Research on Deep Reinforcement Learning for Path Following of Nonholonomic Wheeled Mobile Robots

Doctoral Dissertation

Submitted by:

Yu Cao

Under the supervision of:
Prof. Seiji Hashimoto

Division of Electronics and Informatics
School of Science and Technology
Gunma University

February 2024

Acknowledgement

Engaging in research is a collaborative journey, and my sincere appreciation goes to my supervisor, Prof. Seiji Hashimoto. His guidance and insights have been instrumental in propelling me to my current stage of achievement. Beyond his academic prowess, Prof. Hashimoto's ability to lighten the academic atmosphere with a touch of humor and understanding of his students has not only made the research process more enjoyable but has also created an environment conducive to creativity and learning. I would like to express my gratitude to Assistant Professor. Takahiro Kawaguchi for his meticulous care and unwavering support towards students in the research lab, including myself. His attentiveness has made it possible for us to engage in research more effortlessly.

I would also like to acknowledge and express my appreciation for the contributions of my dissertation committee members: Prof. Nobuaki Nakazawa, Associate Prof. Toshiki Takahashi, Associate Prof. Akihiro Takita, and specially appointed Associate Prof. Yoichi Shiraishi. Their constructive suggestions and generous support have been indispensable to the evolution of my research.

Additionally, heartfelt thanks go out to my fellow graduate students, whose thought-provoking discussions have provided invaluable perspectives. Last but certainly not least, I extend my deepest appreciation to my family for their enduring love and support, which has been my anchor throughout this challenging expedition.

Abstract

Autonomous mobile robots have become integral to the daily lives, providing crucial services across diverse domains. This thesis focuses on path following, a fundamental technology and critical element in achieving autonomous mobility. The challenge of following a predefined path has long been a focal point in the control engineering community. Existing methods predominantly address tracking through steering control, neglecting velocity control or relying on path-specific reference velocities, thereby constraining their generality.

In recent years, Reinforcement Learning (RL), particularly when coupled with Deep Learning (DL) as Deep Reinforcement Learning (DRL), has become a promising field. This combination enables learning for optimized decision-making by exploring the environment and receiving evaluative feedback based on performance. In this thesis, a novel approach that integrates the conventional pure pursuit algorithm with DRL for a nonholonomic wheeled mobile robot has been proposed. The proposed methodology employs pure pursuit for steering control and utilizes the Soft Actor-Critic (SAC) algorithm to train a velocity control strategy within randomly generated path environments.

Through simulation and experimental validation, the proposed approach exhibits notable advancements in path convergence and adaptive velocity adjustments to accommodate paths with varying curvatures. Furthermore, this method holds the potential for broader applicability to vehicles adhering to nonholonomic constraints beyond the specific model examined in this thesis. In summary, this study contributes to the progression of autonomous mobility by harmonizing conventional algorithms with cutting-edge DRL techniques, enhancing the robustness of path following.

要旨

自律移動ロボットは，日常生活において不可欠な存在となり，様々な領域で重要なサービスを提供している． 本研究では，自律移動を実現するための基本技術であり，かつ重要な要素である経路追従に焦点を当てている． あらかじめ定義された経路に従うという課題は，長い間制御工学のコミュニティで焦点となっている． 既存の方法は主に，経路固有の参照速度に依存した操舵制御を介した経路追従が一般的であり，ここでは速度制御が考慮されておらず，そのため一般性が制限されている．

一方，近年では，特に深層強化学習としてディープラーニングと組み合わせた強化学習が有望な分野となっている． この組み合わせにより，環境を探索し，性能に基づく評価フィードバックを受けながら最適な意思決定を学習することが可能である． 本研究では，従来のピュアパーシュートアルゴリズムを深層強化学習と統合する新しいアプローチを提案している． 提案法では，操舵制御にはピュアパーシュートを使用し，ランダムに生成された経路環境内で速度制御ポリシーを学習するためにSoft Actor-Criticアルゴリズムを利用している．

