency with these foundational environments’ expected behaviour and intentions. Subsequently, we meticulously compared our translated versions with Han et al.[9]’s actual implementations. This phase aimed to verify that the specific modifications detailed in Han et al.[9]’s paper¹ were accurately

¹For an in-depth understanding of each modification, including the rationale and implementation details, refer to Han et al.[9]’s original paper [9] and our stable-gym package documentation at <https://rickstaa.dev/stable-gym/envs/envs.html>.

reflected in our work, addressing any discrepancies between the methodologies described in their article and the actual implementation in their codebase. This comprehensive two-phase process was designed to navigate the significant challenges posed by version discrepancies between our modern codebase and the older versions used by Han et al.[9], which relied on outdated libraries. These discrepancies were especially pronounced in seeding methods critical for deterministic behaviour, complicating direct comparisons due to the non-guarantee of identical outcomes from identical seeds in both codebases. To manage these complexities effectively, we tailored our validation process to account for the intricacies introduced by evolving library versions.

A.1.1.1. Comparison with Foundational Environment Sources

The initial step of our validation process was to conduct unit tests that compared our translated environments with the foundational sources essential to Han et al.[9]’s adaptations. These foundational sources include but are not limited to, environments such as CartPole, GRN, and FetchReach, which were central to our research. This phase was pivotal in ensuring that despite Han et al.[9]’s modifications, our translations accurately reflected the expected behaviours of the original environments. We included these unit tests in our continuous integration pipeline to integrate this validation step seamlessly into our workflow. This systematic approach automates the verification process, ensuring that any modifications to our codebase preserve the fidelity of our translations. For the sake of transparency and to aid in the validation of our study, we have made these unit tests available, along with instructions for their execution, at <https://github.com/rickstaa/stable-gym/tree/main/tests>.

A.1.1.2. Analysis of Han et al.’s Adaptations

In the subsequent validation stage, we conducted an exhaustive comparison between the original versions crafted by Han et al.[9] and our environment translations. Our focus was to ascertain the accuracy of our translations in replicating the specific adaptations and modifications Han et al.[9] described in their study. This comparison highlighted several discrepancies between the implementations in Han et al.[9]’s codebase and the descriptions provided in their paper. These discrepancies were carefully reconciled to ensure alignment with the published work. A detailed discussion of these discrepancies and the applied resolutions is presented in Section A.2.1.

After addressing these discrepancies, we tackled the issue of variability in computational behaviour due to different versions of the foundational libraries. We devised Python scripts to analyse the ‘step’ function’s output in the CartPole, (Comp)GRN, and FetchReach environments. We achieved high consistency across library versions through careful manual synchronisation of the initial states, data types for action and observation spaces, and the inputs to the ‘step’ method. This rigorous validation process, applied over 10 subsequent steps in each environment, confirmed the dynamics of the CartPole and GRN environments to a precision of seven significant digits. For the FetchReach environment, we attained a precision of two significant digits, reflecting minor dynamic variations due to differences in Mujoco versions. These results affirm that our environment translations faithfully replicate the dynamics of Han et al.[9]’s environments, aside from slight variations likely due to algorithmic and numerical precision updates. For the sake of transparency and to facilitate replication, the validation scripts, their results, and an in-depth explanation of our methodology are accessible at: <https://github.com/rickstaa/stable-gym/tree/han2020/tests/manual/validation>.

A.1.2. Algorithms Validation

The primary aim of our *Reproduction Study* was to replicate the findings of Han et al.[9] to ensure a solid basis for the novel *Panda Effort Control experiment*. Successfully achieving this replication is an implicit validation of our algorithm translations, affirming their accuracy for subsequent experimental work. Nevertheless, to ensure the precision of our implementations throughout the translation process, we engaged in intermediate validation steps. These steps were instrumental in ensuring that our adaptations accurately embodied the original algorithms from Han et al.[9]’s paper, setting the stage for the experiments conducted in the *Reproduction Study*.

Our initial validation step involved adopting the PyTorch SAC algorithm from the `spinningup`² library, selected for its proven robustness and comprehensive testing. We thoroughly compared this base implementation against Han et al.[9]’s original TF1 codebase and their paper’s algorithmic descriptions. This process enabled us to precisely incorporate features specific to Han et al.[9]’s version of SAC, such

²For more information on the `spinningup` library, visit <https://spinningup.openai.com> (accessed 21-01-2024).

as learning rate decay and adaptive temperature tuning, enhancing the `spinningup` SAC algorithm to align with their study.

Afterwards, we conducted comparative training runs of our translated SAC algorithm against the original codebase to assess the accuracy of our implementation, focusing on the `CartPole` and `Oscillator` environments. Critical metrics such as test performance, test episode length, learning rate behaviours, and the evolution of the α and α_{loss} parameters during training were rigorously compared. Achieving consistency in these performance metrics with Han et al.[9]’s implementation confirmed the effectiveness of our SAC adaptation.

Building upon the solid foundation established with our SAC algorithm implementation, we focused on implementing the LAC algorithm. This endeavour involved augmenting the SAC framework with specific modifications characteristic of the LAC methodology, as delineated in Han et al.[9]’s study. Key among these modifications, as elaborated in Section 2.2.4, was enhancing the critic network to encompass the Lyapunov stability criterion and embedding a Lyapunov-based constraint within the policy optimization framework.

To ensure the integrity of our LAC adaptation, we conducted a comprehensive comparative validation process mirroring the approach taken with SAC. This process particularly emphasized critical metrics such as the average Lyapunov error (I_{error}), λ , and λ_{loss} . The consistency observed in the performance metrics and parameter trajectories between our implementation and Han et al.[9]’s original work validated our adaptation and underscored its fidelity and reliability.

These intermediate validation steps demonstrated the rigour of our translation efforts and reinforced confidence in the *Reproduction Study*. They established a robust basis for the replication objectives, enabling the *Panda Effort Control experiment* to proceed with confidence.

A.2. Code Inconsistencies and Discrepancies

Following our examination of the code translation validation process, this section delves into the specific discrepancies and inconsistencies uncovered between Han et al.[9]’s documented methods and their actual codebase. Our objective is to thoroughly document these variations, highlighting the challenges they introduced to our *Reproduction Study* and the strategies employed to resolve them. Initially focusing on environmental inconsistencies before discussing algorithm implementation variances, we aim to bolster the transparency and integrity of our study. This detailed account enhances the study’s clarity, supporting future researchers in navigating and building upon our work more effectively.

A.2.1. Environment Inconsistencies and Discrepancies

The translation and validation of Han et al.[9]’s study environments unveiled a series of inconsistencies not detailed in the original publication. These include but are not limited to, the randomization of initial states and inconsistencies in the definition of cost functions. Such variances necessitated careful adjustments to ensure our experimental setup faithfully represented Han et al.[9]’s intentions. This subsection catalogues these specific environment-related findings, shedding light on the adjustments we implemented. Documenting these nuances is crucial, as it allows for a more accurate comparison of our reproduction efforts with the original study.

A.2.1.1. CartPole Environment

In translating and validating the `CartPole` environment from Han et al.[9]’s study, we identified several inconsistencies within their original codebase that deviated from the publication’s descriptions. These findings necessitated targeted adjustments to our `CartPole` implementation, ensuring it accurately reflected the experimental design as detailed in the paper.

Randomization of Initial State

During our detailed examination, it became evident that Han et al.[9]’s implementation incorporated randomization of the `CartPole` environment’s initial state. Although the concept of randomization was mentioned in their paper, it needed more specificity regarding the range used. Our analysis of their codebase revealed the ranges to be $[-5, 5]$ for the cart position and $[-0.2, 0.2]$ for the cart velocity, pole angle, and pole angular velocity. These parameters are crucial, as they are selected to ensure various initial conditions within the observation space’s centre, thereby providing diversity without significantly altering the task’s difficulty or nature. To maintain the integrity of our experimental comparison, we adopted these ranges in our translated environment.

Cost Function

Another significant discovery was a discrepancy between the cost function described by Han et al.[9] in their paper and its implementation in the publicly available code. The paper delineates the cost function as:

$$c = \left(\frac{x}{x_{threshold}} \right)^2 + 20 \cdot \left(\frac{\theta}{\theta_{threshold}} \right)^2 \quad (\text{A.1})$$

In contrast, the code implementation calculates the cost as:

$$c = \frac{x^2}{100} + 20 \cdot \left(\frac{\theta}{\theta_{threshold}} \right)^2 \quad (\text{A.2})$$

This variation does not impact the study's outcomes when $x_{threshold}$ is configured to 10, rendering the two formulas effectively equivalent for our purposes. However, this distinction becomes crucial for researchers seeking to adjust the $x_{threshold}$ parameter in their reproduction of Han et al.[9]'s work. Committed to closely mirroring the original study's described methodology, we updated the cost function in both our and Han et al.[9]'s environment code³ to reflect the correction.

A.2.1.2. GRN Environments

As we extended our examination to the GRN and CompGRN environments from Han et al.[9]'s study, we encountered inconsistencies between the paper's descriptions and the actual codebase. These variances and certain under-explained aspects in the publication prompted a closer investigation. This section documents our findings and the subsequent adjustments to align with Han et al.[9]'s intended experimental configurations.

Action Space

While Han et al.[9]'s paper described the action space utilized in their study, it omitted specific details about the range and scaling of actions within this space. Our thorough review of their codebase revealed that actions were constrained to $u \in [0, 1]$. More critically, we discovered that Han et al.[9] implemented a scaling factor 5.0 to actions post-clipping, a nuance not explicitly mentioned in their documentation. This oversight necessitated an adjustment to the mathematical model outlined in formula 15 of Han et al.[9]'s Appendix S2 to incorporate this scaling factor accurately:

$$\begin{aligned} x_1(t+1) &= x_1(t) + dt \cdot \left[-\gamma_1 x_1(t) + \frac{a_1}{K_1 + x_2^2(t)} + b_1 u_1 \right] + \xi_1(t), \\ x_2(t+1) &= x_2(t) + dt \cdot \left[-\gamma_2 x_2(t) + \frac{a_2}{K_2 + x_2^2(t)} + b_2 u_2 \right] + \xi_2(t), \\ x_3(t+1) &= x_3(t) + dt \cdot \left[-\gamma_3 x_3(t) + \frac{a_3}{K_3 + x_2^2(t)} + b_3 u_3 \right] + \xi_3(t), \end{aligned}$$

In this updated mathematical model, we introduced input gains b_i and set them to $b_i = 5.0$ for all $i \in \{1, 2, 3\}$, ensuring our representation aligns with the actual implementation in Han et al.[9]'s codebase.

Observation Space

During our examination of the GRN environments, we noted that while Han et al.[9]'s publication described the components of the observation space, it did not specify the range for these parameters. The codebase indicated a default range of $[-1, 1]$, which was inconsistent with the experimental results depicted in their paper. To reconcile these discrepancies and more accurately reflect the intended experimental setup of Han et al.[9], we adjusted our translation of the observation space. Specifically, we defined the observation space in our implementation to encompass $[0.0, \infty]$ for system states and the reference signal and $[-\infty, \infty]$ for the reference error. This modification ensures that our environment's observation space comprehensively represents the full spectrum of states observable in the GRN environments, consistent with the experimental conditions of the original study.

³See the corrected version of Han et al.[9]'s environment code at <https://github.com/rickstaa/Actor-critic-with-stability-guarantee/tree/rstaa2024>.

Episode Termination Condition Clarification

Upon further investigation of the GRN and CompGRN environments, we uncovered discrepancies in episode termination criteria within Han et al.[9]’s study that were not disclosed. The GRN environment’s code includes a termination condition where episodes end if the cost exceeds 100, a specification absent from both the paper and the CompGRN environment’s code. This omission is significant for accurately replicating the study’s findings, as the presence and absence of this condition in the GRN and CompGRN environments directly influence the experimental outcomes, subtly affecting the algorithm’s convergence by impacting the cost and episode terminations. Our replication efforts confirmed that adhering to the codebase’s specific conditions, terminating GRN environment episodes based on cost and not applying similar criteria to the CompGRN environment are necessary for accurately reproducing the results presented in Han et al.[9]’s publication. We, therefore, adjusted our translations to align with these original experimental configurations.

Episode Length

An additional aspect not explicitly detailed in Han et al.[9]’s publication is the episode length for the GRN and CompGRN environments. Although not directly mentioned, examining the figures and experimental setups described in their paper implies an episode length of 400 steps. This hypothesis was confirmed by the value found within their codebase. As a result, we adopted this episode length to align our *Reproduction Study* as closely as possible with Han et al.[9]’s original experimental conditions.

A.2.1.3. FetchReach Environment

Our examination of the FetchReach environment in Han et al.[9]’s study revealed a notable inconsistency in the cost function’s formulation. Typically, reinforcement learning algorithms, such as the Soft Actor-Critic (SAC), aim to maximise rewards, which, within the FetchReach environment, equates to minimising the negative Euclidean distance between the robot’s end effector and the target. Although this reward structure, employed by the OpenAI Gym’s FetchReach environment, aligns with the common objective of reward maximisation in reinforcement learning algorithms like the SAC algorithm, it presents a significant challenge for the LAC algorithm. The LAC algorithm depends on a positive definite cost function for its stability and performance, a critical aspect detailed in Section 2.2.4. The use of a negatively defined cost function in Han et al.[9]’s codebase inadvertently incentivises the LAC algorithm to increase distance, countering its optimisation objectives and undermining stability. Given that the results in their paper depict a positive and decreasing cost, it suggests this issue may stem from the diverse use of their codebase across different studies. To address this, we modified the cost function in our implementation to be positive definite, ensuring it complies with the LAC algorithm’s stability requirements and supports consistent minimisation behaviour for both SAC and LAC algorithms.

A.2.2. Algorithm Implementation Discrepancies

While replicating the study by Han et al.[9], we identified significant discrepancies between the algorithmic parameters they reported and the actual configurations in their codebase. Addressing these discrepancies was crucial to ensuring that our *Reproduction Study* faithfully mirrored the experimental conditions and outcomes they documented. This precise alignment was indispensable for upholding the integrity of our reproduction efforts and was essential for laying a robust foundation the subsequent *Panda Effort Control Experiment*.

A.2.2.1. Hyperparameter Values

One initial discrepancy identified was the reported versus implemented value of the hyperparameter α_3 . Han et al.[9]’s publication specified an α_3 value of 1.0, but their codebase implemented a value of 0.1. Recognising the significant potential impact of this discrepancy on algorithmic performance, stability, and robustness, our *Reproduction Study* included a hyperparameter optimisation to determine the most effective α_3 value. Our analysis revealed that α_3 values between 0.1 and 0.3 aligned more closely with the original study’s reported outcomes, particularly regarding performance, convergence, and the average value of λ during training. Consequently, to ensure a thorough examination throughout our *Reproduction Study*, we evaluated all α_3 values between 0.1 and 1.0, examining their impacts on the algorithm’s performance and convergence speed in comparison to both the SAC algorithm and the original study’s results. This approach enabled a comprehensive investigation into the role of the α_3 parameter in the reproducibility and integrity of the findings. For an in-depth look at the methodology and outcomes of this hyperparameter optimisation, refer to Sections 3.1.3 and 4.1.1.

Further analysis of Han et al.[9]’s codebase revealed discrepancies in hyperparameters and training durations. Specifically, their code for the finite-horizon version of the LAC algorithm implemented a horizon length of 2, contrary to the reported value of 5. Although Han et al.[9] claimed that variations in finite horizon length do not significantly impact convergence and performance, our experiments showed that a horizon length of 5 is required to match their reported performance and ensure λ convergence (see Section E.1.1.1.5). Using a horizon of 2 resulted in worse performance and incomplete λ convergence. Additionally, we observed a discrepancy in the number of training steps for the CompGRN environment. Additionally, Han et al.[9] did not explicitly state the training durations for each environment, so we inferred these values from Figures 1 and S10 in their paper. For CompGRN, the λ convergence graph suggested a more extended training period of $2e5$ steps, which differed from the $1e5$ steps implemented in the codebase. To align with the results reported in Han et al.[9]’s paper, particularly the λ convergence shown in the graph, we replicated the longer training duration in our study. As can be seen from Figure 4.2 in Section 4.1.1, this adjustment was necessary for achieving consistent λ convergence and matching the performance reported in the original study.

A.2.2.2. Learning Rate Configurations

Another critical discrepancy addressed involved the learning rate applied to the optimization process of the Lagrange multiplier (λ), which is associated with the Lyapunov stability constraint. While Han et al.[9] recommended a learning rate of $3e-4$, the actual codebase utilized a rate of $1e-4$, aligning it with the actor’s learning rate. To accurately reflect the experimental setup reported by Han et al.[9], we adjusted the learning rate for λ in our implementation to match the documented rate. Validation tests in the CartPole, CompGRN, GRN, and FetchReach environments confirmed that this adjustment did not significantly affect the algorithm’s performance, underscoring the robustness of our replication process. The results of this test can be found in Section E.1.1.1.3 of Appendix E. Our examination of the code also revealed that the learning rates for the actor, critic, and minimum entropy constraint (α) were subject to linear decay during training, whereas the λ learning rate remained constant. Since the original article did not mention any specific decay schedules, we adhered to the decay schedules observed in the code for the actor, critic, α , and λ . Maintaining a constant learning rate for λ makes sense, as it is crucial for enforcing the stability constraint throughout the training process. A decaying rate could risk inadequate enforcement of the stability constraint and undermine the Lyapunov stability guarantees central to the LAC algorithm.

A.2.2.3. Network Architecture Configurations

Upon further evaluation, we also uncovered architectural discrepancies in the network configurations for the LAC algorithm, as reported by Han et al.[9], versus their code implementation. The publication described a $[256, 256]$ actor network architecture for both the SAC and LAC algorithms, while the LAC codebase implemented a more modest $[64, 64]$ setup, diverging from the documented SAC configuration. Recognising the critical role of network architecture in determining algorithm performance, we aligned our study with the publication’s specifications, adopting a $[256, 256]$ architecture for both SAC and LAC in our implementation. Validation tests in all key environments utilised in our *Reproduction Study*, aimed at assessing performance impact and convergence rates, affirmed that this architectural adjustment had a negligible effect on LAC’s performance, detailed in Section E.1.1.1.4 of Appendix E.

Additionally, discrepancies were identified in the critic network architectures within the CompGRN environment, where the publication specified a $[256, 256, 16]$ configuration, in contrast to the $[64, 64, 16]$ setup found in the codebase. Adhering to the documented $[256, 256, 16]$ architecture, we conducted validation tests to ensure this adjustment did not compromise our *Reproduction Study*. The results, indicating no adverse impact on performance and convergence, are detailed in Section E.1.1.1.6 of Appendix E. Moreover, as stated in Chapter 3, we identified a general architectural discrepancy between the SAC and LAC critic networks in Han et al.[9], where the SAC critic followed a $[256, 256]$ structure, while the LAC critic included an extra output layer. To address this, we conducted an experiment in our main study, modifying the SAC critic to match the LAC architecture. The results, evaluating the impact of this difference on performance and convergence, are presented in Section 4.1.3 of Chapter 4.

A.3. Enhancements for Algorithmic Numerical Stability

In our *Reproduction Study*, we introduced modifications to the algorithmic parameters to bolster numerical stability, a crucial aspect that underpins the reliability of simulation results. Specifically, we

adjusted the initial value of the discount factor, λ , from the value of 1.0 used by Han et al.[9] to 0.99. This slight reduction is a commonly used technique to prevent potential numerical instabilities that may arise at the extremes of parameter ranges, such as with λ . This adjustment was applied consistently across both Han et al.[9]’s codebase and our own, providing a controlled basis for comparison. After thorough analysis, we confirmed that adjusting λ to 0.99 did not significantly impact the study’s results, underscoring the reliability of our findings.

Furthermore, we refined our approach to the entropy regularization parameter, α , in the loss function. Han et al.[9]’s original implementation, based on the Softlearning package, used $\log \alpha$ in its loss function and optimized α in the log space. However, since the completion of their work, the package has been updated to directly utilize the α parameter in its loss function while still optimizing α in the log space—a change aimed at enhancing numerical stability⁴. Motivated by this advancement and to mitigate the numerical issues that direct optimization of α can introduce, particularly in steep or ill-conditioned optimization landscapes, we incorporated the $\log \alpha$ loss function into our *Reproduction Study*.

The same procedure was applied to the λ parameter to further improve numerical stability. This approach contrasts with the λ optimization formula depicted in Han et al.[9]’s paper. However, upon further investigation into Han et al.[9]’s codebase, we found that they also used $\log \lambda$ optimization in practice. These decisions were made to ensure that our methods reflect current best practices and contribute to the robustness and reliability of the results. Through meticulous analysis, we determined that switching to $\log \alpha$ and $\log \lambda$ optimizations did not materially alter the algorithm’s convergence rates or performance outcomes compared to the original method. This finding confirms that our methodological refinements, though aimed at improving numerical stability, maintain the study’s overall integrity and contribute to the robustness of our reproduction effort.

⁴For further details on this development, please refer to the discussion at <https://github.com/rail-berkeley/softlearning/issues/136> (last accessed 21-01-2024).