提案されたアプローチは，経路の収束と変化する曲率を考慮した適応的な速度調整において有効性を示すことをシミュレーションと実験により示す． さらに，この手法は，本研究で検証されたモデル以外の，非ホロノミック制約に従う車両にも広範囲に適用可能である． 以上，本研究は伝統的なアルゴリズムを最先端の深層強化学習技術と調和させ，経路追従のロバストネスを向上させることで，自律移動の進化に貢献している．

Contents

List of Figures

List of Tables

Acronyms

RL Reinforcement Learning

DL Deep Learning

DRL Deep Reinforcement Learning

SAC Soft Actor-Critic

MPC Model Predictive Control

DDPG Deep Deterministic Policy Gradient

PPO Proximal Policy Optimization

ANN Artificial Neural Network

TD Temporal Difference

ReLU Rectified Linear Unit

SGD Stochastic Gradient Descent

MLP Multi-Layer Perceptron

RNN Recurrent Neural Network

CNN Convolutional Neural Network

MDP Markov Decision Process

DP Dynamic Programming

MC Monte Carlo

DNN Deep Neural Network

DQN Deep Q-Network

PG Policy Gradient

MERL Maximum Entropy Reinforcement Learning

ONNX Open Neural Network Exchange

FLOPs Floating-Point Operations

LiDAR Laser imaging, Detection, and Ranging

ROS Robot Operating System

SSH Secure Shell Protocol

LAN Local Area Network

SLAM Simultaneous Localization and Mapping

RBPF Rao-Blackwellized Particle Filters

AMCL Adaptive Monte Carlo Localization

CHAPTER 1

Introduction

1.1 Background and motivation

Autonomous mobile robots have garnered increasing attention in recent years due to their growing applications across various sectors, including companies, industries, hospitals, schools, agriculture, and households. They contribute to improving service levels and the efficiency of daily activities. The demand for mobile robots is on the rise thanks to technological advancements, enabling them to perform tasks such as carrying heavy objects, monitoring, and engaging in search and rescue missions [1], as illustrated in Figure 1.1.

While realizing the complete potential of autonomous driving necessitates the integration of various technologies such as localization, perception, and planning, a foundational technical requirement involves ensuring the vehicle's capability to stabilize at a designated point or dynamically track reference signals and trajectories. Path following has been extensively explored as an alternative to trajectory tracking in different types of vehicles, owing to its relative flexibility with respect to time constraints [2, 3]. In trajectory tracking problems, the reference trajectory determines both when and where the vehicle should be in the state space. In contrast, the objective of path following is to maintain the vehicle's proximity to a predefined geometric path and guide its movement along it, without necessitating preassigned time information [4, 5].

Pure pursuit stands out as one of the earliest strategies suggested for path following [6, 7]. Its simplicity has contributed to its widespread adoption in applications requiring real-time control. In cases where the reference path lacks curvature, and the vehicle maintains a constant speed, the pure pursuit controller operates by fitting a circle through the current

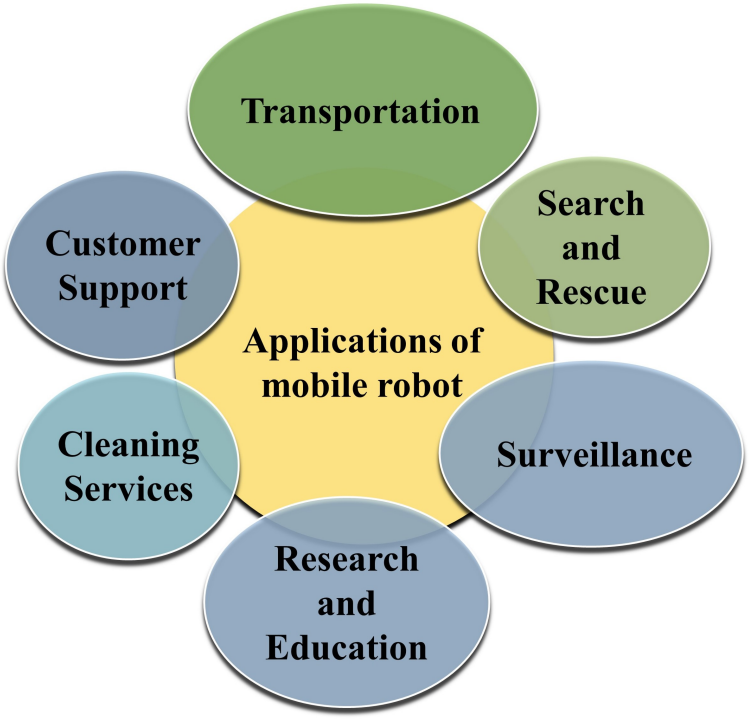

Figure 1.1: Applications of mobile robot [1].

configuration of the vehicle to a point on the path located ahead, utilizing a designated look-ahead distance denoted as L [2, 8]. Methods based on feedback control [9] focus on the error at the current moment, indicating that intervention only occurs after an error has occurred. Model Predictive Control (MPC) is an anticipatory control method that utilizes the mathematical model of a system to predict its future behavior. It determines control inputs by solving a constrained optimization problem. This approach is capable of ensuring control laws for both speed and steering simultaneously [4, 10]. However, the control performance heavily relies on the accuracy of the model. Additionally, due to the necessity of solving optimization problems at each sampling time, real-time computational costs need to be taken into consideration.

RL is a model-free learning approach wherein an agent acquires optimal decisions through interactions with its environment. In recent years, RL has achieved notable successes in various domains, gaining considerable attention and widespread recognition, particularly in fields like gaming and robotics. In scenarios involving discrete action spaces, RL-based approaches have showcased their capability to acquire proficiency in mastering Atari games [11, 12] and surpassing world champions in the game of Go [13]. On the other hand, when confronted with continuous action spaces, RL techniques have effectively tackled a myriad of intricately

nonlinear physics challenges. These include tasks such as robotic manipulation, bipedal locomotion, and game dynamics [14].

1.2 Related works

Since the adoption of Deep Neural Network (DNN) for approximating the action-value function [11, 12], also known as the Q-function, RL has demonstrated the ability to effectively handle high-dimensional state spaces. In contemporary DRL, methods can be categorized into deterministic and stochastic based on the output approach of the policy. Deterministic methods, like the Deep Deterministic Policy Gradient (DDPG) algorithm [15], have a policy network that directly outputs actions. On the other hand, stochastic methods, such as the Proximal Policy Optimization (PPO) algorithm [16], output a probability density for actions and sample actions from this distribution.

In recent years, a growing number of researchers have explored the application of DRL methods to address the challenge of path following. Zhang et al. [17] proposed an enhanced DDPG algorithm for addressing the path following control problem of Unmanned Aerial Vehicles using a kinematics model. In their proposal, DDPG is solely responsible for steering control. Rubí et al. [18] introduced three sequentially improved methods grounded in DDPG for achieving path following and adaptive velocity control of a quadrotor. Fine-tuning an agent, initially trained in a simulation environment, through experimentation poses challenges. To ensure stable performance of the agent's output in their experiments, they employed a correction constant. Meyer et al.. [19] trained an underactuated autonomous surface vehicle for path following based on PPO, where one of their metrics is the success rate, indicating whether the vehicle reached the goal. In a more recent study, Ma et al. [20] introduced a path-following control scheme utilizing a sample-observed SAC for an underwater vehicle. In their approach, only experiences with low values of cross-track error were used during training, showcasing successful path tracking.

Despite the success RL has found in path following, the current literature exhibits notable shortcomings. The majority of articles rely on the vehicle's kinematics model and lack experimental validation. Although simulations provide abundant data and improve agent training safety, a performance gap persists when transitioning models to real robots [21]. Most articles concentrate on steering control, opting for either constant velocity settings or encouraging velocity to follow a manually set reference value on path points using a reward function. This approach significantly constrains the exploration space for velocity control.

1.3 Outline of dissertation

The dissertation is divided into five main chapters. In Chapter 2, important concepts and notations utilized in the dissertation are introduced. An overview of Artificial Neural Network (ANN), especially DL, is provided. Additionally, RL is thoroughly introduced, covering the main deep RL methods, namely value-based, policy-based, and actor-critic methods.