B

Panda Effort Control Experiment

This appendix outlines the methodology for the proposed *Panda Effort Control Experiment*, designed to evaluate the performance of the LAC and SAC algorithms in a complex, simulated environment. The primary objectives of this experiment are to assess the algorithms' convergence, performance, stability, robustness, and generalization capabilities. This experiment builds on findings from the *Reproduction Study* detailed in chapter 3, and aims to explore these algorithms in a more challenging scenario: controlling a 7-DOF effort-controlled Panda Emika Franka robot in an environment lacking disturbance rejection mechanisms and characterized by non-linear dynamics. Structured in several key sections, this appendix details the proposed experimental setup, the hyperparameter tuning process, and the training and evaluation methodologies. An overview of the codebase created for performing this experiment is provided in Appendix C. Our aim with this appendix is to establish a solid foundation for future research and exploration in this domain.

B.1. Experimental Setup

As outlined in Appendix A of the *Reproduction Study*, several modifications were made to Han et al.[9]'s original study to resolve inconsistencies and ensure a fair comparison between LAC and SAC. These adjustments enabled the LAC agent to successfully learn, validating our implementation. In addition to these changes, the following modifications will be made to the original LAC implementation to further align it with SAC and ensure a fair comparison:

- **Network Architecture:** Both SAC and LAC critic networks will use a [256, 256] architecture, diverging from Han et al.[9]'s use of [64, 64] for LAC. To further ensure a fair comparison, we will include an additional output layer in the SAC critic network, consistent with the LAC architecture. This adjustment aligns with the additional experiment discussed in Section 3.1.5 of Chapter 3.
- **Initial α Value:** Han et al.[9] used an initial value of 2.0 for the α parameter in LAC, compared to 1.0 in SAC, resulting in a higher learning rate for the LAC entropy. For consistency and fairness, we will use an initial α value of 1.0 for both SAC and LAC.
- **Update Frequency:** In the original study, LAC used an 80/100 update frequency and SAC used a 50/100 update frequency, meaning the number of environment steps and policy updates differed between the two. We will unify this by using a 1/1 ratio for both SAC and LAC, where one policy update occurs after each environment step.
- **Learning Rate Decay:** A per-epoch learning rate decay will be applied to stabilize learning, as opposed to the step-based decay used in the original study.
- **Horizon Type:** Han et al.[9] utilized an infinite horizon variant of SAC, while the LAC algorithm was implemented as a finite horizon variant. To ensure a fair comparison, we will evaluate both SAC and LAC in their respective horizon types: SAC infinite horizon with LAC infinite horizon, and SAC finite horizon with LAC finite horizon.

The experiment can be conducted using a simulated Panda Emika Franka robot with effort-controlled joints, a 7-DOF robotic manipulator. This simulation environment is provided by the manufacturer,

Franka Emika, and is accessible as a ROS package¹. Since the original Panda simulation provided by Franka Emika is not directly suited for training reinforcement learning algorithms, a Gymnasium wrapper package, called `ros_gazebo_gym`², can be utilized. This package wraps the original Panda simulation, making it compatible with the Gymnasium package, commonly used to create reinforcement learning environments. The complete codebase and a detailed description of the implementation are provided in Appendix C. The Panda environment within the `ros_gazebo_gym` package includes several reinforcement learning tasks for training, such as *PandaReach*, *PandaTrack*, *PandaPush*, *PandaSlide*, and *PandaPickAndPlace*. For this experiment, the focus will be on the *PandaReach* and *PandaTrack* tasks, due to their relevance to our stability-focused research [22]. These tasks involve minimal interaction with the environment, which is a critical requirement for ensuring that the stability constraints derived for the LAC algorithm are consistently upheld [22]. The following subsections provide a detailed description of these task environments.

B.1.1. PandaReach Task

In the **PandaReach** task, the robot maneuvers its end-effector to a specific target position. The goal is to minimize the distance between the end-effector and the target, measured by the following cost function:

$$c = -\sqrt{(x_{ee} - x_{goal})^2 + (y_{ee} - y_{goal})^2 + (z_{ee} - z_{goal})^2} \quad (\text{B.1})$$

To simplify the task, three of the seven joints can be fixed, limiting the robot's movement to a single plane sagittal to the robot's base. The remaining four joints will be effort-controlled, allowing the robot to apply torque to the joints to reach the target position. The task concludes when the perpendicular distance between the end-effector and the target falls below or above a specified threshold (i.e., 0.1 meters or 0.5 meters, respectively) or when the maximum number of timesteps (500) is reached.

B.1.1.1. PandaTrack Task

The **tracking task** environment elevates the challenge of the reach task by introducing a time-varying trajectory that the robot must follow. The target y and z positions will vary sinusoidally within the robot's plane of motion, resembling a figure-8 pattern. Specifically, the target position is defined parametrically by the following equations:

$$\begin{aligned} y_{goal}(t) &= A \cdot \sin(\omega t) \\ z_{goal}(t) &= B \cdot \sin(2\omega t) \end{aligned}$$

where A and B are the amplitudes of the motion, ω is the angular frequency, and t is time. The robot's objective is to maintain minimal average deviation from this trajectory, with the following cost function:

$$c = -\frac{1}{T} \sum_{t=0}^T \sqrt{(x_{ee} - x_{goal})^2 + (y_{ee} - y_{goal})^2 + (z_{ee} - z_{goal})^2} \quad (\text{B.2})$$

Unlike the reach task, the tracking task will not end when the perpendicular distance between the end-effector and the target falls below a specified threshold. Instead, the episode concludes when the maximum number of timesteps (1000) is reached or when the error between the end-effector and the target position exceeds 0.1 meters. Apart from these changes, the tracking task environment is identical to the reach task environment.

B.2. Hyperparameter Tuning

To ensure optimal performance of the LAC and SAC algorithms in the *Panda Effort Control Experiment*, extensive hyperparameter tuning will be conducted using the Ray Tune package[16]. Ray Tune is a scalable hyperparameter optimization library that supports efficient, parallel trials and adaptive scheduling, making it ideal for the computationally intensive tuning required in the Panda environment. This

¹The Franka Emika Panda ROS package, provided by the manufacturer for the Panda Emika Franka robot simulation, is available at https://github.com/frankaemika/franka_ros (last accessed 21-01-2024).

²The `ros_gazebo_gym` package which is available at <https://github.com/rickstaa/ros-gazebo-gym>.

Hyperparameter	Range (Step size)
α_3	[0.1, 1.0] (step size = 0.1)
Replay Buffer Size	$[10^3, 10^6]$ (log scale)
Minibatch size	[32, 512] (step size = 32)
Actor Learning Rate	$[10^{-6}, 10^{-3}]$ (log scale)
Critic Learning Rate	$[10^{-6}, 10^{-3}]$ (log scale)
Temperature Learning Rate	$[10^{-6}, 10^{-3}]$ (log scale)
Lyapunov learning rate	$[10^{-6}, 10^{-3}]$ (log scale)
Soft replacement (τ)	[0.005, 0.01] (step size = 0.001)
Discount (γ)	[0.9, 0.999] (step size = 0.01)
Critic structure (f_ϕ)	{64, 128, 256, 512}
Actor structure (f_θ)	{64, 128, 256, 512}

Table B.1: The hyperparameters tuned in the *Panda Effort Control Experiment*.

approach is more suitable than simpler methods, such as the GridSearch technique used in the *Reproduction Study*. The initial hyperparameters for the tuning process will be based on those used in the FetchReach environment from Han et al.[9], as detailed in Table E.1, since this environment closely resembles the Panda environment in terms of degrees of freedom. While we acknowledge that the control scheme between the FetchReach and Panda environments differs (position control versus effort control), FetchReach offers the closest dynamic similarities to our new environment. Table B.1 lists the hyperparameters to be tuned, along with their respective ranges. The final hyperparameter values will be presented in the final report, accompanied by the rationale for each selection.

B.3. Algorithm Training

After completing the hyperparameter tuning process, the LAC and SAC algorithms can be trained in the *Panda Effort Control Experiment* using the tuned hyperparameters. It is recommended to train each algorithm for 500 episodes on the *PandaReach* task and 1000 episodes on the *PandaTrack* task. Each episode in the *PandaReach* task should last a maximum of 500 timesteps, while episodes in the *PandaTrack* task can last up to 1000 timesteps. These episode counts were based on preliminary experiments indicating convergence within this range, though researchers may adjust based on hyperparameter tuning results or signs of early or late convergence.

To monitor progress, performance evaluations should be conducted every 10 episodes by running 10 additional episodes in the environment with the trained policy. The mean cost across these episodes can be calculated, as defined in the cost functions for the *PandaReach* and *PandaTrack* tasks (see Sections B.1.1 and B.1.1.1). Logging and tracking of training performance can be managed through Weights and Biases (WandB) integration, which is included in the `stable_learning_control` package. WandB allows storage of training outputs, such as parameters used, console logs, trained models, and test results, but it is optional and can be replaced with other logging methods. The full codebase is available on GitHub³ and is explained in Appendix C.

To ensure reproducibility, it is highly recommended that training be conducted within the provided Docker container, which guarantees a consistent environment across different machines. This container can be built using the Dockerfile in Appendix C and is available on Docker Hub⁴. Although training multiple seeds in parallel is possible, each training run should be conducted on a single GPU for consistency. If control frequency instabilities occur, the simulator should be paused between timesteps to maintain a fixed (real-time) control frequency and prevent noise during training.

B.4. Convergence and Performance Evaluation

After training the LAC and SAC algorithms in the *Panda Effort Control Experiment*, the performance and convergence of the algorithms will be evaluated across multiple dimensions. These evaluations are essential for understanding the algorithms' efficiency, stability, robustness, and generalizability. Throughout the experiment, the performance of both LAC and SAC will be directly compared to assess

³The full codebase for algorithm training is available at <https://github.com/rickstaa/stable-learning-control> (last accessed 21-01-2024).

⁴The Docker container can be found at <https://hub.docker.com/r/rickstaa/stable-learning-control> (last accessed 21-01-2024).

their relative strengths and weaknesses in the tasks.

B.4.1. Performance Evaluation

The primary metric for performance evaluation is the *final average test performance*. This metric reflects the cost obtained during evaluation episodes and serves as a reliable indicator of the algorithm's precision in both the *PandaReach* and *PandaTrack* tasks.

- **Final Average Test Performance:** This metric is calculated by averaging the cost over 10 evaluation episodes conducted at the end of the training process. Lower costs indicate better performance, reflecting the robot's ability to either reach the target (in the *PandaReach* task) or accurately follow the trajectory (in the *PandaTrack* task).

For a detailed methodology on how this metric is calculated, refer to the *Reproduction Study* (Section 3.1.4 in 3). Ensuring consistency with these prior experiments guarantees that performance is measured in a reliable and comparable way.

B.4.2. Convergence Evaluation

In addition to assessing final performance, the convergence of the LAC and SAC algorithms will be critically evaluated. Convergence measures how efficiently and consistently the algorithms reach stable performance during training. The key aspects of convergence include:

- **Test Performance Convergence Speed:** This metric tracks how quickly the algorithm's cost decreases and stabilizes over time. The convergence speed is measured by the number of episodes it takes for the cost to consistently fall below a predefined threshold (set at 95% of total convergence). This threshold is based on the difference between the maximum observed cost and the final average test performance, ensuring that it reflects meaningful stabilization.
- **λ Convergence:** This metric focuses on how efficiently the λ value stabilizes during training, ensuring compliance with stability constraints. A rapid convergence of the λ value toward zero indicates that the algorithm successfully learns to maintain system stability.

Both *test performance convergence speed* and λ *convergence* will be compared between LAC and SAC to evaluate which algorithm achieves stable performance more quickly and consistently. For detailed definitions of these convergence metrics, refer to Section 3.1.4 of the *Reproduction Study*.

B.5. Stability Analysis

After evaluating convergence, the next critical step is to assess stability in both the *PandaReach* and *PandaTrack* tasks. Stability is crucial in real-world robotics, where algorithms must maintain consistent control over time. Stability will be evaluated by examining the following metrics in static and dynamic tasks:

B.5.1. Reach Task Stability

In the *PandaReach* task, stability will be measured by the algorithm's ability to maintain the end-effector at a specified reference point after reaching the target. Once the end-effector reaches the target, it will be required to hold this position for 100 additional timesteps. The key metric for evaluating stability in this task is:

- **Variance in End-Effector Position:** Stability is quantified by calculating the variance in the end-effector's position over the 100-timestep period. Lower variance indicates better stability, reflecting the algorithm's ability to maintain a steady position.

A comparison of LAC and SAC based on the variance in the end-effector's position will reveal which algorithm exhibits superior stability in static control scenarios.

B.5.2. Tracking Task Stability

For the *PandaTrack* task, stability will be evaluated based on the algorithm's ability to follow a time-varying trajectory, such as a figure-8 pattern. The critical stability metric for this dynamic task is:

- **Variance in Trajectory Tracking:** Stability in the tracking task will be assessed by measuring the variance in tracking accuracy over time. This metric reflects how consistently the algorithm can follow the prescribed trajectory. Lower variance indicates better performance in maintaining accurate tracking.

Comparing the LAC and SAC algorithms based on the variance in trajectory tracking will help determine which algorithm is better suited for dynamic tasks requiring adaptation to changing environments.

Both of these stability metrics—variance in the end-effector’s position and variance in trajectory tracking—will provide valuable insights into the long-term reliability of LAC and SAC in maintaining stable control.

B.6. Robustness and Generalization Experiments

The final phase of the experiment involves testing the robustness and generalization capabilities of LAC and SAC under various conditions. These evaluations are critical for understanding how well the algorithms perform in dynamic, unpredictable environments.

B.6.1. Robustness Evaluation

Robustness will be tested by introducing random disturbances into the *PandaReach* and *PandaTrack* tasks. The goal is to assess the algorithms’ ability to maintain consistent performance when subjected to external disruptions. Each algorithm will be tested 10 times with added random noise to the effort-controlled joints, simulating realistic disturbances that could affect performance. Robustness will be evaluated based on:

- **Performance Under Disturbance:** The variance in performance (as measured by the cost function) across all trials will be computed and averaged. Lower performance variance indicates higher robustness, reflecting the algorithm’s ability to handle unpredictable disturbances while maintaining control.

Comparing the robustness of LAC and SAC under these conditions will reveal which algorithm better withstands real-world uncertainties.

B.6.2. Generalization Evaluation

To assess generalization, the algorithms will be tested on previously unseen scenarios. Generalization evaluates how well the algorithms adapt to novel environments. The algorithms will be tested on new trajectories such as a flipped figure-8 and a circular path, which differ from the training tasks. These variations challenge the algorithms’ ability to adapt their learned policies. Generalizability will be evaluated based on:

- **Generalization Performance:** The algorithms’ performance in these new tasks will be measured using the same cost metrics as in the original tasks. Lower costs indicate better generalization, as they reflect the algorithm’s ability to adapt quickly and effectively to new conditions.

A comparison of LAC and SAC in these generalization tasks will provide insights into their adaptability and suitability for real-world deployment.

B.6.3. Conclusion on Robustness and Generalization

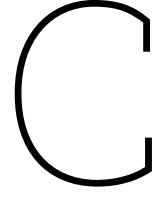
Evaluating robustness and generalization is essential for understanding how LAC and SAC perform in dynamic, unpredictable environments. These metrics offer deeper insights into each algorithm’s ability to handle disturbances and adapt to new, unforeseen scenarios, which are critical for practical robotic systems.

The proposed *Panda Effort Control Experiment* provides a comprehensive framework for evaluating LAC and SAC in a complex, effort-controlled robotic environment. Several factors contribute to the complexity of the Panda environment:

- **Higher Degrees of Freedom (DOF):** The Panda Emika Franka robot, with 7 degrees of freedom, presents a significantly more intricate control challenge compared to simpler systems like CartPole or Fetch Reach.

- **Effort Control Complexity:** The focus on effort control, where the robot's joints are managed by controlling the applied torque, introduces additional complexity. The robot must manage its stability in real-time while responding to dynamic forces and torques.
- **Realistic Simulation in Gazebo:** The Gazebo simulator introduces realistic physics and environmental interactions, including friction and inertia. This realism increases the challenge for the control algorithms, which must perform reliably under conditions that closely mimic real-world scenarios.
- **Need for Robustness and Generalization:** Given these complexities, the algorithms must not only perform well in controlled environments but also demonstrate robustness and generalization to unseen situations. This is essential for practical deployment, where robots may face unpredictable challenges.

The complexity of the 7-DOF robot and the realistic simulation environment underscores the necessity for robust and generalizable control algorithms. The outcomes of the *Panda Effort Control Experiment* will provide valuable insights into the capabilities of LAC and SAC, particularly their ability to handle real-world robotic tasks in complex, dynamic environments.



Codebase Structure and Overview

This chapter outlines our extensive codebase, which, along with the open-source contributions described in the following chapter (Chapter D), constitutes the backbone of this thesis's research. This codebase primarily utilizes Python and the Robot Operating System (ROS) with critical components selectively developed in C++ to enhance performance and compatibility. Hosted on GitHub, it comprises a suite of modules that, while independently crafted, are interconnected to enhance overall functionality. This modular approach bolsters our results' reproducibility and encourages further exploration and research within this domain. Here, we provide an overview focusing on the architecture and functionality of each module, alongside their specific roles and interconnectedness in achieving our research goals. For detailed installation and usage guidelines, please refer to the comprehensive documentation of each component's GitHub repository.

C.1. Codebase Overview

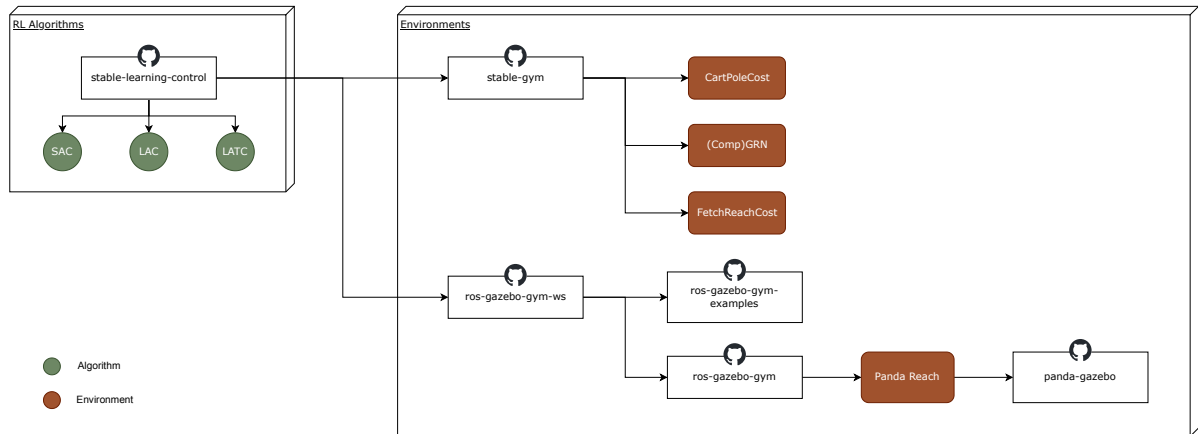


Figure C.1: A diagrammatic representation of the codebase, emphasizing the interconnection between six distinct modules and their respective roles within the research.

Our codebase consists of six uniquely functional yet interrelated modules, forming a cohesive system that underpins our research objectives. Figure C.1 visually details this interconnected structure, elucidating the role and synergy among the modules:

- **Stable Learning Control** - This versatile framework facilitates training and evaluation of various stable reinforcement learning control algorithms, focusing on theoretical stability and robustness.
- **Stable Gym** - Offers Gymnasium environments with specifically designed cost functions, tailored to stable reinforcement learning agents.

- **ROS Gazebo Gym** - A bridge between ROS, Gazebo, and Gymnasium, providing diverse robotic simulation environments, including the crucial Panda gym environment for this research.
- **ROS Gazebo Gym Examples** - Demonstrates the use of Gymnasium environments within the `ros_gazebo_gym` package through practical examples.
- **ROS Gazebo Gym Workspace** - A Catkin workspace easing the setup and usage of the `ros_gazebo_gym` framework.
- **Panda Gazebo** - An optimized Gazebo simulation for the Panda Emika Franka, tailored to enhance the Panda Gymnasium environment in `ros_gazebo_gym`.

These modules can be categorized into two main groups: the *Stable Learning Control* package, which includes the reinforcement learning algorithms fundamental to our research, and five additional packages that provide simulation environments on which these algorithms are trained. The following sections offer brief overviews of each package, with links to their extensive documentation and GitHub repositories.

C.1.1. Stable Learning Control

The Stable Learning Control package (i.e. `stable_learning_control`), accessible at <https://github.com/rickstaa/stable-learning-control>, is integral to our research. It comprises a suite of stable reinforcement learning algorithms critical for our experiments and the necessary tools for their training and evaluation. Designed for compatibility with any Gymnasium environment, it is especially synergistic with the `stable_gym` and `ros_gazebo_gym` packages, which are also part of this research. The package includes:

- **Soft Actor-Critic (SAC)** - An off-policy algorithm that enhances a stochastic policy with entropy augmentation, enabling balanced exploration and exploitation [7]. Directly utilized in this thesis.
- **Lyapunov Actor-Critic (LAC)** - A robust off-policy reinforcement learning algorithm incorporating Lyapunov stability theory [9], also employed in the research detailed in this thesis.
- **Lyapunov Actor Twin-Critic (LATC)** - An innovative enhancement of LAC, developed by the authors, combining LAC's stability with the efficiency of a dual-critic design. This algorithm is not used in the current research but is intended for inclusion in future studies.

In addition to offering TensorFlow 2 and PyTorch implementations (with our research predominantly using the latter), the package is characterized by its modular architecture that eases the integration of new algorithms. Furthermore, it features a user-friendly Command Line Interface for tuning, training, evaluating, and visualizing algorithms, enhancing its practical utility. Detailed documentation is available at <https://rickstaa.dev/stable-learning-control>.

C.1.2. Stable Gym

The Stable Gym package (i.e. `stable_gym`), available at <https://github.com/rickstaa/stable-gym>, presents a diverse collection of Gymnasium environments, each tailored with positive definite cost functions suitable for stable reinforcement learning agents. The environments included are:

- **Biological Environments**: Such as the Oscillator environment, akin to the one in Han et al.[9].
- **Classic Control Environments**: Featuring modified versions of well-known Gym environments such as CartPole, alongside the inclusion of Ex3EKF as used in our related paper [28].
- **Mujoco Environments**: Adaptations of popular Mujoco environments, including AntCost, HalfCheetah, and Humanoid.
- **Robotics Environments**: Featuring simulations such as FetchReach, QuadXHover, and Mini-taurBullet.

These environments, designed for stable reinforcement learning algorithms that necessitate positive definite cost functions, are highly conducive to research in reinforcement learning emphasizing stability. Initially developed to complement algorithms found in the Stable Learning Control package, they are adaptable to a wide range of reinforcement learning agents. This package incorporates the translated environments from Han et al.[9]'s study, utilized in our *Reproduction Study*, thereby directly

addressing our research question. For more detailed information about the package and its extensive list of environments, please visit <https://rickstaa.dev/stable-gym>.

🔔 Attention!

To accurately replicate the results of our *Reproduction Study*, the use of the `han2020` branch of the `stable_gym` package is essential. Available at <https://github.com/rickstaa/stable-gym/tree/han2020>, this branch is vital as it contains the specific environment configurations used in Han et al.[9]'s original research [9], which is the focus of our study's replication efforts.

C.1.3. ROS Gazebo Gym

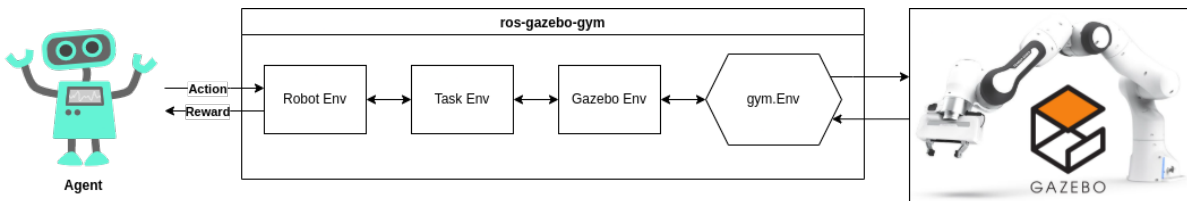


Figure C.2: The structure of ROS Gazebo Gym, illustrating the integration between ROS, Gazebo, and Gymnasium, and its application in reinforcement learning scenarios.

The Ros Gazebo Gym package (i.e. `ros_gazebo_gym`), available at <https://github.com/rickstaa/ros-gazebo-gym>, is a comprehensive framework that plays a crucial role in our codebase, seamlessly integrating ROS, Gazebo, and Gymnasium. As depicted in Figure C.2, this integration is key to developing and training reinforcement learning algorithms in realistic robot simulations, effectively bridging the gap between simulated and real-world robotic applications. This package is particularly notable for including the Panda Gymnasium environments to be used in the *Panda Effort Control Experiment*, illustrating its direct relevance to our research.

Inspired by the `openai_ros`¹ package, the structure of `ros_gazebo_gym` is divided into three main classes, enhancing both readability and the potential for extension:

- **Task Environment:** Manages the complete learning task for the robot. This includes setting up the initial state, defining learning objectives, calculating rewards, and determining episode termination conditions, thus orchestrating the overall learning process.
- **Robot Environment:** Specifies and integrates the robot within the Gazebo simulation, ensuring efficient sensor data acquisition and actuator control. This environment is key to translating algorithmic commands into robotic actions.
- **Gazebo Environment:** Establishes and maintains the connection to the Gazebo simulator. It enables crucial interactions like controller switching, simulation pausing, and simulator resetting, which are essential for coherent simulation control and management.

This framework's compartmentalized structure enables the ROS Gazebo Gym environments to be imported as standard Gymnasium environments. It provides researchers with an efficient method to quickly create Gymnasium wrappers for ROS simulations. A notable feature of this package is its automated system that, upon import, downloads, builds, and installs all necessary ROS packages required to run the included robotics environments. This process significantly streamlines setup, reducing space requirements and enhancing the user experience. Along with its comprehensive documentation, `ros_gazebo_gym` is an invaluable tool for reinforcement learning research, seamlessly bridging the gap between simulation training and practical robotics applications. For a thorough exploration of the package and its diverse applications in robotic simulations, please visit <https://rickstaa.dev/ros-gazebo-gym>.

¹For more information on the `openai_ros` package, visit https://wiki.ros.org/openai_ros (last accessed 21-01-2024).

C.1.4. ROS Gazebo Gym Examples

The ROS Gazebo Gym Examples package (i.e., `ros_gazebo_gym_examples`), available at <https://github.com/rickstaa/ros-gazebo-gym-examples>, offers a comprehensive set of practical demonstrations explicitly designed for the Gymnasium environments provided by the `ros_gazebo_gym` package. These examples are particularly valuable for researchers and developers, as they demonstrate the application of reinforcement learning algorithms within the specialized Gymnasium environments. Notably relevant to our research is the example in which a SAC agent can be trained across the Panda task environments. This provides a salient demonstration of the sophisticated reinforcement learning techniques applied in realistic simulations of the Panda robot, yielding actionable insights into these operational processes. To delve into these examples and to leverage the full potential of the ROS Gazebo Gym package, please visit https://rickstaa.dev/ros-gazebo-gym/get_started/usage.html#usage-examples.²

C.1.5. ROS Gazebo Gym Workspace

The ROS Gazebo Gym Workspace package (i.e., `ros_gazebo_gym_ws`), accessible at <https://github.com/rickstaa/ros-gazebo-gym-ws>, is a meticulously designed Catkin workspace created to facilitate the setup and use of the ROS Gazebo Gym framework. It integrates the submodules of both the `ros_gazebo_gym` and `ros_gazebo_gym_examples` packages, offering a streamlined approach to clone and build the complete framework along with its examples in a single action. This workspace is tailored to expedite researchers' engagement with ROS Gazebo Gym task environments, significantly enhancing the workflow for both the development and experimentation phases. For detailed instructions on utilising this workspace, including practical usage examples, please visit https://rickstaa.dev/ros-gazebo-gym/get_started/usage.html#usage-examples.

C.1.6. Panda Gazebo

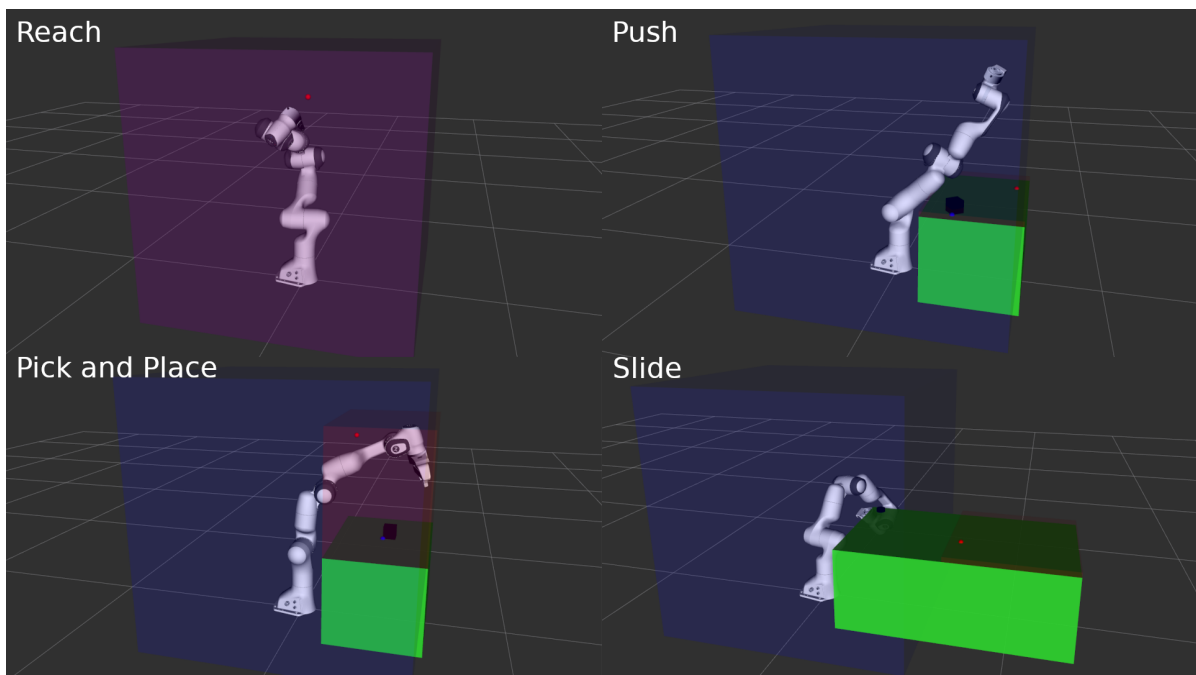


Figure C.3: A selection of simulated environments within the Panda Gazebo package, showcasing diverse scenarios optimized for reinforcement learning tasks involving the Panda Emika Franka robot.

The Panda Gazebo package (i.e., `panda_gazebo`), accessible at <https://github.com/rickstaa/panda-gazebo>, substantially extends Gazebo's simulation capabilities for the Panda Emika Franka robot. It builds upon the `franka_gazebo`² package, enriching it with tools specially designed for reinforcement

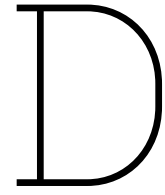
²Part of `franka_ros`, the `franka_gazebo` package can be found at https://github.com/frankaemika/franka_ros (accessed 21-01-2024).

learning research. Essential for the *Panda Effort Control Experiment*, it offers a solid simulation base within the ROS Gazebo Gym framework.

The package's design segregates the robot's simulation elements, like control and sensor data handling, from the `ros_gazebo_gym` package, thus simplifying the Panda robot's integration for reinforcement learning tasks. It includes a suite of launch files, ROS nodes, services, and Gazebo plugins that streamline using the Panda robot simulation with various reinforcement learning algorithms. These launch files configure the Panda robot within various Gazebo environments, each tailored for distinct reinforcement learning challenges, as illustrated in Figure C.3:

- **Panda Reach Task World:** Provides a clear environment for reach task training with the robot.
- **Panda Push Task World:** Positions the Panda robot and a box on a table for push task simulations.
- **Panda Pick and Place World:** Includes a table and cube, creating a scenario for pick-and-place exercises.
- **Panda Slide Task World:** Contains a puck and a designated target area to practice precision sliding tasks.

The package's ROS nodes significantly improve robot control and sensor data handling, offering services for meticulous joint trajectory planning, precise command execution, and accurate end-effector positioning. The included Gazebo plugins, such as the `panda_joint_locker` plugin, add gazebo control features, including joint locking and unlocking, to facilitate more sophisticated simulation scenarios. By providing a comprehensive set of tools and functionalities, the Panda Gazebo package is an indispensable toolkit for researchers engaged in Panda robot simulations within the reinforcement learning domain. Comprehensive documentation detailing the package's full spectrum of tools and functionalities is available at <https://rickstaa.dev/panda-gazebo/>.



Additional Contributions and Byproducts of Research

D.1. Article: Deep reinforcement learning control approach to mitigating actuator attacks

During my studies, I had the privilege of collaborating with my initial supervisor, [Dr. Wei Pan](#), and his research group on several articles. One notable contribution was to the article *Deep Reinforcement Learning Control Approach to Mitigating Actuator Attacks*, which was accepted for publication in *Automatica*, Volume 152, June 2023. My primary role in this project involved the implementation and evaluation of the proposed algorithm, a contribution that earned me the position of third author. The article is accessible [here](#), and the abstract is included below. For my involvement in other articles, I received an honourable mention. In addition, I peer-reviewed several articles for the *IEEE Robotics and Automation Letters* journal. Collaborating with Dr. Wei Pan and his research group was an immensely valuable experience, significantly enhancing my research capabilities and academic proficiency.

Abstract of the Article

Deep Reinforcement Learning Control Approach to Mitigating Actuator Attacks

This paper investigates the deep reinforcement learning based secure control problem for cyber-physical systems (CPS) under false data injection attacks. We describe the CPS under attacks as a [Markov decision process](#) (MDP), based on which the secure [controller design](#) for CPS under attacks is formulated as an action policy learning using data. Rendering the SAC learning algorithm, a Lyapunov-based SAC learning algorithm is proposed to offline train a secure policy for CPS under attacks. Different from the existing results, not only the convergence of the learning algorithm but the stability of the system using the learned policy is proved, which is quite important for security and stability-critical applications. Finally, both a satellite attitude control system and a robot arm system are used to show the effectiveness of the proposed scheme, and comparisons between the proposed learning algorithm and the classical PD controller are also provided to demonstrate the advantages of the control algorithm designed in this paper.

D.2. Open Source Contributions

D.2.1. Franka Emika Panda Robot Gazebo Simulation

In my commitment to open-source collaboration, I significantly contributed to the [franka_ros](#) repository, focusing on enhancements and bug fixes. The absence of a Gazebo simulation in the original [franka_ros](#) package led me to engage with and contribute to Erdal Pekel's [panda_simulation](#). Further, with the beta release of the [franka_gazebo](#) package, I identified and resolved numerous bugs, enhancing its reliability and performance. My contributions are detailed [here](#). This work, essential for

my research, contributed directly to my thesis and significantly benefited the wider robotics community while enhancing my robotic simulation and software development skills.

D.2.2. Panda MoveIt Config

In the absence of a Panda MoveIt configuration package for ROS Noetic, a necessity for my thesis, I undertook the migration of the [panda_moveit_config](#) package to this newer version. Supervised by [Robert Haschke](#), my modifications were successfully integrated into the official [panda_moveit_config](#) repository. This integration not only facilitated my thesis work but also extended the package's utility to the ROS Noetic community at large. The comprehensive details of my contributions are documented [here](#).

D.2.3. Minor Contributions

In addition to the significant contributions mentioned previously, I have also made valuable, though more minor, contributions to a range of repositories in the field:

- [ros/ros_distro](#) - Submitted pull requests for adding dependencies to the rosdep database.
- [ros-planning/moveit](#) - Contributed new features and bug fixes to the MoveIt repository.
- [ros-planning/moveit_tutorials](#) - Addressed small bugs in the MoveIt tutorials through pull requests.
- [openai/spinningup](#) - Fixed minor bugs in the SpinningUp repository.
- [Farama-Foundation/Gymnasium](#) - Opened several pull requests to rectify bugs in the Gymnasium package.
- [PickNikRobotics/moveit_sim_controller](#) - Made a targeted pull request to resolve a bug in the MoveIt Sim Controller repository.
- [catkin/catkin_tools](#) - Submitted a pull request to fix a specific issue in the Catkin Tools repository.

Supplementary Figures and Tables

E.1. Figures

E.1.1. Reproduction Study

E.1.1.1. Additional Alpha3 Tuning Experiments

E.1.1.1.1 Extended Seed Analysis for GRN and CompGRN Environments

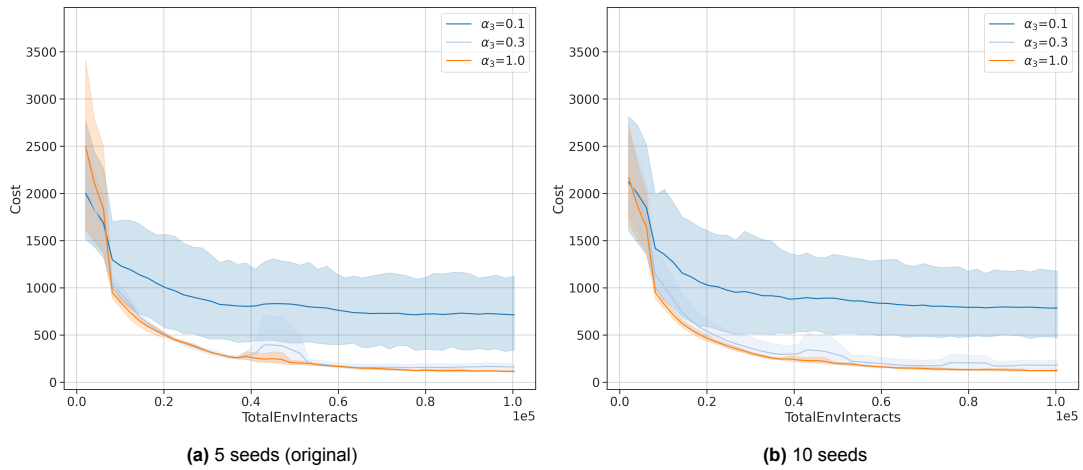


Figure E.1: Average test performance, measured as mean cost, and convergence of the **LAC** algorithm for select α_3 values in the **GRN environment** between *five* and *ten* runs with different seeds. Each line represents the mean cost across these seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals. For clarity, only key α_3 values are shown.

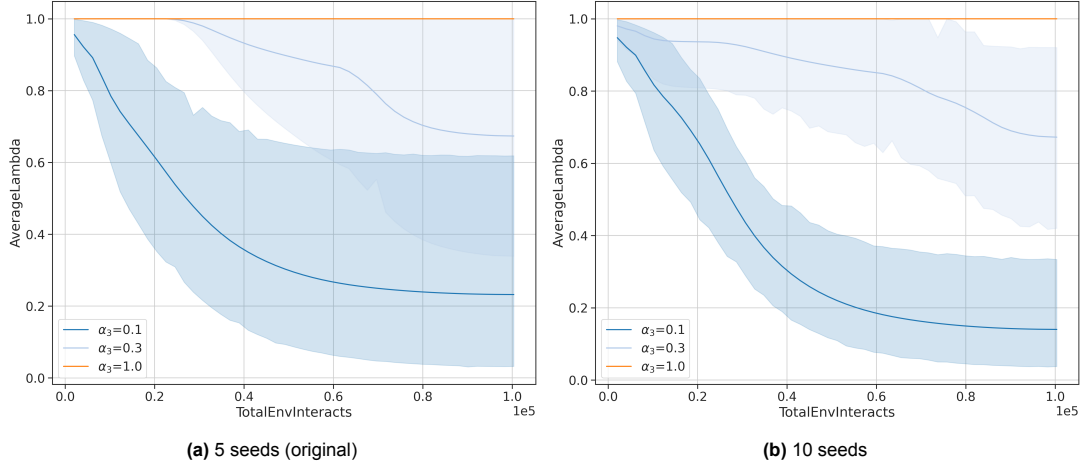


Figure E.2: Convergence of λ values during **LAC** algorithm training in the **GRN environment** between *five* and *ten* runs with different seeds for select α_3 values, illustrating adherence to the stability constraint. Each line represents the mean λ value across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals. For clarity, only key α_3 values are shown.

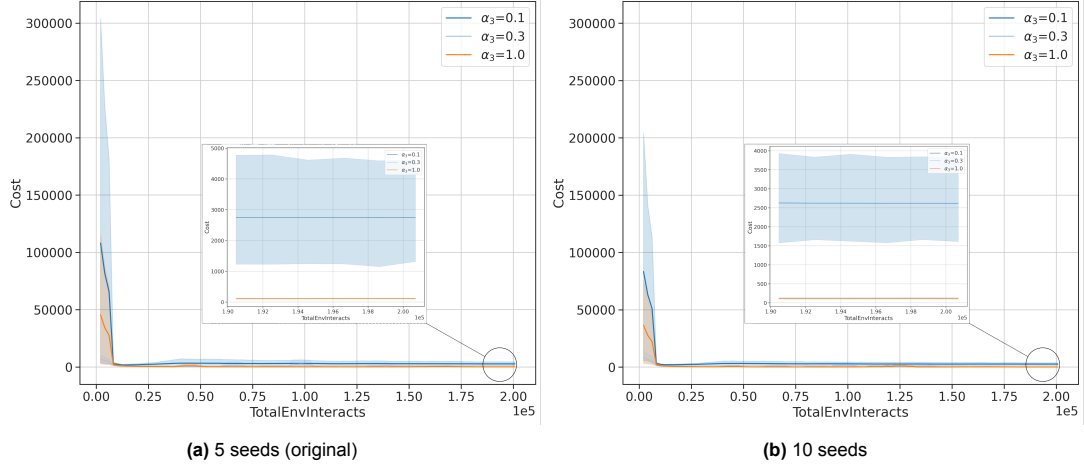


Figure E.3: Average test performance, measured as mean cost, and convergence of the **LAC** algorithm for select α_3 values in the **CompGRN environment** between *five* and *ten* runs with different seeds. Each line represents the mean cost across these seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals. For clarity, only key α_3 values are shown.