Chapter 3 addresses the problem of path-following, exploring fundamental kinematics concepts and introducing the pure pursuit algorithm adapted for wheeled mobile robots.

Chapter 4 enhances the conventional RL framework by incorporating entropy regularization. This addition emphasizes exploratory behavior during optimal policy determination, with a specific implementation of the SAC algorithm. Building on the groundwork laid in Chapter 3, the design of a path-following controller is discussed in detail.

Chapter 5 presents the main results derived from applying SAC to path following, showcasing outcomes from both simulations and experiments. This chapter also discusses the disparities between simulated and real-world environments and proposes solutions. The focus is on the adaptive adjustment of velocity to reduce lateral deviation.

The final chapter summarizes the findings and limitations of this dissertation. It also includes a discussion of future work, outlining potential research directions and areas for improvement.

References

[1] M. B. Alatise and G. P. Hancke, "A review on challenges of autonomous mobile robot and sensor fusion methods," *IEEE Access*, vol. 8, pp. 39830–39846, 2020.

[2] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," *IEEE Transactions on intelligent vehicles*, vol. 1, no. 1, pp. 33–55, 2016.

[3] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda, "A survey of autonomous driving: Common practices and emerging technologies," *IEEE access*, vol. 8, pp. 58443–58469, 2020.

[4] T. Faulwasser, B. Kern, and R. Findeisen, "Model predictive path-following for constrained nonlinear systems," in *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*, pp. 8642–8647, IEEE, 2009.

[5] A. P. Aguiar, J. P. Hespanha, and P. V. Kokotović, "Performance limitations in reference tracking and path following for nonlinear systems," *Automatica*, vol. 44, no. 3, pp. 598–610, 2008.

[6] R. C. Coulter *et al.*, *Implementation of the pure pursuit path tracking algorithm*. Carnegie Mellon University, The Robotics Institute, 1992.

[7] O. Amidi and C. E. Thorpe, "Integrated mobile robot control," in *Mobile Robots V*, vol. 1388, pp. 504–523, SPIE, 1991.

[8] N. H. Amer, H. Zamzuri, K. Hudha, and Z. A. Kadir, "Modelling and control strategies in path tracking control for autonomous ground vehicles: a review of state of the art and challenges," *Journal of intelligent & robotic systems*, vol. 86, pp. 225–254, 2017.

[9] C. Samson, "Path following and time-varying feedback stabilization of a wheeled mobile robot," *Second International Conference on Automation, Robotics and Computer Vision*, vol. 3, 1992.

[10] T. Faulwasser and R. Findeisen, "Nonlinear model predictive control for constrained output path following," *IEEE Transactions on Automatic Control*, vol. 61, no. 4, pp. 1026–1039, 2015.

[11] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[13] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[14] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and

K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, pp. 1928–1937, PMLR, 2016.

[15] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

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

[17] Y. Zhang, Y. Zhang, and Z. Yu, "Path following control for uav using deep reinforcement learning approach," *Guidance, Navigation and Control*, vol. 1, no. 01, p. 2150005, 2021.

[18] B. Rubí, B. Morcego, and R. Pérez, "Deep reinforcement learning for quadrotor path following with adaptive velocity," *Autonomous Robots*, vol. 45, no. 1, pp. 119–134, 2021.

[19] E. Meyer, H. Robinson, A. Rasheed, and O. San, "Taming an autonomous surface vehicle for path following and collision avoidance using deep reinforcement learning," *IEEE Access*, vol. 8, pp. 41466–41481, 2020.

[20] R. Ma, Y. Wang, S. Wang, L. Cheng, R. Wang, and M. Tan, "Sample-observed soft actor-critic learning for path following of a biomimetic underwater vehicle," *IEEE Transactions on Automation Science and Engineering*, pp. 1–10, 2023.

[21] W. Zhao, J. P. Queralta, and T. Westerlund, "Sim-to-real transfer in deep reinforcement learning for robotics: a survey," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 737–744, 2020.