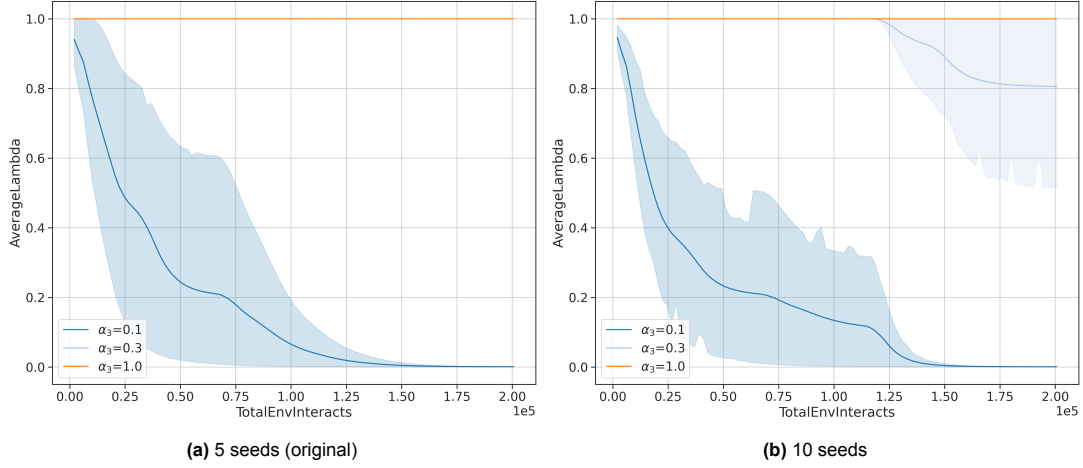


Figure E.4: Convergence of λ values during **LAC** algorithm training in the **CompGRN environment** between *five* and *ten* runs with different seeds for select α_3 values, illustrating adherence to the stability constraint. Each line represents the mean λ value across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

E.1.1.1.2 Impact of Prolonged Training on GRN Environment

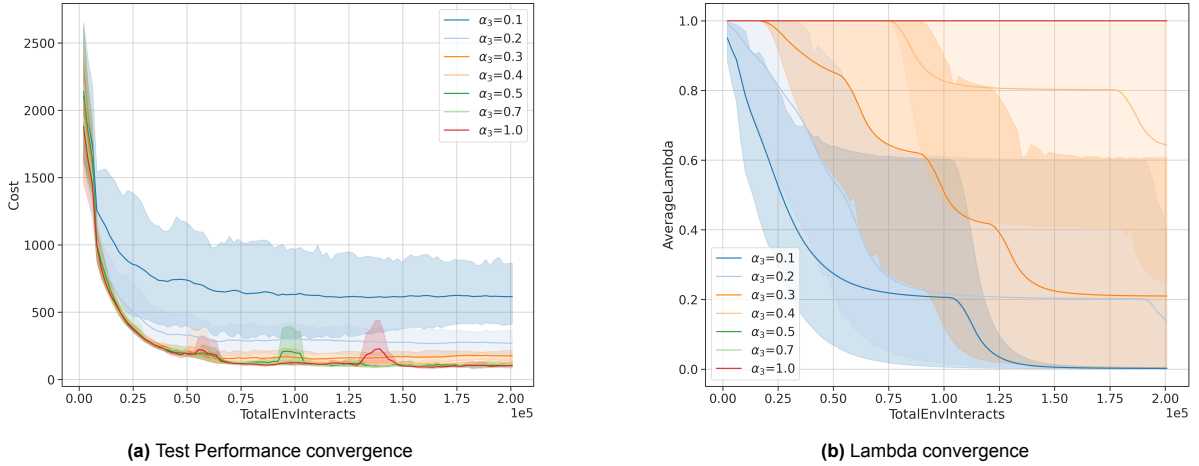


Figure E.5: Average Test Performance, measured as mean cost, and Lambda convergence of the **LAC** algorithm for select α_3 values in the **GRN environment** after *prolonged training*. Each line represents the mean cost or Lambda value across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

E.1.1.1.3 Impact of Lambda Learning Rate

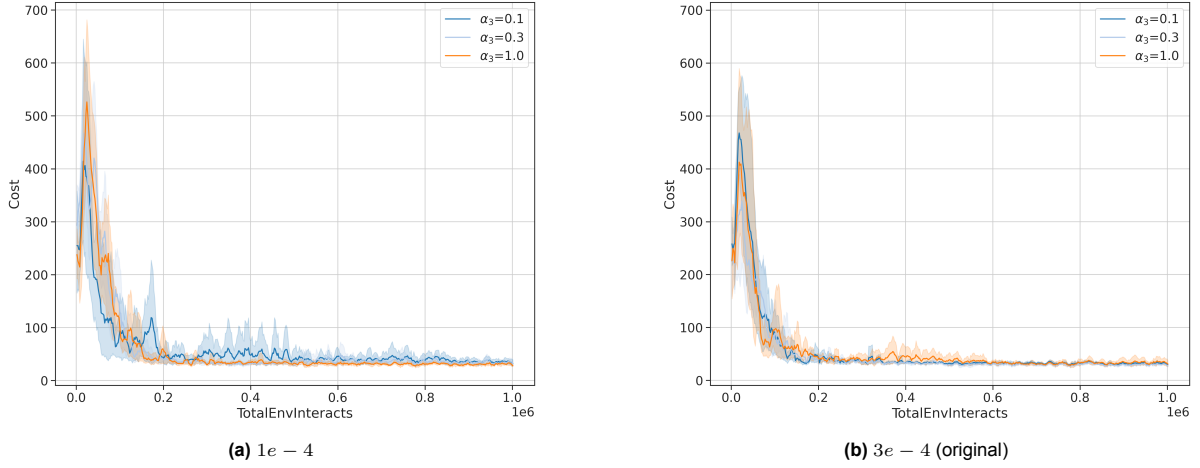


Figure E.6: Average test performance, measured as mean cost, and convergence of the **LAC** algorithm for select α_3 values between *two different λ learning rates* in the **CartPole environment**. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

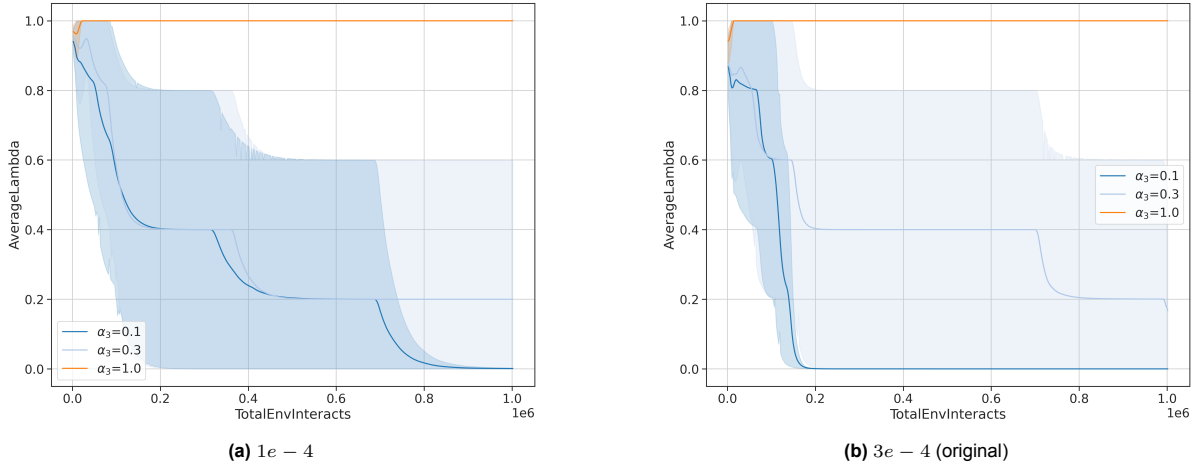


Figure E.7: Convergence of λ values for the **LAC** algorithm for select α_3 values between *two different λ learning rates* in the **CartPole environment**. Each line represents the mean λ across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

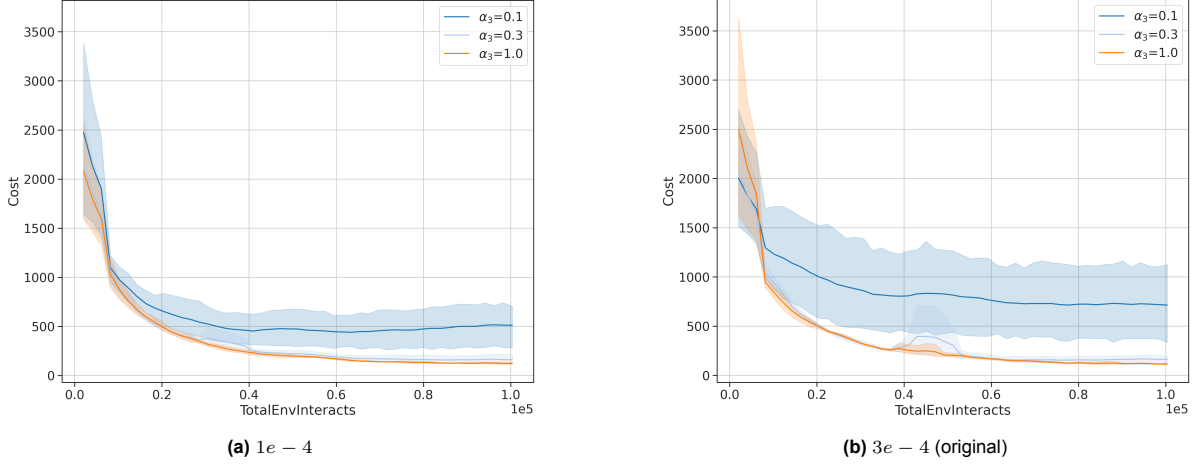


Figure E.8: Average test performance, measured as mean cost, and convergence of the **LAC** algorithm for select α_3 values between *two different λ learning rates* in the **GRN environment**. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

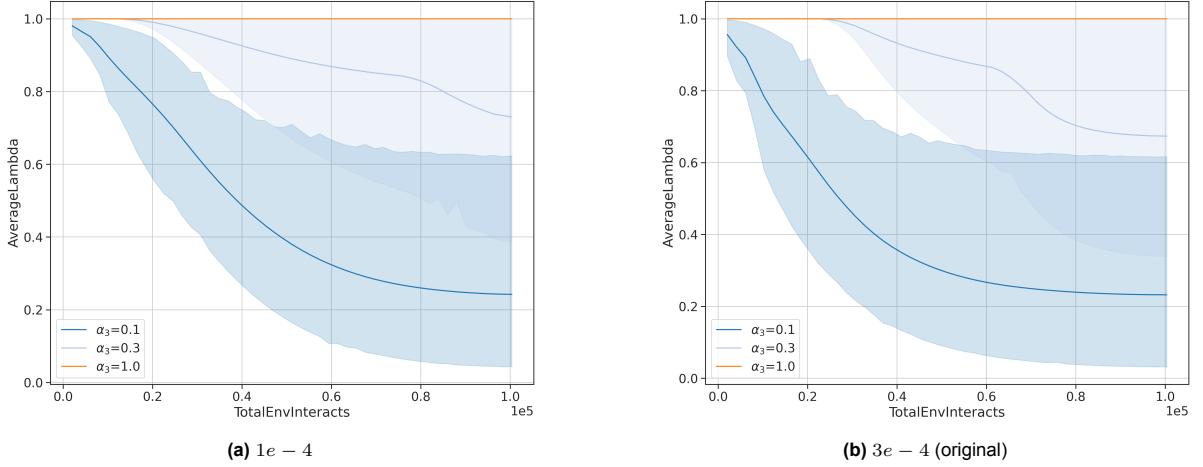


Figure E.9: Convergence of λ values for the **LAC** algorithm for select α_3 values between *two different λ learning rates* in the **GRN environment**. Each line represents the mean λ across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

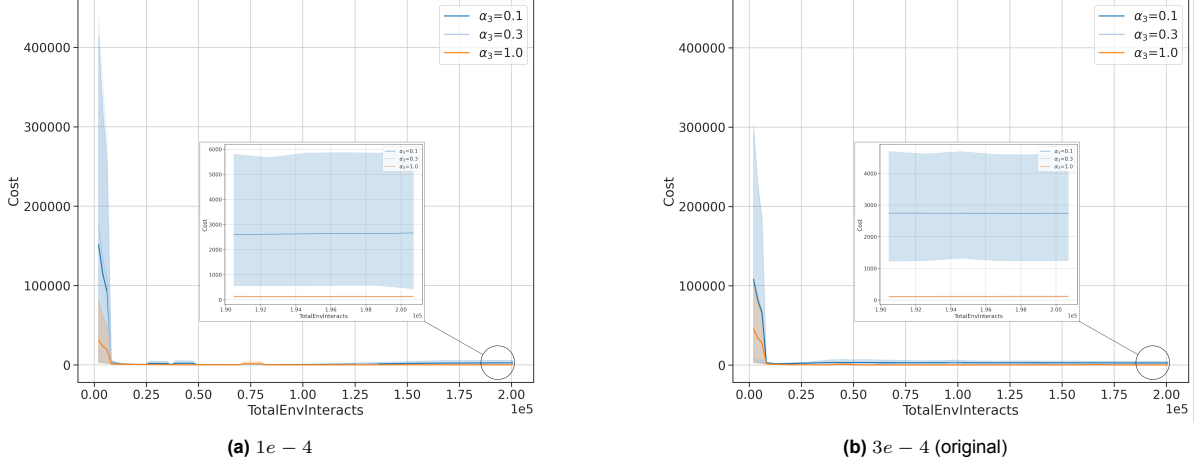


Figure E.10: Average test performance, measured as mean cost, and convergence of the **LAC** algorithm for select α_3 values between two different λ learning rates in the **CompGRN** environment. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

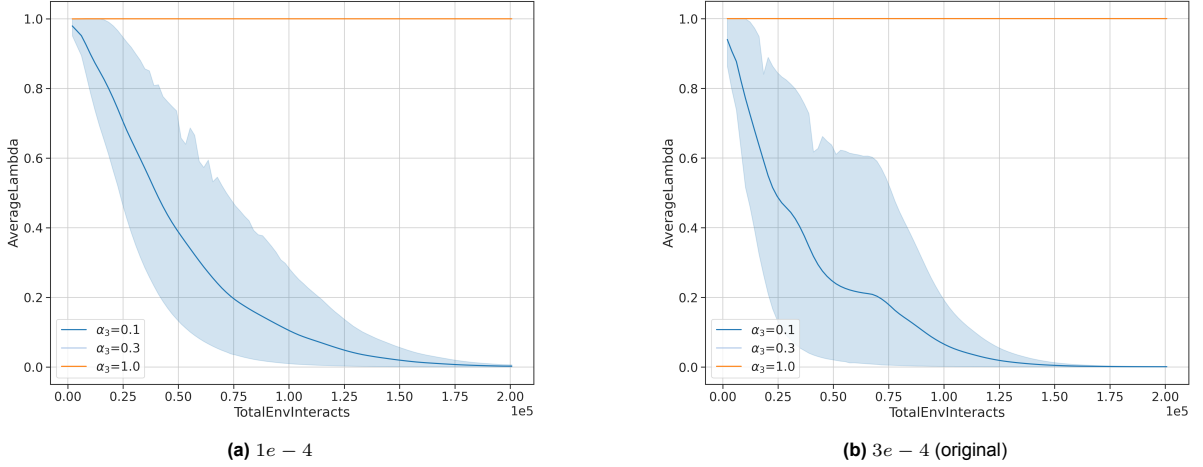


Figure E.11: Convergence of λ values for the **LAC** algorithm for select α_3 values between two different λ learning rates in the **CompGRN** environment. Each line represents the mean λ across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

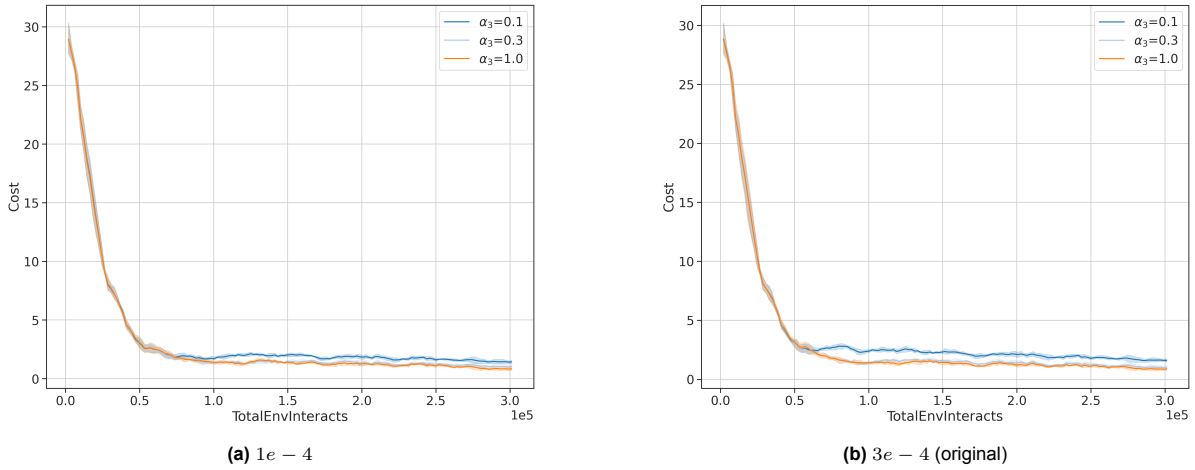


Figure E.12: Average test performance, measured as mean cost, and convergence of the **LAC** algorithm for select α_3 values between *two different λ learning rates* in the **FetchReach** environment. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

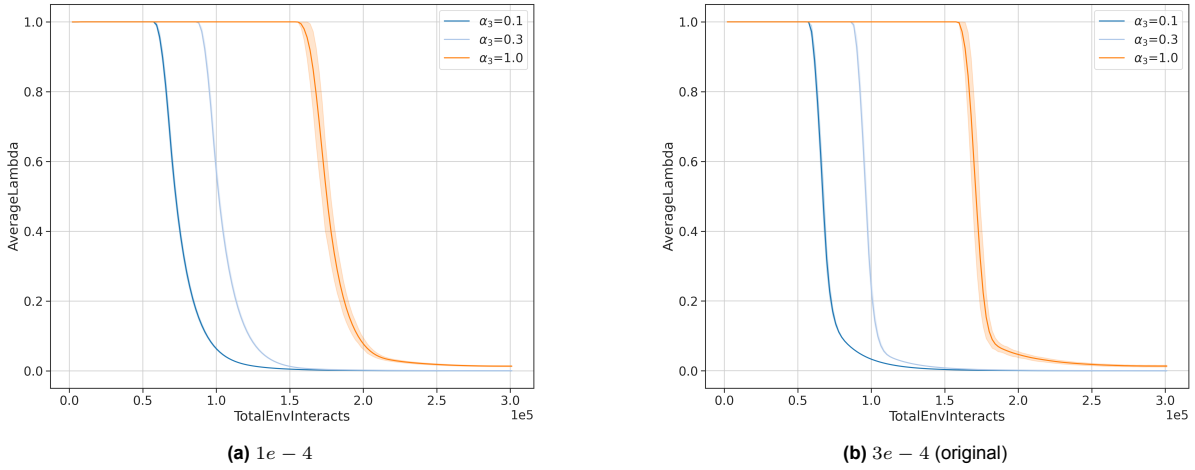


Figure E.13: Convergence of λ values for the **LAC** algorithm for select α_3 values between *two different λ learning rates* in the **FetchReach** environment. Each line represents the mean λ across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

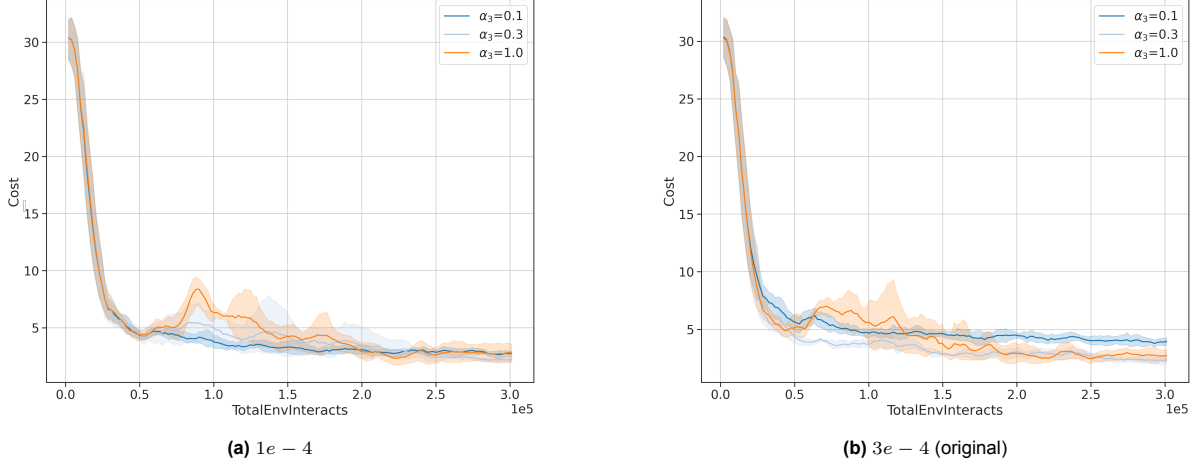


Figure E.14: Average test performance, measured as mean cost, and convergence of the **LAC** algorithm for select α_3 values between *two different λ learning rates* in the **FetchReach (infinite horizon) environment**. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

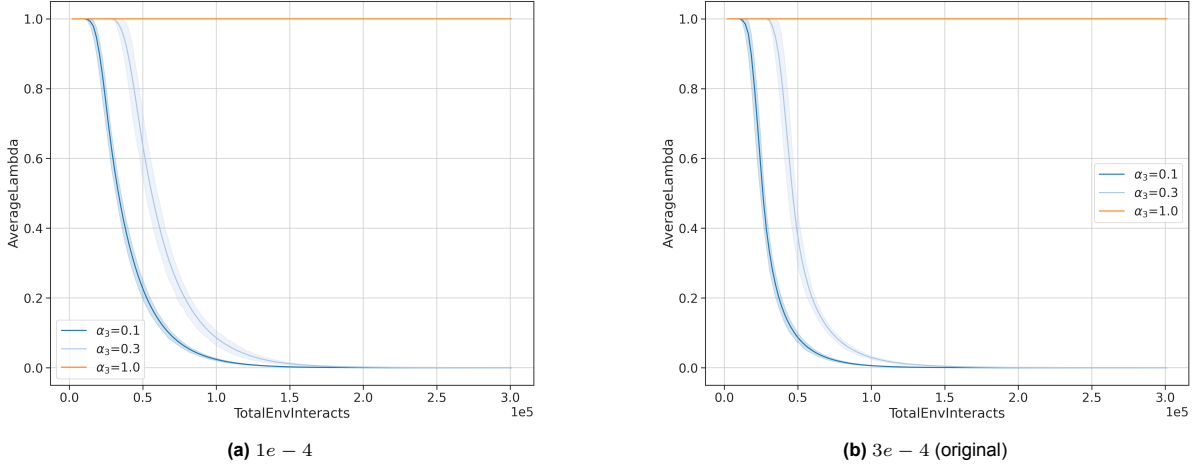


Figure E.15: Convergence of λ values for the **LAC** algorithm for select α_3 values between *two different λ learning rates* in the **FetchReach (infinite horizon) environment**. Each line represents the mean λ across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

E.1.1.1.4 Impact of Actor Network Structure

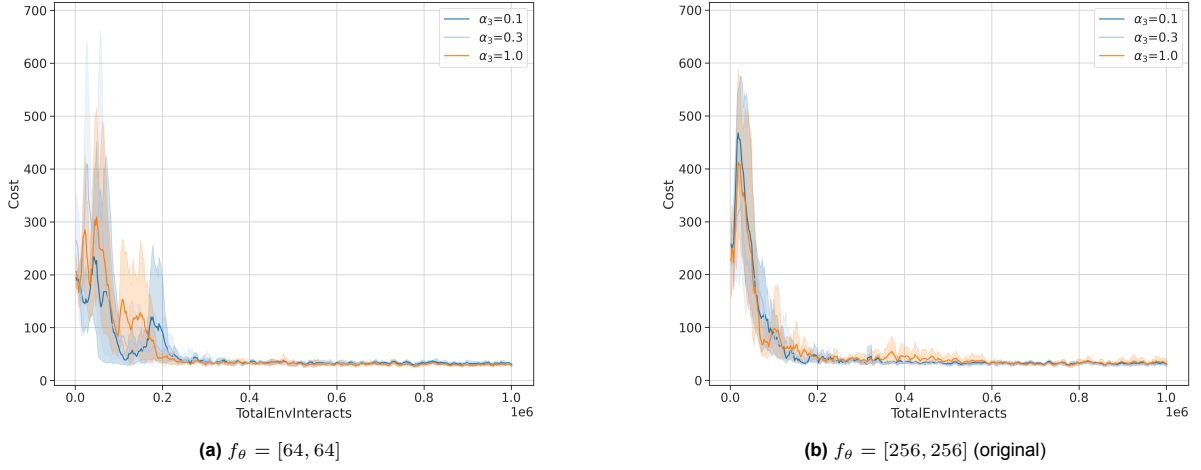


Figure E.16: Average test performance, measured as mean cost, and convergence of the LAC algorithm for select α_3 values between two different actor network structures in the **CartPole** environment. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

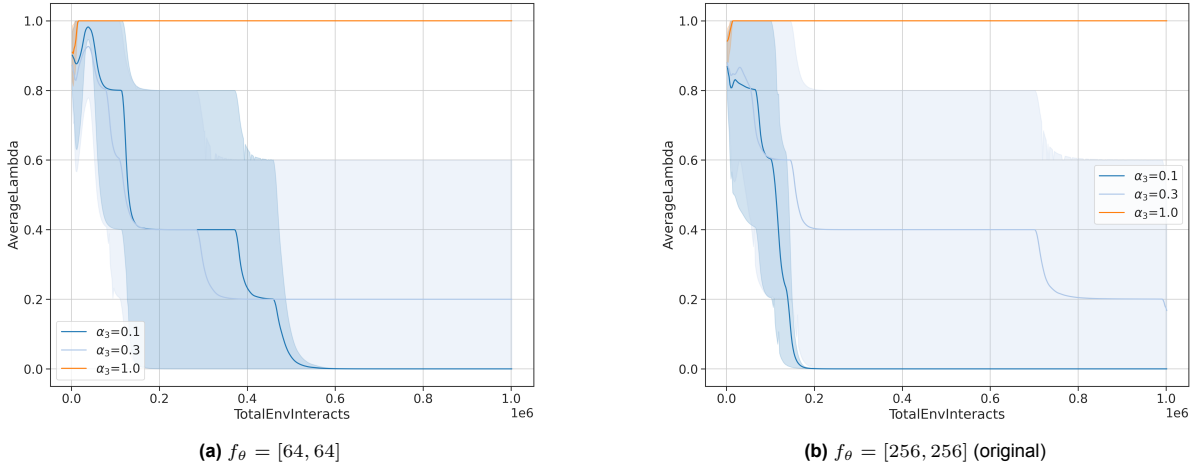


Figure E.17: Convergence of λ values for the LAC algorithm for select α_3 values between two different actor network structures in the **CartPole** environment. Each line represents the mean λ across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

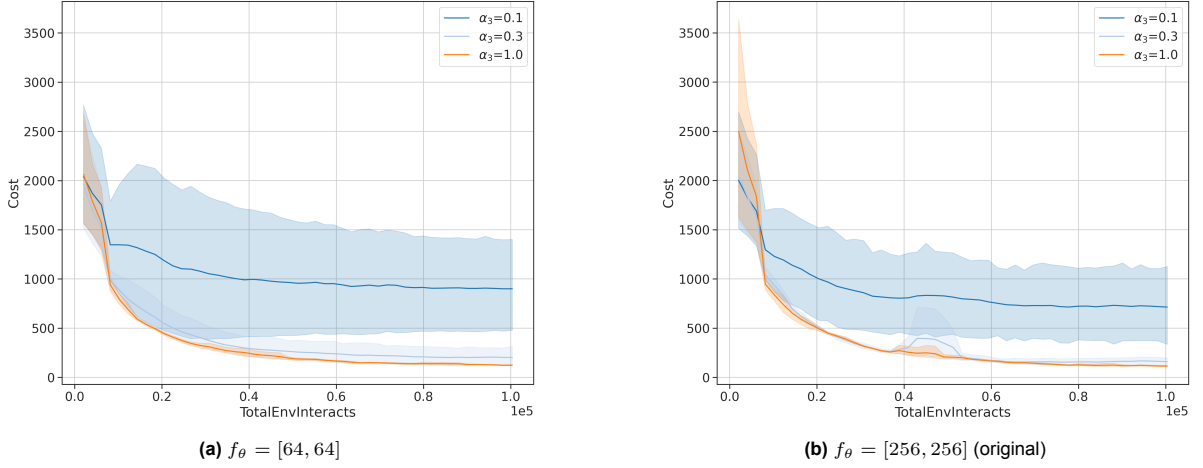


Figure E.18: Average test performance, measured as mean cost, and convergence of the **LAC** algorithm for select α_3 values between *two different actor network structures* in the **GRN environment**. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

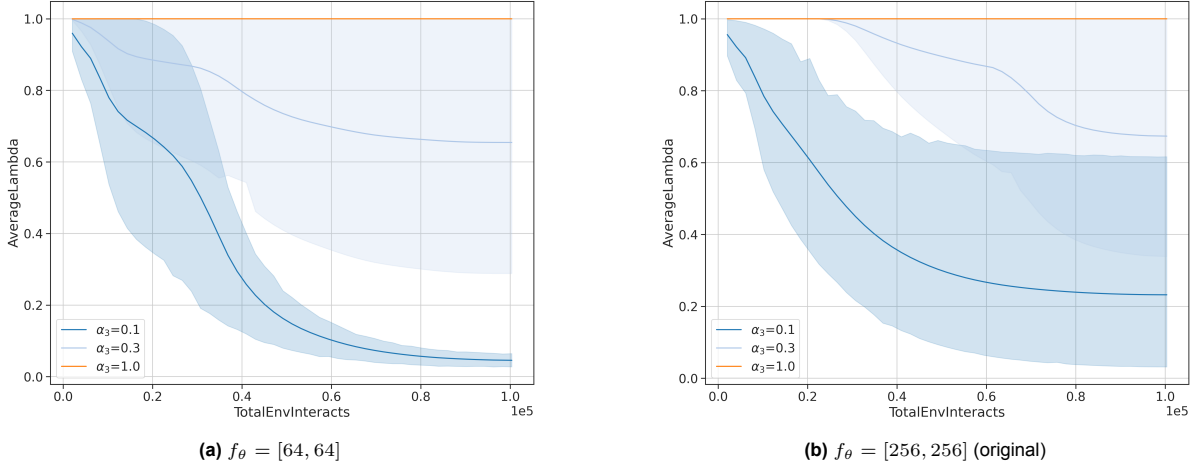


Figure E.19: Convergence of λ values for the **LAC** algorithm for select α_3 values between *two different actor network structures* in the **GRN environment**. Each line represents the mean λ across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

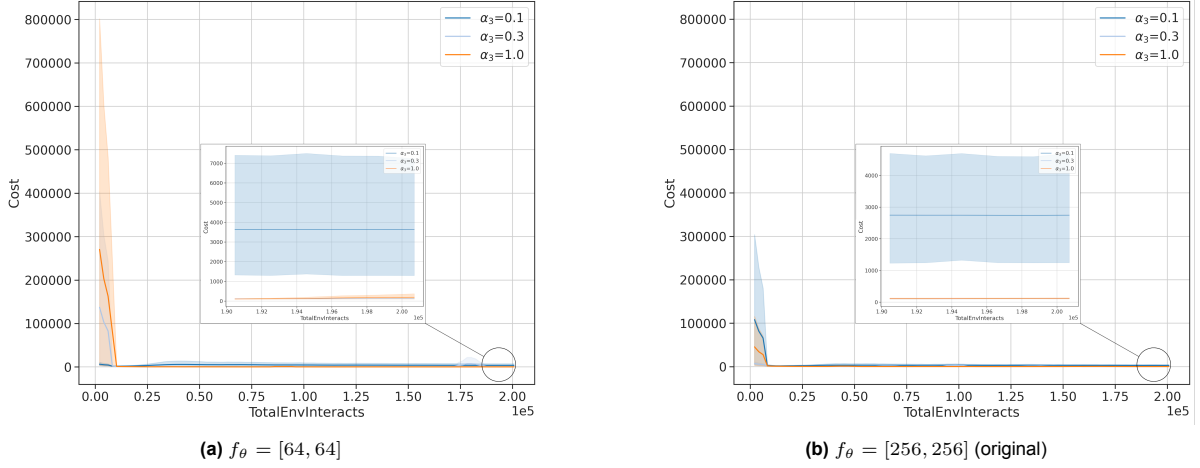


Figure E.20: Average test performance, measured as mean cost, and convergence of the **LAC** algorithm for select α_3 values between *two different actor network structures* in the **CompGRN environment**. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

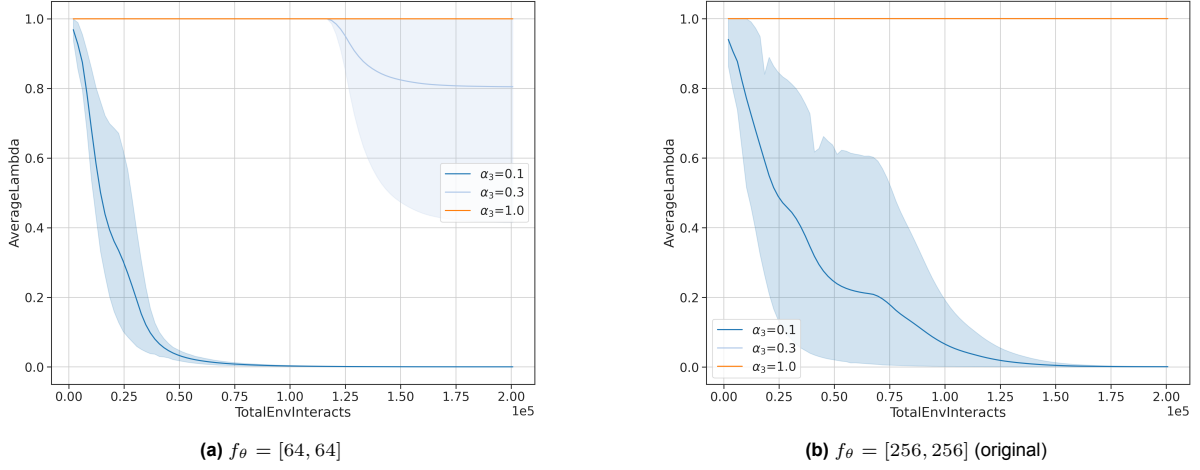


Figure E.21: Convergence of λ values for the **LAC** algorithm for select α_3 values between *two different actor network structures* in the **CompGRN environment**. Each line represents the mean λ across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

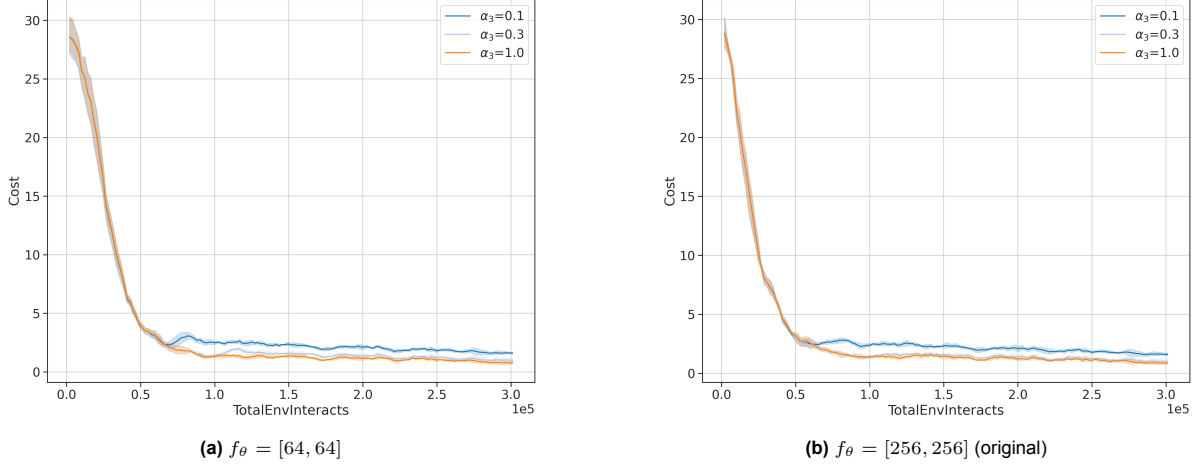


Figure E.22: Average test performance, measured as mean cost, and convergence of the **LAC** algorithm for select α_3 values between *two different actor network structures* in the **FetchReach** environment. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

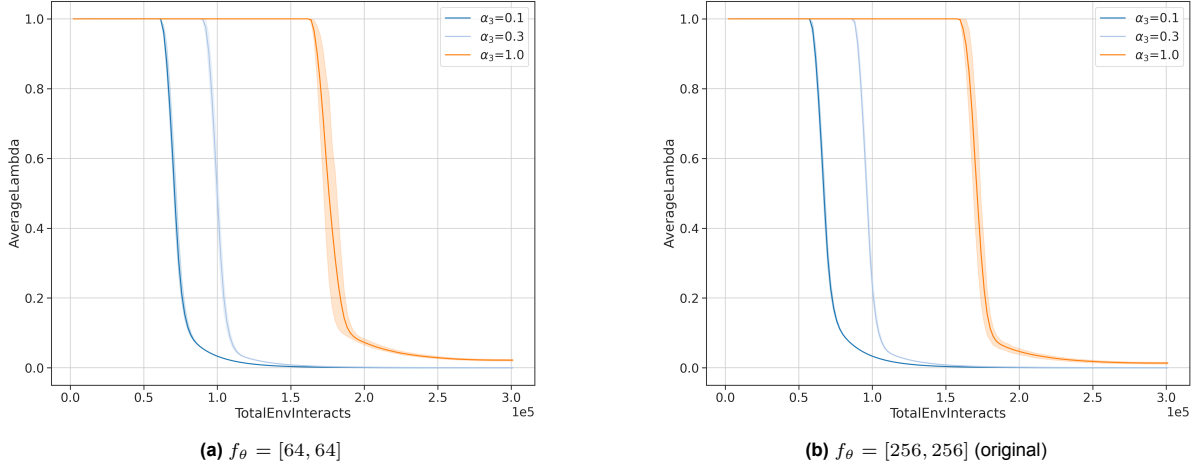


Figure E.23: Convergence of λ values for the **LAC** algorithm for select α_3 values between *two different actor network structures* in the **FetchReach** environment. Each line represents the mean λ across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

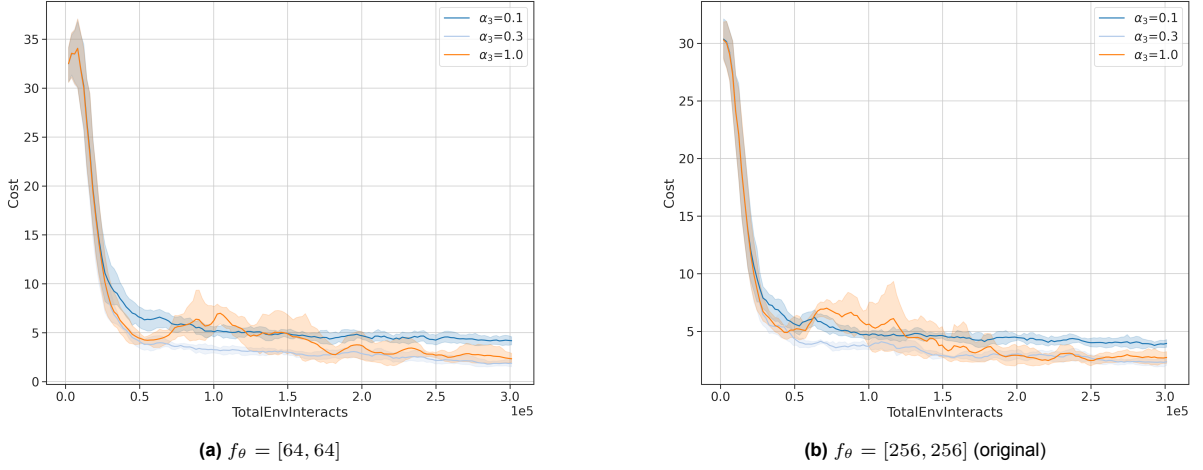


Figure E.24: Average test performance, measured as mean cost, and convergence of the **LAC** algorithm for select α_3 values between *two different actor network structures* in the **FetchReach (infinite horizon)** environment. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

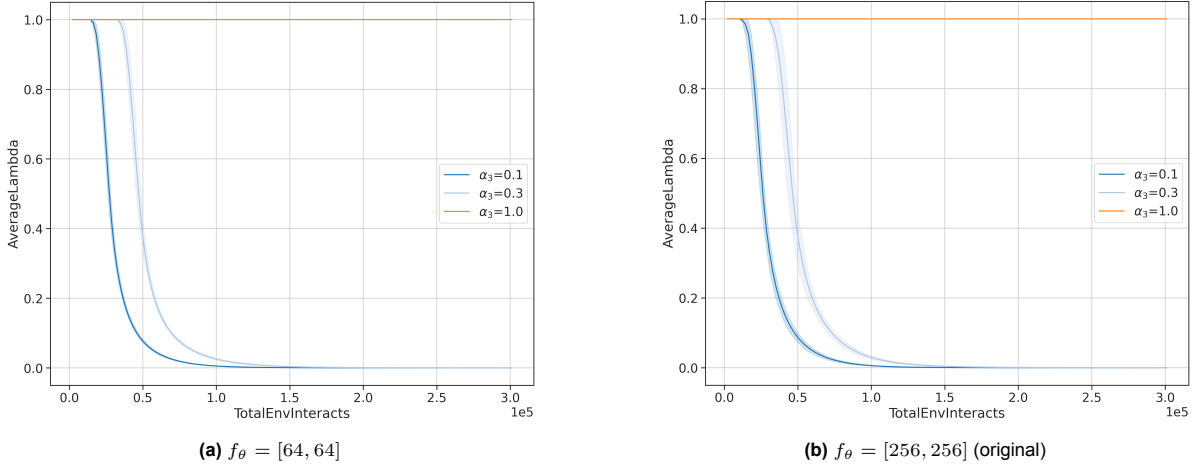


Figure E.25: Convergence of λ values for the **LAC** algorithm for select α_3 values between *two different actor network structures* in the **FetchReach (infinite horizon)** environment. Each line represents the mean λ across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