CHAPTER 2

Background and theory

This chapter offers a comprehensive overview of essential fundamentals to foster a deeper understanding of the dissertation topic. Section 2.1 introduces basic concepts of ANN, offering a glimpse into DL. In Section 2.2, the fundamentals of traditional RL are covered, including basic algorithms such as Temporal Difference (TD) learning. The integration of knowledge from Sections 2.1 and 2.2 is presented in Section 2.3, which focuses on DRL. This section delves further into the theoretical background closely aligned with the dissertation approach.

2.1 Artificial neural networks

The artificial neuron draws inspiration from the biological neurons in the human brain, totaling around 86 billion, processing sensory information and forming connections on the order of 10^{14} to 10^{15}. A biological neuron features numerous branching extensions called dendrites and a lengthy extension known as the axon. The axon exhibits variable lengths, ultimately splitting into branches.Input signals are collected by each neuron through its dendrites, and if a certain action potential is reached, the neuron fires through its axon, forwarding the output to connected neurons. The axon further extends, branching out to establish connections with the dendrites of other neurons through synapses [1, 2].

The artificial neuron simplifies this process by having k input connections and processing inputs through a weighted sum, adding a bias b, and applying an activation function:

$$f\left(\sum_{i=1}^{k}(w_i x_i) + b\right)$$

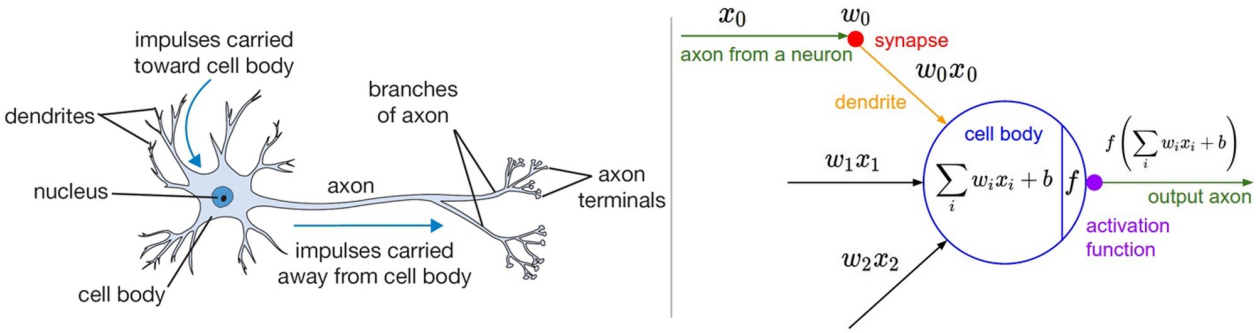

Figure 2.1: A biological neuron (left) and an artificial neuron (right) [3].

where the process is illustrated in Figure 2.1, highlighting the parallels between biological and artificial neurons.

The firing rate of the neuron is characterized by an activation function, denoted as $f(\cdot)$, which signifies the frequency of the spikes along the axon. Typical activation functions include the following; both the functions themselves and their derivatives are represented in Figure 2.2.

- **Step function**. In general, it is commonly used in the naive perceptron, that is, a single-layer neural network, to represent logical circuits such as AND gates and OR gates.

$$f(u) = \begin{cases} 0 & \text{if } u < 0 \\ 1 & \text{if } u \geq 0 \end{cases} \tag{2.1.1}$$

- **Sigmoid/Logistic function**. A historically prevalent choice for an activation function that takes a real-valued input and squashes it to a range between 0 and 1. It is crucial to acknowledge that the output is not centered around zero.

$$f(u) = \frac{1}{1 + e^{-u}} \tag{2.1.2}$$

- **Hyperbolic tangent function**. Similar to the Sigmoid function, despite mapping input values to a range between -1 and 1, the output is zero-centered.

$$f(u) = \frac{e^{u} - e^{-u}}{e^{u} + e^{-u}} \tag{2.1.3}$$

- **Rectified Linear Unit (ReLU) function**. A more widely used activation function in recent years. Due to the absence of exponential operations, it is computationally more