E.1.1.1.5 Impact of Shoter Finite Horizon Length

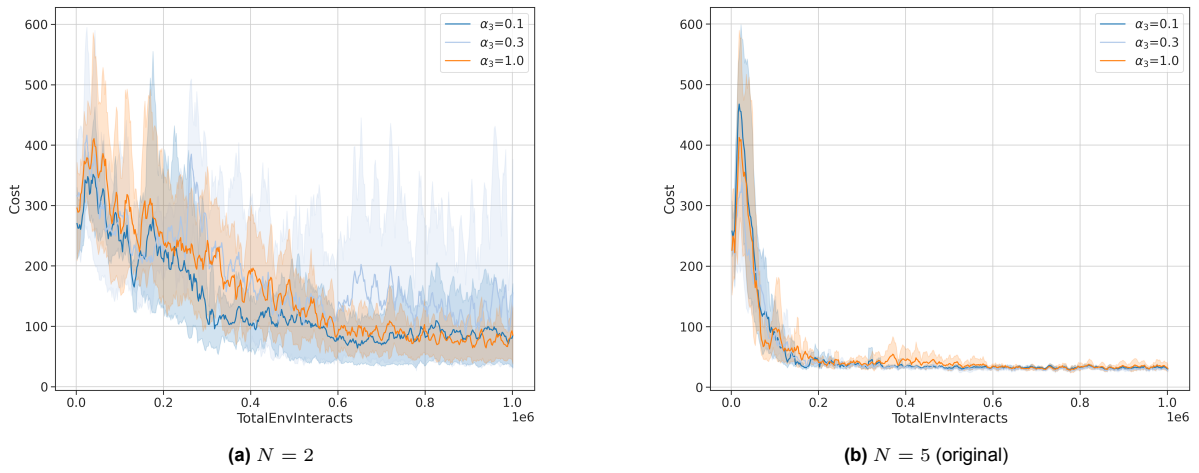


Figure E.26: Average test performance, measured as mean cost, and convergence of the **LAC** algorithm for select α_3 values between two different finite horizon lengths in the **CartPole** environment. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

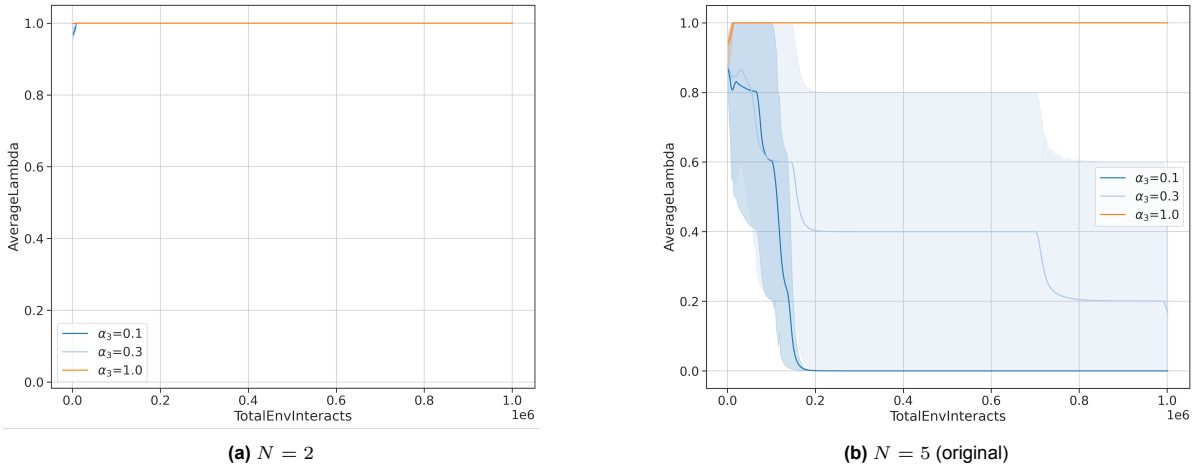


Figure E.27: Convergence of λ values for the **LAC** algorithm for select α_3 values between two different finite horizon lengths in the **CartPole** environment. Each line represents the mean λ across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

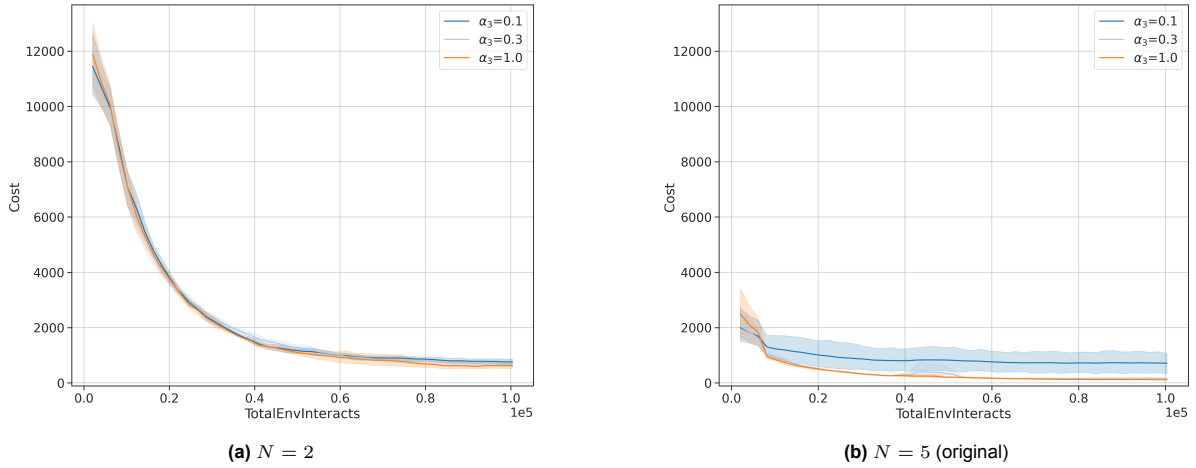


Figure E.28: Average test performance, measured as mean cost, and convergence of the **LAC** algorithm for select α_3 values between two different finite horizon lengths in the **GRN environment**. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

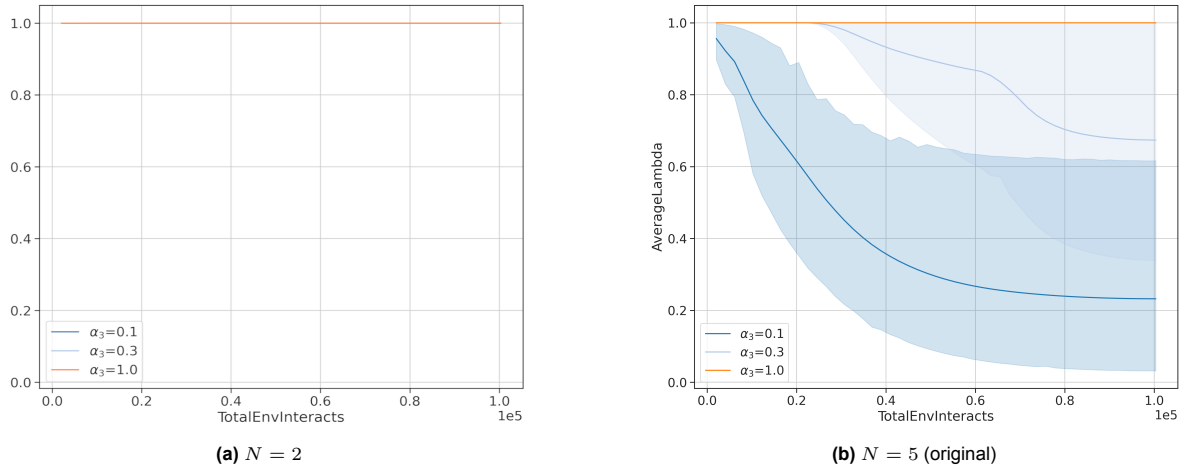


Figure E.29: Convergence of λ values for the **LAC** algorithm for select α_3 values between two different finite horizon lengths in the **GRN environment**. Each line represents the mean λ across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

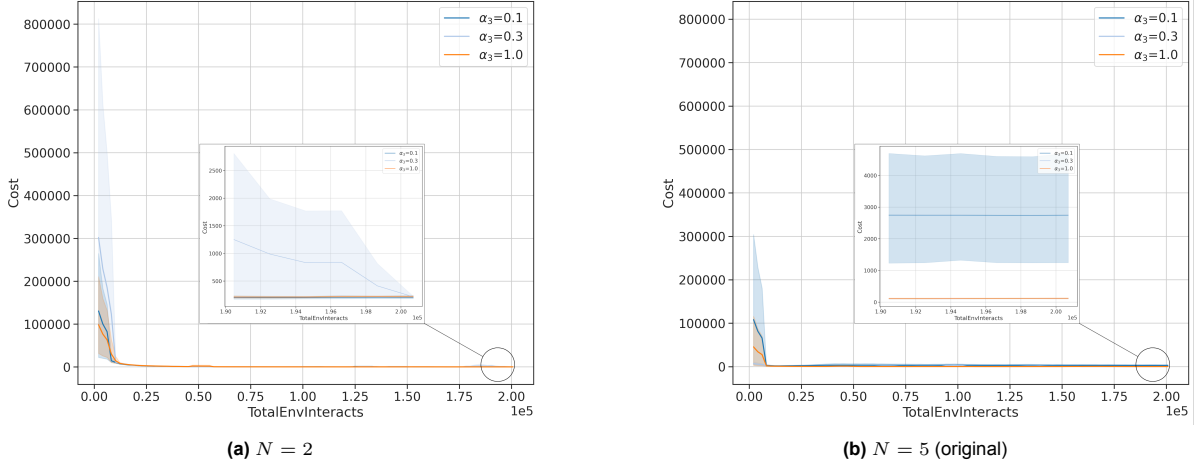


Figure E.30: Average test performance, measured as mean cost, and convergence of the **LAC** algorithm for select α_3 values between two different finite horizon lengths in the **CompGRN environment**. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

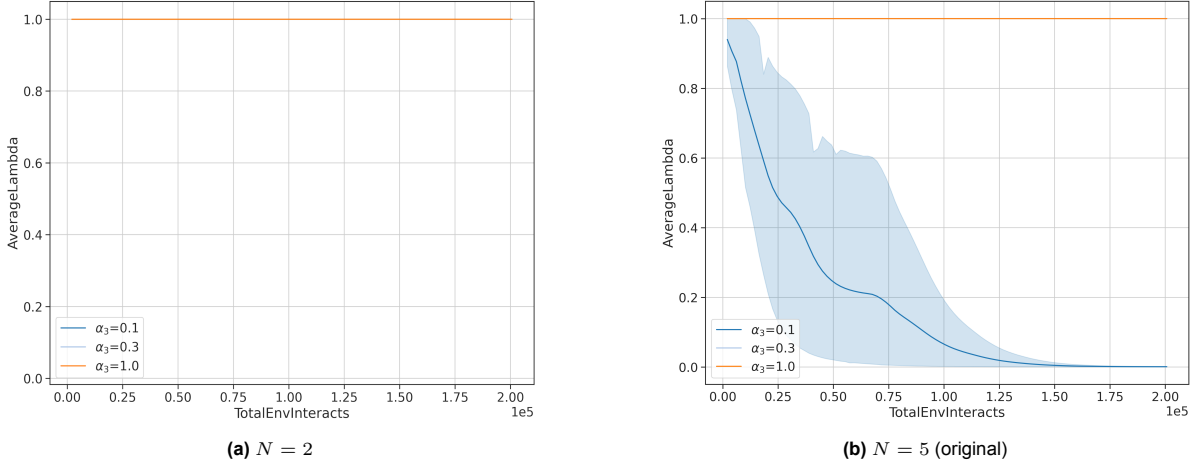


Figure E.31: Convergence of λ values for the **LAC** algorithm for select α_3 values between two different finite horizon lengths in the **CompGRN environment**. Each line represents the mean λ across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

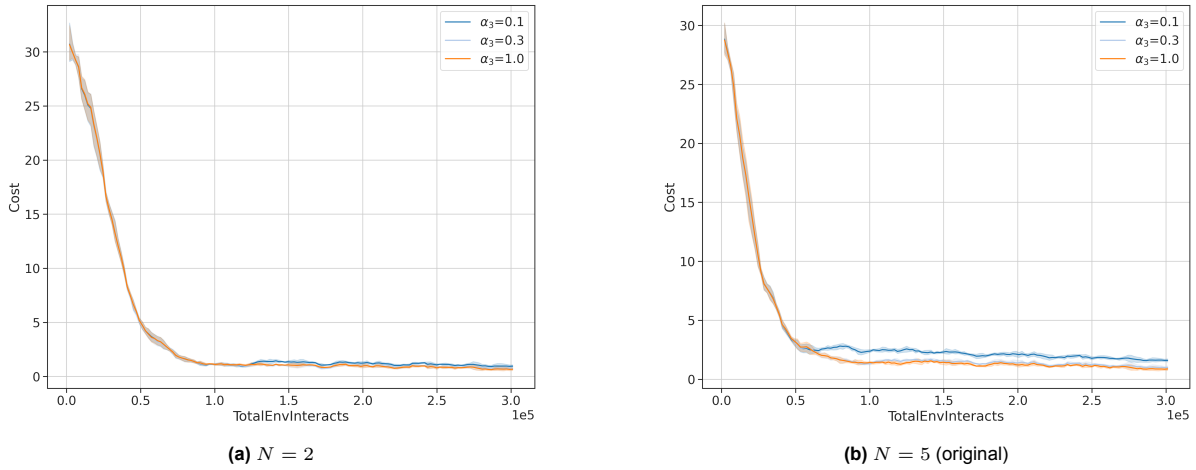


Figure E.32: Average test performance, measured as mean cost, and convergence of the **LAC** algorithm for select α_3 values between two different finite horizon lengths in the **FetchReach** environment. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

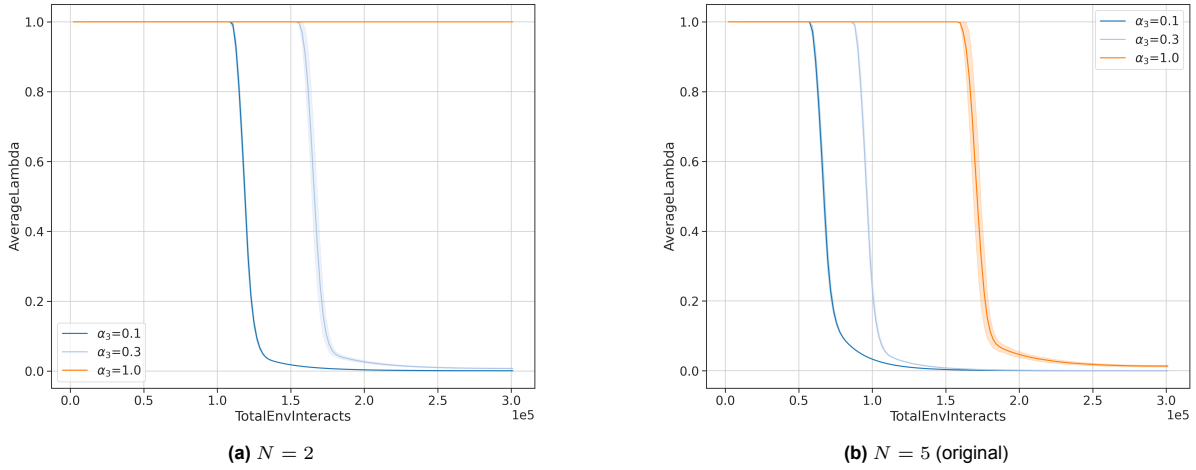


Figure E.33: Convergence of λ values for the **LAC** algorithm for select α_3 values between two different finite horizon lengths in the **FetchReach** environment. Each line represents the mean λ across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

E.1.1.1.6 Effect of Critic Network Structure

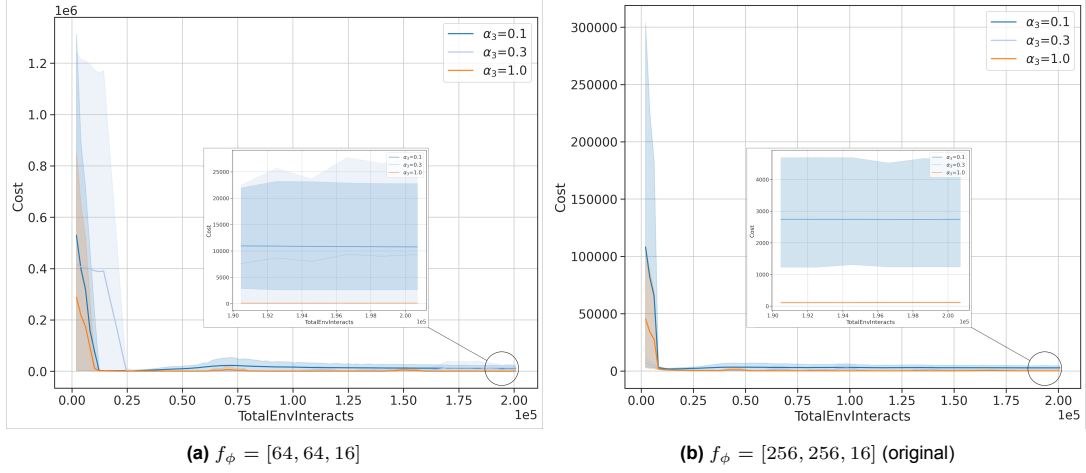


Figure E.34: Average test performance, measured as mean cost, and convergence of the **LAC** algorithm for select α_3 values between *two different critic network architectures* in the **CompGRN** environment. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

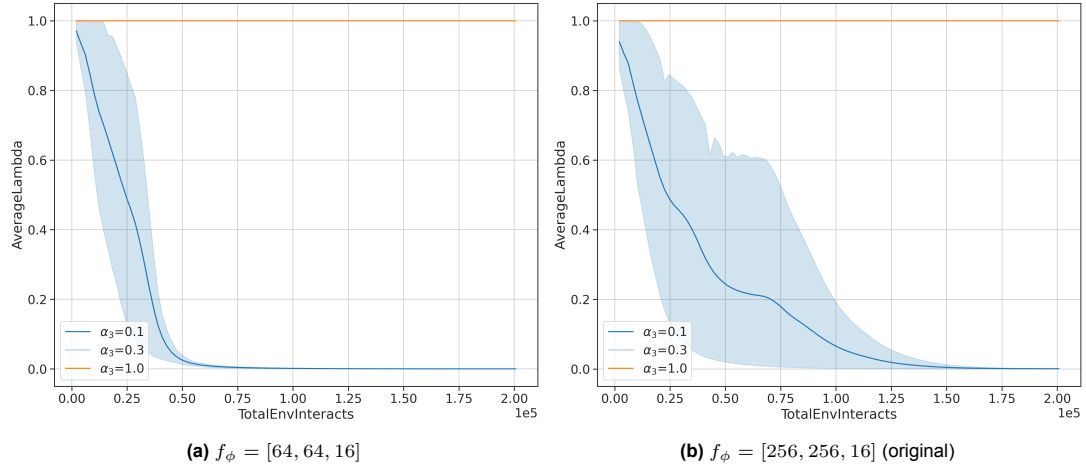


Figure E.35: Convergence of λ values for the **LAC** algorithm for select α_3 values between *two different critic network architectures* in the **CompGRN** environment. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

E.1.1.1.7 Analysis of Alpha3 in Han et al.'s Original Codebase

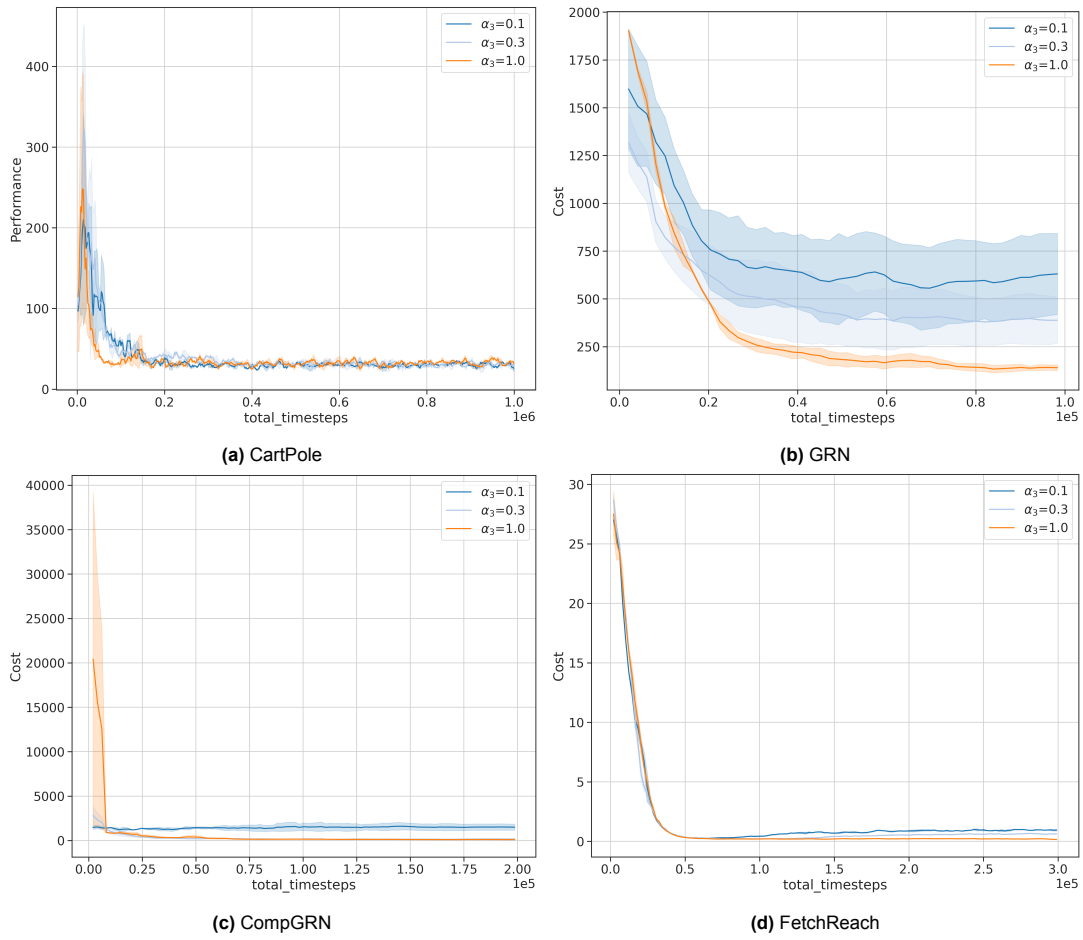


Figure E.36: Average test performance, measured as mean cost, and convergence of the LAC algorithm for select α_3 values in different environments in the **original codebase** of Han et al.[9]. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

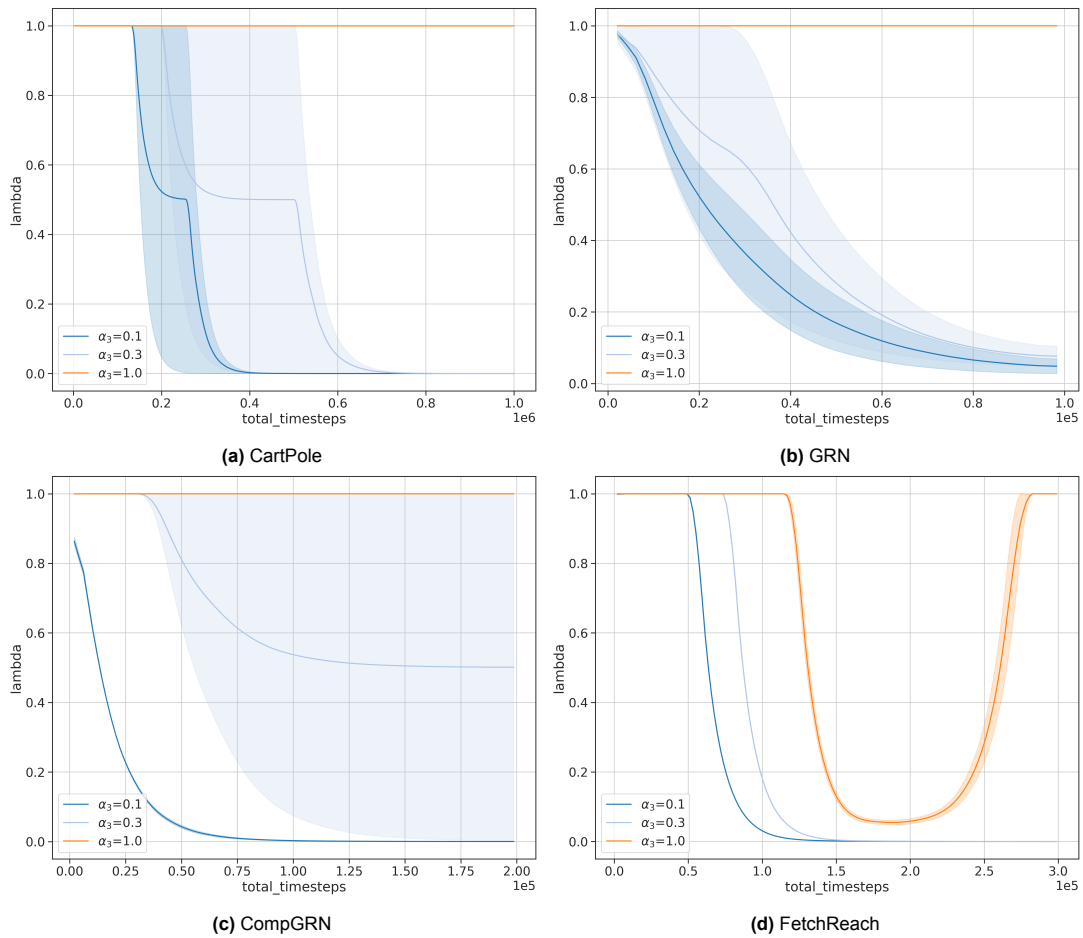


Figure E.37: Convergence of λ values during LAC algorithm training across various environments for select α_3 values in different environments in the **original codebase** of Han et al.[9], illustrating adherence to the stability constraint. Each line represents the mean λ value across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

E.1.1.2. Extended Alpha3 Tuning Test Performance and Convergence Results

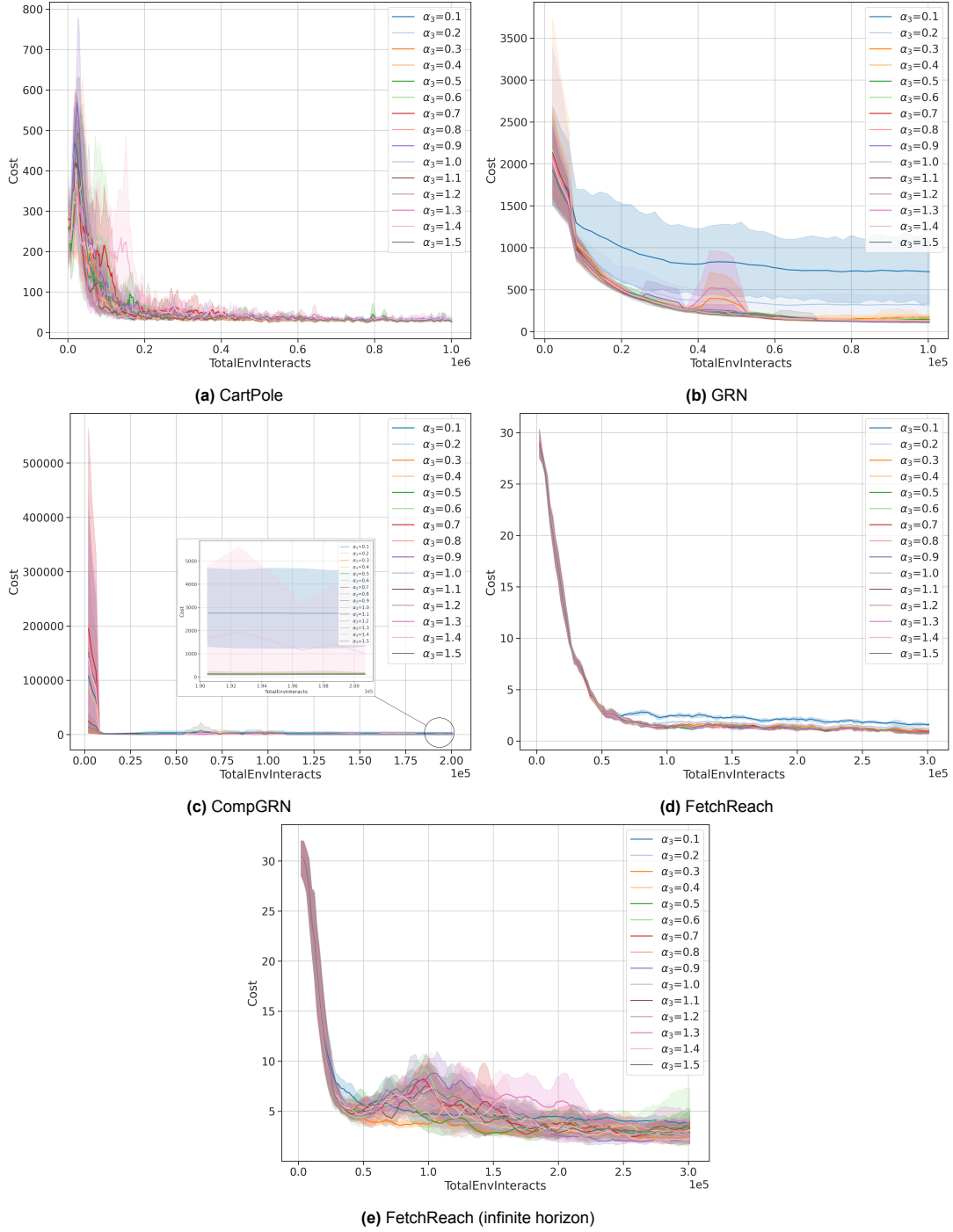


Figure E.38: Average test performance, measured as mean cost, and convergence of the LAC algorithm for select α_3 values in different environments. Each line represents the mean cost across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

E.1.1.3. Extended Alpha3 Tuning Lambda Convergence Results

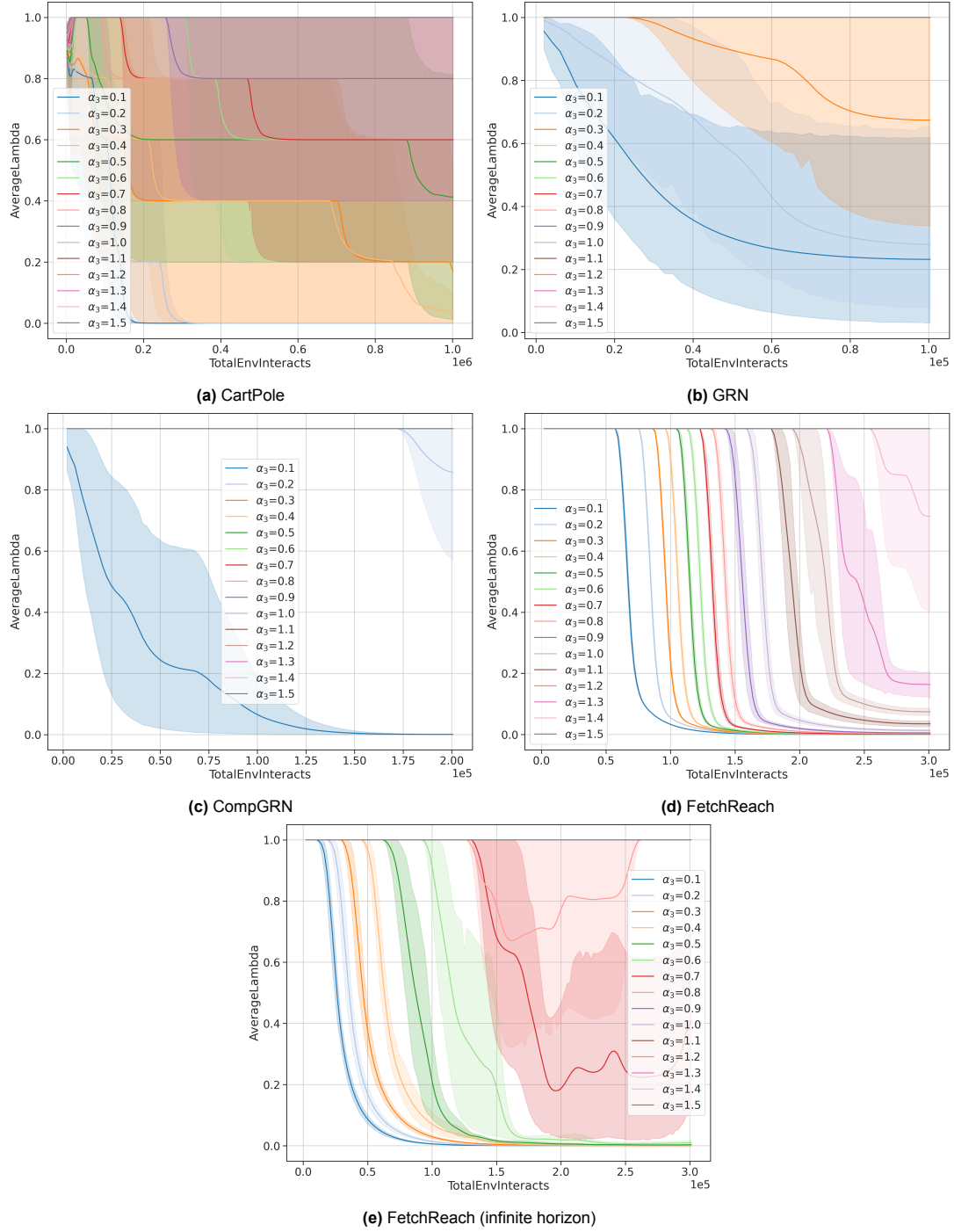


Figure E.39: Convergence of λ values during LAC algorithm training across various environments for select α_3 values, illustrating adherence to the stability constraint. Each line represents the mean λ value across five runs with different seeds, smoothed with a 5-epoch moving average to mitigate short-term fluctuations. Shaded areas denote 95% confidence intervals.

E.2. Tables

E.2.1. Reproduction Study

E.2.1.1. Alpha3 Tuning Hyperparameters

Hyperparameters	CartPole	GRN	CompGRN	FetchReach	FetchReach
Lyapunov candidate	Sum of cost	Sum of cost	Sum of cost	Sum of cost	Value
Time horizon (N)	5	5	5	5	∞
Minibatch size	256	256	256	256	256
Replay buffer size	1e6	1e6	1e6	1e6	1e6
Actor learning rate	$1e-4$	$1e-4$	$1e-4$	$1e-4$	$1e-4$
Critic learning rate	$3e-4$	$3e-4$	$3e-4$	$3e-4$	$3e-4$
Temperature learning rate	$1e-4$	$1e-4$	$1e-4$	$1e-4$	$1e-4$
Lyapunov learning rate	$3e-4$	$3e-4$	$3e-4$	$3e-4$	$3e-4$
Actor learning rate decay	Linear	Linear	Linear	Linear	Linear
Critic learning rate decay	Linear	Linear	Linear	Linear	Linear
Temperature learning rate decay	Linear	Linear	Linear	Linear	Linear
Lyapunov learning rate decay	Constant	Constant	Constant	Constant	Constant
Final actor learning rate	$1e-10$	$1e-9$	$1e-9$	$3.33e-10$	$3.33e-10$
Final critic learning rate	$3e-10$	$3e-9$	$3e-9$	$1e-9$	$1e-9$
Final temperature learning rate	$1e-10$	$1e-9$	$1e-9$	$3.33e-10$	$3.33e-10$
Target entropy	-1	-3	-4	-5	-5
Soft replacement (τ)	0.005	0.005	0.005	0.005	0.005
Discount (γ)	NaN	NaN	NaN	NaN	0.995
Stability strictness (α_3)	0.1-1.5	0.1-1.5	0.1-1.5	0.1-1.5	0.1-1.5
Critic structure (f_ϕ)	(64,64,16)	(256,256,16)	(256,256,16)	(64,64,16)	(64,64,16)
Actor structure (f_θ)	(256,256)	(256,256)	(256,256)	(256,256)	(256,256)
Update after (steps)	1000	1000	1000	1000	1000
Update interval (steps)	100	100	100	100	100
Steps per update	80	80	80	80	80
Total environment interactions	1e6	1e5	2e5	3e5	1e5
Initial entropy (α_0)	2.0	2.0	2.0	2.0	2.0
Initial stability (λ_0)	0.99	0.99	0.99	0.99	0.99

Table E.1: Hyperparameters employed in the α_3 tuning process for our *Reproduction Study*. The highlighted α_3 parameter was adjusted from 0.1 to 1.5 in 0.1 increments. In some literature, the Polyak factor (ρ) is used, equivalent to $1 - \tau$.

E.2.1.2. Additional Alpha3 Tuning Experiments

E.2.1.2.1 Extended Seed Analysis for GRN and CompGRN Environments

Number of Seeds	α_3	Cost Mean	Cost Std.	Cost Min	Cost Max	Convergence Epoch Mean	Convergence Epoch Std.
5	0.1	723.70	500.84	114.16	1471.02	30.60	3.05
	0.3	161.79	47.83	117.90	224.25	24.40	3.36
	1.0	120.49	11.29	111.65	139.27	26.60	4.34
10	0.1	791.72	601.52	114.16	2187.00	27.00	9.75
	0.3	175.23	76.51	109.40	366.55	26.10	7.32
	1.0	127.44	16.11	111.65	164.82	26.10	4.65

Table E.2: Test performance and convergence statistics for the **GRN environment** with varying α_3 values between five and ten runs with different seeds. Performance metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations.

Number of Seeds	α_3	Final Lambda Mean	Final Lambda Std.	Lambda Min	Lambda Max	Convergence Epoch Mean	Convergence Epoch Std.
5	0.1	0.23	0.43	0.02	1.00	-	-
	0.3	0.68	0.44	0.17	1.00	-	-
	1.0	1.00	0.00	1.00	1.00	-	-
10	0.1	0.14	0.30	0.02	1.00	-	-
	0.3	0.69	0.40	0.11	1.00	-	-
	1.0	1.00	0.00	1.00	1.00	-	-

Table E.3: Lambda convergence statistics for the **GRN environment** with varying α_3 values between five and ten runs with different seeds. Lambda mean metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations. Entries marked with '-' in the Convergence Epoch column failed to reach a lambda value below 0.1 within the observed epochs, indicating a lack of convergence under those conditions.

Number of Seeds	α_3	Cost Mean	Cost Std.	Cost Min	Cost Max	Convergence Epoch Mean	Convergence Epoch Std.
5	0.1	2741.10	2207.44	456.91	6336.07	37.60	41.99
	0.3	110.48	15.73	97.83	136.21	36.40	27.88
	1.0	123.22	36.35	98.08	185.62	29.20	34.81
10	0.1	2627.12	1915.75	456.91	6336.07	46.22	38.16
	0.3	124.99	34.80	97.83	211.29	29.80	23.00
	1.0	113.84	26.45	98.08	185.62	33.60	26.54

Table E.4: Test performance and convergence statistics for the **CompGRN environment** with varying α_3 values between five and ten runs with different seeds. Performance metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations.

Number of Seeds	α_3	Final Lambda Mean	Final Lambda Std.	Lambda Min	Lambda Max	Convergence Epoch Mean	Convergence Epoch Std.
5	0.1	8.00e-04	1.07e-03	2.02e-04	2.71e-03	33.20	19.93
	0.3	1.00	0.00	1.00	1.00	-	-
	1.0	1.00	0.00	1.00	1.00	-	-
10	0.1	7.39e-04	8.19e-04	2.02e-04	2.71e-03	33.10	19.79
	0.3	0.81	0.41	0.02	1.00	-	-
	1.0	1.00	0.00	1.00	1.00	-	-

Table E.5: Lambda convergence statistics for the **CompGRN environment** with varying α_3 values between five and ten runs with different seeds. Lambda mean metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations. Entries marked with '-' in the Convergence Epoch column failed to reach a lambda value below 0.1 within the observed epochs, indicating a lack of convergence under those conditions.

α_3	Cost Mean (GRN)	Cost Mean (CompGRN)
0.1	1471.01978	2140.55732
0.1	502.671742	6336.06989
0.1	542.805409	1114.68964
0.1	2186.99897	2986.12607
0.1	500.244046	5717.90567
0.1	114.164514	2162.5815
0.1	667.163692	456.914499
0.1	576.932388	1888.36629
0.1	865.883641	1763.78682
0.1	489.270504	1704.22548

Table E.6: Test performance comparison for each seed in the **GRN** and **CompGRN** environments with $\alpha_3 = 0.1$. Metrics are calculated from the final 10 epochs for stability. Highlighted cells indicate cost exceeding Han et al.[9]'s results.

E.2.1.3. Alpha3 Tuning Performance and Convergence Statistics

α_3	Cost Mean	Cost Std.	Cost Min	Cost Max	Convergence Epoch Mean	Convergence Epoch Std.
0.1	32.29	1.35	31.09	34.50	116.80	63.78
0.2	30.34	2.18	27.19	32.50	113.40	61.67
0.3	29.62	1.23	28.66	31.73	120.60	46.51
0.4	30.22	0.93	29.08	31.19	113.00	65.97
0.5	30.99	3.17	28.76	36.33	215.60	143.95
0.6	29.98	1.81	28.19	32.34	113.60	64.76
0.7	30.16	2.35	27.42	33.65	195.80	99.14
0.8	29.70	1.37	27.46	31.11	148.00	87.04
0.9	33.82	7.61	28.68	46.65	124.20	49.75
1.0	33.55	7.18	27.97	45.83	178.00	83.10
1.1	30.60	1.79	28.38	32.68	110.60	81.25
1.2	29.14	1.26	27.59	30.69	139.60	85.83
1.3	31.08	1.59	28.97	32.81	238.60	102.81
1.4	32.11	2.69	29.76	36.69	168.80	60.70
1.5	29.95	1.01	28.87	31.38	188.20	88.78

Table E.7: Test performance and convergence metrics for the **CartPole environment** with varying α_3 values, averaged over 5 randomly seeded runs. Performance metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations.