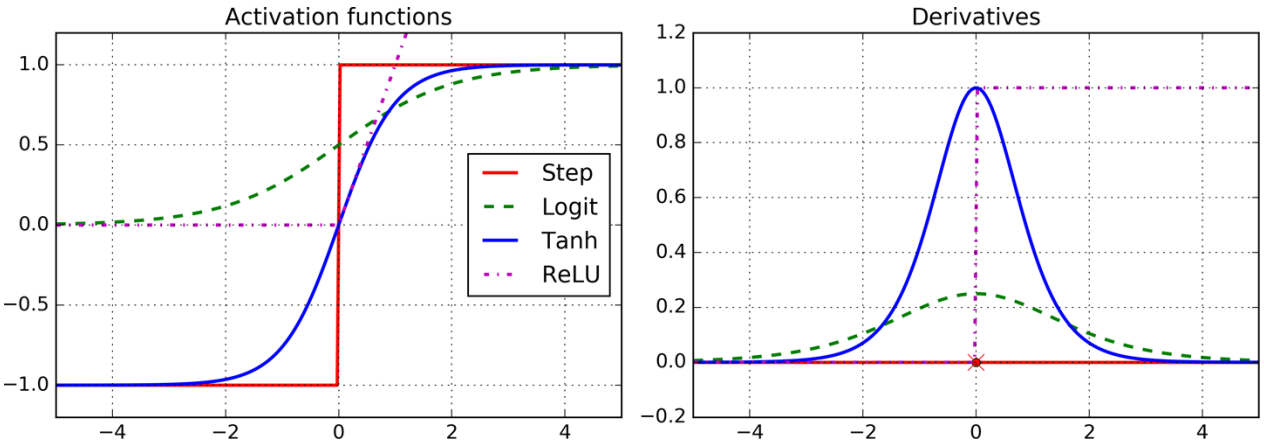

Figure 2.2: Comparison of activation functions and their derivatives [4].

efficient.

$$f(u) = \begin{cases} u & \text{if } u > 0 \\ 0 & \text{if } u \leq 0 \end{cases} \tag{2.1.4}$$

Non-linear activation functions play a vital role in neural networks. In essence, linear functions should be avoided as activation functions [5, 6]. The rationale behind this lies in the fact that utilizing a linear function would render the neural network equivalent to a single-layer network, irrespective of the depth of the network. For example, for a linear function $h(u) = cu$, $y(u) = h(h(h(u)))$ corresponds to a three-layer neural network. In such a scenario, the operation can be represented equivalently by a single-layer network $y(u) = c^3u$.

ANNs are conceptualized as assemblies of neurons interconnected in an acyclic graph, wherein the outputs of certain neurons can serve as inputs to others. It is imperative to avoid cycles in this connectivity to prevent the occurrence of infinite loops during the forward pass of a network [7, 8]. The predominant layer type is the fully-connected layer, where neurons in two consecutive layers are fully and pairwise connected. However, within a single layer, neurons do not share connections. Figure 2.3 provides an illustration of neural network topologies employing a series of fully-connected layers.

2.1.1 Learning process

The distinctive feature of neural networks is their ability to learn from data. Learning, in this context, refers to the process of automatically acquiring the optimal weight parameters from training data. Let $\mathbf{w}$ represent the set of weights of all neurons in the neural network. The objective of the learning process is to optimize a parameter set $\mathbf{w}$ that leads to the most

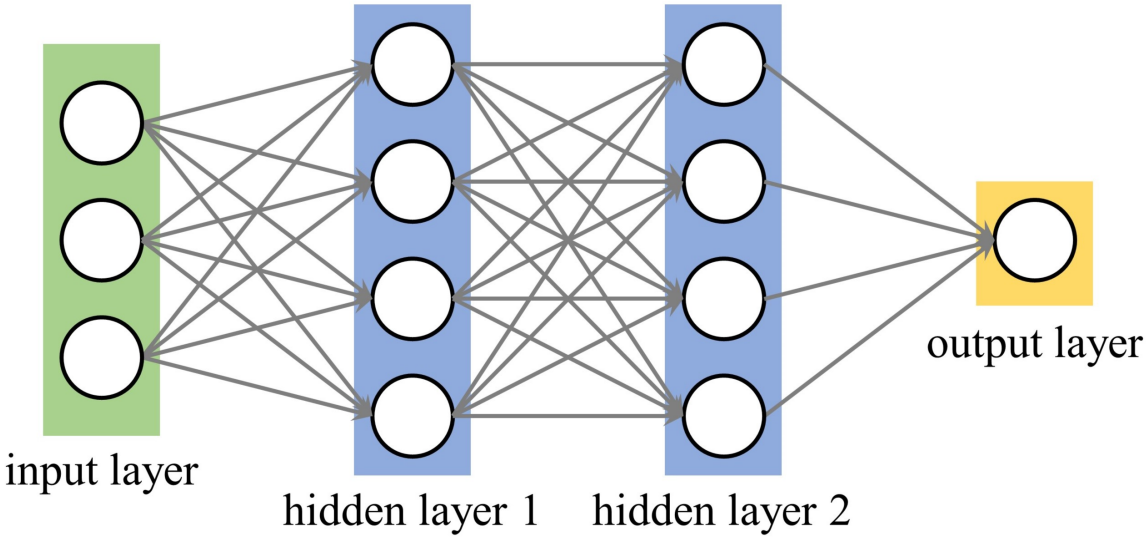

Figure 2.3: A fully-connected, feedforward neural network with a singular input layer, dual hidden layers, and a sole output layer.

effective function approximation. In supervised learning, the true output T for a given input X is provided and utilized to adjust the parameters. This process is iterative and involves the following steps [9, 10].

1. **Forward propagation**. The input X is propagated through the network, resulting in the predicted output $\hat{Y} = f_{\mathbf{w}}(X)$.

2. **Loss computation**. Loss is employed to reflect the current learning status and seek the optimal parameters, typically measured through mean squared error or cross-entropy error.

   - Mean squared error
   $$L(\mathbf{w}) = \frac{1}{k}\sum_{i}^{k}(\hat{Y}_i - T_i)^2 \tag{2.1.5}$$

   - Cross-entropy error
   $$L(\mathbf{w}) = -\sum_{i}^{k} T_i \log \hat{Y}_i \tag{2.1.6}$$

   In practical applications, the correct labels are often represented in a one-hot encoding [11], where only the index corresponding to the correct solution label is 1, and the rest are 0. It can be observed that mean squared error is influenced by each predicted value, while the result of cross-entropy error is determined solely by the prediction