α_3	Cost Mean	Cost Std.	Cost Min	Cost Max	Convergence Epoch Mean	Convergence Epoch Std.
0.1	723.70	500.84	114.16	1471.02	30.60	3.05
0.2	320.78	147.02	129.45	531.01	22.60	6.02
0.3	161.79	47.83	117.90	224.25	24.40	3.36
0.4	160.14	74.67	117.49	292.57	30.60	10.62
0.5	139.53	27.94	113.54	186.70	28.80	4.92
0.6	127.78	15.56	105.40	141.20	25.20	3.83
0.7	120.96	4.67	116.74	128.91	25.60	2.41
0.8	127.11	6.95	116.61	133.66	26.80	4.09
0.9	122.11	2.82	117.62	125.02	28.60	4.16
1.0	120.49	11.29	111.65	139.27	26.60	4.34
1.1	120.39	5.87	115.88	128.87	27.00	3.39
1.2	123.30	6.66	114.37	129.37	26.60	3.36
1.3	118.90	5.08	110.83	124.78	28.60	4.34
1.4	135.11	20.35	118.00	168.25	26.80	4.76
1.5	114.05	7.26	104.39	121.79	30.20	5.54

Table E.8: Test performance and convergence metrics for the **GRN environment** with varying α_3 values, averaged over 5 randomly seeded runs. Performance metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations. Highlighted rows indicate significantly higher cost compared to Han et al.[9]’s results.

α_3	Cost Mean	Cost Std.	Cost Min	Cost Max	Convergence Epoch Mean	Convergence Epoch Std.
0.1	2741.10	2207.44	456.91	6336.07	37.60	41.99
0.2	115.27	13.21	93.16	128.37	33.40	35.43
0.3	110.48	15.73	97.83	136.21	36.40	27.88
0.4	129.68	58.24	95.25	232.80	24.60	22.52
0.5	139.46	67.24	86.66	254.26	16.30	18.52
0.6	109.50	10.40	96.91	122.23	44.40	36.70
0.7	99.41	10.78	87.28	111.21	15.00	7.65
0.8	109.45	11.64	97.16	128.67	18.40	14.84
0.9	114.81	34.81	82.93	172.16	38.40	36.48
1.0	123.22	36.35	98.08	185.62	29.20	34.81
1.1	106.71	9.41	97.35	118.89	26.40	20.23
1.2	118.90	33.26	94.30	177.15	45.20	33.31
1.3	105.74	12.71	89.73	121.46	24.00	24.21
1.4	1126.49	2243.94	98.29	5140.30	38.20	35.25
1.5	100.66	9.32	86.22	110.36	27.40	18.41

Table E.9: Test performance and convergence metrics for the **CompGRN environment** with varying α_3 values, averaged over 5 randomly seeded runs. Performance metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations. Highlighted rows indicate significantly higher cost compared to Han et al.[9]’s results.

α_3	Cost Mean	Cost Std.	Cost Min	Cost Max	Convergence Epoch Mean	Convergence Epoch Std.
0.1	1.60	0.14	1.38	1.78	39.80	12.87
0.2	1.22	0.12	1.11	1.42	30.20	3.11
0.3	1.01	0.13	0.85	1.20	34.20	1.92
0.4	0.93	0.15	0.76	1.17	34.20	1.92
0.5	0.90	0.10	0.81	1.06	33.60	3.65
0.6	0.89	0.14	0.76	1.11	34.00	2.24
0.7	0.92	0.13	0.81	1.14	34.00	3.08
0.8	0.84	0.12	0.76	1.06	34.00	2.24
0.9	0.82	0.14	0.72	1.06	34.80	2.28
1.0	0.86	0.12	0.71	1.05	34.40	2.19
1.1	0.84	0.12	0.71	1.04	34.40	2.30
1.2	0.82	0.16	0.72	1.10	34.40	1.82
1.3	0.82	0.10	0.76	1.00	36.00	1.58
1.4	0.83	0.14	0.69	1.07	35.00	2.55
1.5	0.81	0.13	0.69	1.01	36.40	3.13

Table E.10: Test performance and convergence metrics for the **FetchReach environment** with varying α_3 values, averaged over 5 randomly seeded runs. Performance metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations.

α_3	Cost Mean	Cost Std.	Cost Min	Cost Max	Convergence Epoch Mean	Convergence Epoch Std.
0.1	3.92	0.41	3.45	4.50	60.60	21.41
0.2	2.78	0.62	2.18	3.79	65.80	10.69
0.3	2.34	0.40	1.73	2.72	87.60	37.31
0.4	2.99	1.36	1.98	4.77	83.00	28.50
0.5	3.05	2.06	1.24	6.42	81.00	24.83
0.6	4.54	2.95	1.85	8.96	76.80	22.62
0.7	2.72	1.31	1.77	4.59	97.80	15.87
0.8	3.08	0.47	2.27	3.40	98.20	35.05
0.9	2.17	0.47	1.42	2.70	91.60	6.84
1.0	2.74	0.74	1.98	3.70	84.80	33.45
1.1	3.30	0.73	2.35	4.28	77.60	22.35
1.2	2.29	0.52	1.85	3.11	102.80	27.37
1.3	4.03	1.42	2.26	5.97	101.80	16.78
1.4	2.19	0.31	2.02	2.75	91.80	39.71
1.5	3.43	1.26	2.38	5.18	85.20	32.41

Table E.11: Test performance and convergence metrics for the **FetchReach (infinite horizon) environment** with varying α_3 values, averaged over 5 randomly seeded runs. Performance metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations.

E.2.1.4. Alpha3 Tuning Lambda Convergence Statistics

α_3	Final Lambda Mean	Final Lambda Std.	Lambda Min	Lambda Max	Convergence Epoch Mean	Convergence Epoch Std.
0.1	4.44e-10	1.89e-10	3.01e-10	7.57e-10	59.80	22.06
0.2	1.18e-09	9.64e-10	5.47e-10	2.79e-09	70.20	45.36
0.3	0.20	0.44	5.63e-10	0.98	-	-
0.4	0.04	0.09	8.45e-10	0.20	219.00	193.39
0.5	0.42	0.53	1.43e-09	1.00	-	-
0.6	0.60	0.55	7.72e-09	1.00	-	-
0.7	0.60	0.55	3.35e-09	1.00	-	-
0.8	0.80	0.45	7.09e-09	1.00	-	-
0.9	0.80	0.45	1.97e-08	1.00	-	-
1.0	1.00	0.00	1.00	1.00	-	-
1.1	1.00	0.00	1.00	1.00	-	-
1.2	1.00	0.00	1.00	1.00	-	-
1.3	1.00	0.00	1.00	1.00	-	-
1.4	1.00	0.00	1.00	1.00	-	-
1.5	1.00	0.00	1.00	1.00	-	-

Table E.12: Lambda convergence statistics for the **CartPole environment** with varying α_3 values, averaged over 5 randomly seeded runs. Lambda mean metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations. Entries marked with '-' in the Convergence Epoch column failed to reach a lambda value below 0.1 within the observed epochs, indicating a lack of convergence under those conditions. Highlighted row indicates a higher than expected final lambda value.

α_3	Final Lambda Mean	Final Lambda Std.	Lambda Min	Lambda Max	Convergence Epoch Mean	Convergence Epoch Std
0.1	0.23	0.43	0.02	1.00	-	-
0.2	0.29	0.40	0.06	1.00	-	-
0.3	0.68	0.44	0.17	1.00	-	-
0.4	1.00	0.00	1.00	1.00	-	-
0.5	1.00	0.00	1.00	1.00	-	-
0.6	1.00	0.00	1.00	1.00	-	-
0.7	1.00	0.00	1.00	1.00	-	-
0.8	1.00	0.00	1.00	1.00	-	-
0.9	1.00	0.00	1.00	1.00	-	-
1.0	1.00	0.00	1.00	1.00	-	-
1.1	1.00	0.00	1.00	1.00	-	-
1.2	1.00	0.00	1.00	1.00	-	-
1.3	1.00	0.00	1.00	1.00	-	-
1.4	1.00	0.00	1.00	1.00	-	-
1.5	1.00	0.00	1.00	1.00	-	-

Table E.13: Lambda convergence statistics for the **GRN environment** with varying α_3 values, averaged over 5 randomly seeded runs. Final Lambda mean metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations. Entries marked with '-' in the Convergence Epoch column failed to reach a lambda value below 0.1 within the observed epochs, indicating a lack of convergence under those conditions.

α_3	Final Lambda Mean	Final Lambda Std.	Lambda Min	Lambda Max	Convergence Epoch Mean	Convergence Epoch Std
0.1	8.00e-04	1.07e-03	2.01e-04	2.71e-03	33.2	19.93
0.2	0.89	0.25	0.45	1.00	-	-
0.3	1.00	0.00	1.00	1.00	-	-
0.4	1.00	0.00	1.00	1.00	-	-
0.5	1.00	0.00	1.00	1.00	-	-
0.6	1.00	0.00	1.00	1.00	-	-
0.7	1.00	0.00	1.00	1.00	-	-
0.8	1.00	0.00	1.00	1.00	-	-
0.9	1.00	0.00	1.00	1.00	-	-
1.0	1.00	0.00	1.00	1.00	-	-
1.1	1.00	0.00	1.00	1.00	-	-
1.2	1.00	0.00	1.00	1.00	-	-
1.3	1.00	0.00	1.00	1.00	-	-
1.4	1.00	0.00	1.00	1.00	-	-
1.5	1.00	0.00	1.00	1.00	-	-

Table E.14: Lambda convergence statistics for the **CompGRN environment** with varying α_3 values, averaged over 5 randomly seeded runs. Final Lambda metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations. Entries marked with '-' in the Convergence Epoch column failed to reach a lambda value below 0.1 within the observed epochs, indicating a lack of convergence under those conditions.

α_3	Final Lambda Mean	Final Lambda Std.	Lambda Min	Lambda Max	Convergence Epoch Mean	Convergence Epoch Std
0.1	2.15e-04	4.20e-06	2.11e-04	2.21e-04	47.2	0.45
0.2	3.77e-04	4.46e-06	3.71e-04	3.83e-04	52.0	0.00
0.3	5.23e-04	7.08e-06	5.17e-04	5.34e-04	56.2	0.45
0.4	6.64e-04	1.94e-05	6.40e-04	6.86e-04	60.2	0.45
0.5	8.19e-04	2.34e-05	7.82e-04	8.40e-04	64.2	0.45
0.6	1.00e-03	5.08e-05	9.54e-04	1.07e-03	68.2	0.84
0.7	1.67e-03	2.09e-04	1.50e-03	1.95e-03	72.4	0.89
0.8	3.12e-03	5.00e-04	2.59e-03	3.78e-03	78.0	1.22
0.9	5.86e-03	1.25e-03	4.49e-03	7.75e-03	84.6	2.30
1.0	0.01	0.003	0.009	0.02	94.8	2.77
1.1	0.04	0.01	0.02	0.05	107.2	4.55
1.2	0.08	0.02	0.06	0.09	117.2	3.83
1.3	0.16	0.05	0.10	0.23	-	-
1.4	0.76	0.35	0.23	1.00	-	-
1.5	1.00	0.00	1.00	1.00	-	-

Table E.15: Lambda convergence statistics for the **FetchReach environment** with varying α_3 values, averaged over 5 randomly seeded runs. Final Lambda mean metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations. Entries marked with '-' in the Convergence Epoch column failed to reach a lambda value below 0.1 within the observed epochs, indicating a lack of convergence under those conditions.

α_3	Mean of Seed Means	Std of Seed Means	Min of Seed Means	Max of Seed Means	Convergence Epoch Mean	Convergence Epoch Std
0.1	3.44e-05	6.35e-06	2.43e-05	4.01e-05	31.4	2.19
0.2	1.10e-04	2.44e-05	7.82e-05	1.40e-04	37.6	2.70
0.3	3.18e-04	9.56e-05	2.17e-04	4.33e-04	45.2	2.17
0.4	1.03e-03	4.44e-04	5.25e-04	1.62e-03	54.6	7.80
0.5	2.84e-03	2.04e-03	1.06e-03	5.39e-03	59.4	4.67
0.6	9.08e-03	7.59e-03	4.01e-03	0.02	79.4	17.30
0.7	0.31	0.40	0.008	1.00	71.33	61.34
0.8	1.00	0.00	1.00	1.00	-	-
0.9	1.00	0.00	1.00	1.00	-	-
1.0	1.00	0.00	1.00	1.00	-	-
1.1	1.00	0.00	1.00	1.00	-	-
1.2	1.00	0.00	1.00	1.00	-	-
1.3	1.00	0.00	1.00	1.00	-	-
1.4	1.00	0.00	1.00	1.00	-	-
1.5	1.00	0.00	1.00	1.00	-	-

Table E.16: Lambda convergence statistics for the **FetchReach (infinite horizon) environment** with varying Alpha3 values, averaged over 5 randomly seeded runs. Final Lambda mean metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations. Entries marked with '-' in the Convergence Epoch column failed to reach a lambda value below 0.1 within the observed epochs, indicating a lack of convergence under those conditions.

E.2.1.5. LAC SAC Comparison Performance and Convergence Statistics

Condition	Cost Mean	Cost Std.	Cost Min	Cost Max	Convergence Epoch Mean	Convergence Epoch Std.
0.1	31.95	1.31	30.40	34.03	116.80	63.78
0.2	29.96	2.12	27.87	33.00	83.40	31.25
0.3	29.44	0.93	28.25	30.76	119.00	44.54
0.4	29.77	2.53	25.38	31.43	113.00	65.97
0.5	30.80	4.18	25.56	36.92	220.00	138.27
0.6	29.76	2.47	26.97	33.56	113.80	64.66
0.7	29.52	3.58	23.92	33.71	195.80	99.14
0.8	29.34	2.66	25.40	32.34	109.00	69.76
0.9	32.21	3.29	28.82	35.88	124.20	49.75
1.0	33.76	8.43	25.58	47.35	178.00	83.10
1.1	30.27	2.35	27.85	33.69	110.60	81.25
1.2	28.95	2.42	25.85	32.33	137.80	83.52
1.3	31.00	2.96	26.11	33.60	225.80	103.15
1.4	31.67	0.93	30.09	32.37	167.20	60.99
1.5	29.53	1.33	27.19	30.45	188.00	88.96
SAC	36.38	12.91	25.24	56.19	451.75	34.70

Table E.17: Test performance and convergence statistics for the **CartPole environment** with different conditions. Performance metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations.

Condition	Cost Mean	Cost Std.	Cost Min	Cost Max	Convergence Epoch Mean	Convergence Epoch Std.
0.1	723.70	500.84	114.16	1471.02	30.60	3.05
0.2	320.78	147.02	129.45	531.01	22.60	6.02
0.3	161.79	47.83	117.90	224.25	24.40	3.36
0.4	160.14	74.67	117.49	292.57	30.60	10.62
0.5	139.53	27.94	113.54	186.70	28.80	4.92
0.6	127.78	15.56	105.40	141.20	25.20	3.83
0.7	120.96	4.67	116.74	128.91	25.60	2.41
0.8	127.11	6.95	116.61	133.66	26.80	4.09
0.9	122.11	2.82	117.62	125.02	28.60	4.16
1.0	120.49	11.29	111.65	139.27	26.60	4.34
1.1	120.39	5.87	115.88	128.87	27.00	3.39
1.2	123.30	6.66	114.37	129.37	26.60	3.36
1.3	118.90	5.08	110.83	124.78	28.60	4.34
1.4	135.11	20.35	118.00	168.25	26.80	4.76
1.5	114.05	7.26	104.39	121.79	30.20	5.54
SAC	517.63	357.86	214.35	1125.49	28.00	11.36

Table E.18: Test performance and convergence statistics for the **GRN environment** with different conditions. Performance metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations.

Condition	Cost Mean	Cost Std.	Cost Min	Cost Max	Convergence Epoch Mean	Convergence Epoch Std.
0.1	2741.10	2207.44	456.91	6336.07	37.60	41.99
0.2	115.27	13.21	93.16	128.37	33.40	35.43
0.3	110.48	15.73	97.83	136.21	36.40	27.88
0.4	129.68	58.24	95.25	232.80	24.60	22.52
0.5	139.46	67.24	86.66	254.26	16.25	18.52
0.6	109.50	10.40	96.91	122.23	44.40	36.70
0.7	99.41	10.78	87.28	111.21	15.00	7.65
0.8	109.45	11.64	97.16	128.67	18.40	14.84
0.9	114.81	34.81	82.93	172.16	38.40	36.48
1.0	123.22	36.35	98.08	185.62	29.20	34.81
1.1	106.71	9.41	97.35	118.89	26.40	20.23
1.2	118.90	33.26	94.30	177.15	45.20	33.31
1.3	105.74	12.71	89.73	121.46	24.00	24.21
1.4	1126.49	2243.94	98.29	5140.30	38.20	35.25
1.5	100.66	9.32	86.22	110.36	27.40	18.41
SAC	3684.69	3357.31	1064.39	8684.41	54.00	29.07

Table E.19: Test performance and convergence statistics for the **CompGRN environment** with different conditions. Performance metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations.

Condition	Cost Mean	Cost Std.	Cost Min	Cost Max	Convergence Epoch Mean	Convergence Epoch Std.
0.1	1.60	0.14	1.38	1.78	39.80	12.87
0.2	1.22	0.12	1.11	1.42	30.20	3.11
0.3	1.01	0.13	0.85	1.20	34.20	1.92
0.4	0.93	0.15	0.76	1.17	34.20	1.92
0.5	0.90	0.10	0.81	1.06	33.60	3.65
0.6	0.89	0.14	0.76	1.11	34.00	2.24
0.7	0.92	0.13	0.81	1.14	34.00	3.08
0.8	0.84	0.12	0.76	1.06	34.00	2.24
0.9	0.82	0.14	0.72	1.06	34.80	2.28
1.0	0.86	0.12	0.71	1.05	34.40	2.19
1.1	0.84	0.12	0.71	1.04	34.40	2.30
1.2	0.82	0.16	0.72	1.10	34.40	1.82
1.3	0.82	0.10	0.76	1.00	36.00	1.58
1.4	0.83	0.14	0.69	1.07	35.00	2.55
1.5	0.81	0.13	0.69	1.01	36.40	3.13
SAC	5.27	0.87	4.09	6.44	100.00	35.09

Table E.20: Test performance and convergence statistics for the **FetchReach environment** with different conditions. Performance metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations.

Condition	Cost Mean	Cost Std.	Cost Min	Cost Max	Convergence Epoch Mean	Convergence Epoch Std.
0.1	3.92	0.41	3.45	4.50	60.60	21.41
0.2	2.78	0.62	2.18	3.79	65.80	10.69
0.3	2.34	0.40	1.73	2.72	87.60	37.31
0.4	2.99	1.36	1.98	4.77	83.00	28.50
0.5	3.05	2.06	1.24	6.42	81.00	24.83
0.6	4.54	2.95	1.85	8.96	76.80	22.62
0.7	2.72	1.31	1.77	4.59	97.80	15.87
0.8	3.08	0.47	2.27	3.40	98.20	35.05
0.9	2.17	0.47	1.42	2.70	91.60	6.84
1.0	2.74	0.74	1.98	3.70	84.80	33.45
1.1	3.30	0.73	2.35	4.28	77.60	22.35
1.2	2.29	0.52	1.85	3.11	102.80	27.37
1.3	4.03	1.42	2.26	5.97	101.80	16.78
1.4	2.19	0.31	2.02	2.75	91.80	39.71
1.5	3.43	1.26	2.38	5.18	85.20	32.41
SAC	5.27	0.87	4.09	6.44	100.00	35.09

Table E.21: Test performance and convergence statistics for the **FetchReach (infinite horizon) environment** with different conditions. Performance metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations.

E.2.1.6. Extra SAC Critic Network Experiment

Critic Network f_ϕ	Cost Mean	Cost Std.	Cost Min	Cost Max	Convergence Epoch Mean	Convergence Epoch Std.
[256, 256]	36.38	12.91	25.24	56.19	451.75	34.70
[256, 256, 16]	28.00	2.48	23.67	29.78	376.60	143.67
[64, 64, 16]	41.08	18.12	30.97	73.36	392.75	62.66

Table E.22: Test performance and convergence statistics for the **CartPole environment** with different critic network architectures. Performance metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations.

Critic Network f_ϕ	Cost Mean	Cost Std.	Cost Min	Cost Max	Convergence Epoch Mean	Convergence Epoch Std.
[256, 256]	517.63	357.86	214.35	1125.49	28.00	11.36
[256, 256, 16]	228.07	164.72	141.60	521.44	24.75	5.68

Table E.23: Test performance and convergence statistics for the **GRN environment** with different critic network architectures. Performance metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations.

Critic Network f_ϕ	Cost Mean	Cost Std.	Cost Min	Cost Max	Convergence Epoch Mean	Convergence Epoch Std.
[256, 256]	3684.69	3357.31	1064.39	8684.41	54.00	29.07
[256, 256, 16]	652.60	507.48	190.60	1508.33	36.80	13.37

Table E.24: Test performance and convergence statistics for the **CompGRN environment** with different critic network architectures. Performance metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations.

Critic Network f_ϕ	Cost Mean	Cost Std.	Cost Min	Cost Max	Convergence Epoch Mean	Convergence Epoch Std.
[256, 256]	5.27	0.87	4.09	6.44	100.00	35.09
[256, 256, 16]	13.98	5.42	8.30	22.55	126.60	13.11
[64, 64, 16]	4.93	1.22	3.53	6.68	100.80	11.05

Table E.25: Test performance and convergence statistics for the **FetchReach environment** with different critic network architectures. Performance metrics are calculated from the last 10 epochs to ensure stability. Convergence Epoch is defined when mean cost reaches 95% of its maximum, with a 5-epoch moving average applied to smooth short-term fluctuations.