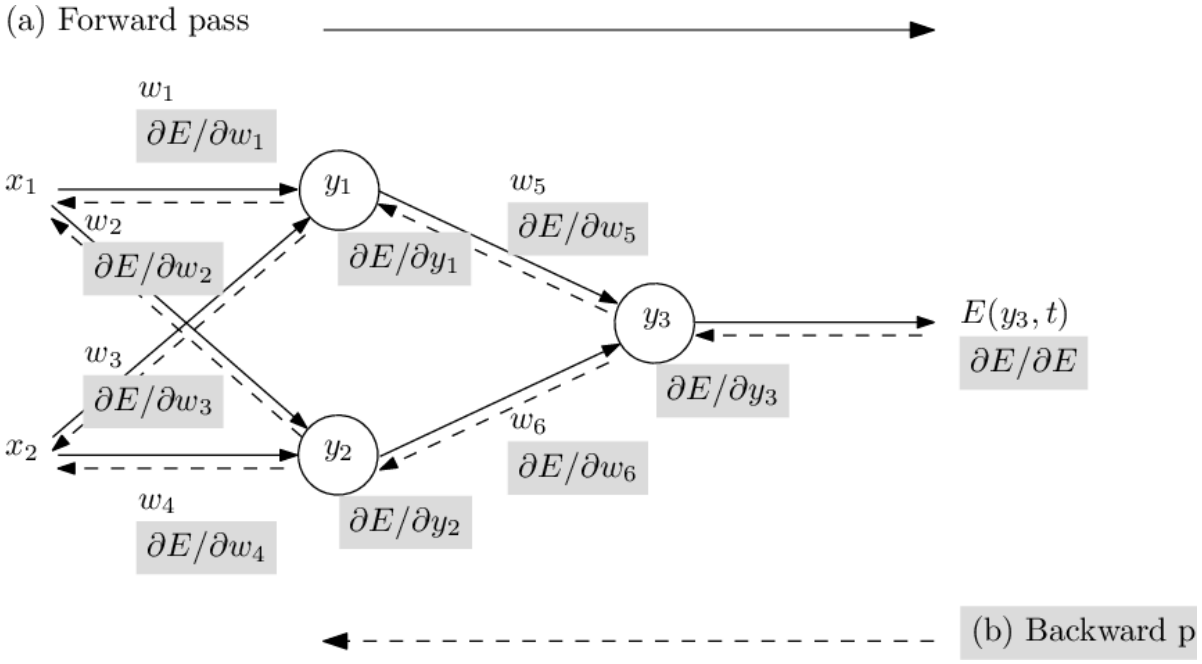

Figure 2.4: An overview of backpropagation [13].

corresponding to the correct solution.

3. **Backpropagation**. Backpropagation [12] is an efficient method for computing gradients of weight parameters, utilizing the chain rule to provide local gradients for each hidden neuron. For instance, given the functions $y = g(x)$ and $z = f(g(x))$, the derivative $\frac{\partial z}{\partial x}$ can be calculated as:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x} \tag{2.1.7}$$

Here, the chain rule enables us to systematically compute gradients layer by layer, allowing the calculated gradients to propagate to the preceding layers, facilitating the effective adjustment of weight parameters in a neural network. Computational graphs provide a more intuitive understanding of error backpropagation. An overview is illustrated in Figure 2.4.

4. **Update**. Neural network parameter optimization is challenging due to the complexity of the parameter space, making it difficult to easily find the optimal solution. The process of utilizing gradients of the parameters, updating them along the opposite of gradient direction, and repeatedly iterating to approach the optimal parameters is known as Stochastic Gradient Descent (SGD) [14]. This is a simple and commonly used method,

and Equation (2.1.8) illustrates the corresponding update rule.

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} \tag{2.1.8}$$

where α represents the learning rate, a hyperparameter that requires tuning.

2.1.2 Mini-batch learning

When learning from training data, the goal is to calculate the value of the loss function specifically for the training data. Therefore, all training data is considered. However, computing the sum of the loss function over all data can be time-consuming. Moreover, when dealing with large datasets, such as millions or tens of millions of samples, using the entire dataset becomes impractical. Hence, a subset is selected from the entire dataset to serve as an approximation [15]. This approach is also applicable to neural network learning, where a batch of data is chosen from the training data in each iteration, known as a mini-batch, and learning is performed on each mini-batch. This method is referred to as mini-batch learning.

Taking the mean squared error formula (Equation (2.1.5)) as an example, the error for a mini-batch takes the following form:

$$L(\mathbf{w}) = \frac{1}{n}\frac{1}{k}\sum_{i}^{n}\sum_{j}^{k}(\hat{Y}_{ij} - T_{ij})^2 \tag{2.1.9}$$

where N represents the number of data samples in the batch, and the subscript ij indicates the j-th element of the i-th data sample. This computes the average loss for a single sample, providing a unified standard independent of the quantity of training data.

2.1.3 Deep learning

A deep feedforward network, also known as a feedforward neural network or a Multi-Layer Perceptron (MLP), is a typical DL model. It is called feedforward because information flows from the input through intermediate hidden layers to the output, with no feedback connections directly linking the model's output to itself. Introducing feedback connections transforms the model into a Recurrent Neural Network (RNN), designed for handling sequential data, including dynamic and time-related information. RNNs store information from preceding passes in an internal hidden state, subsequently leveraging this stored information in subsequent passes [16].

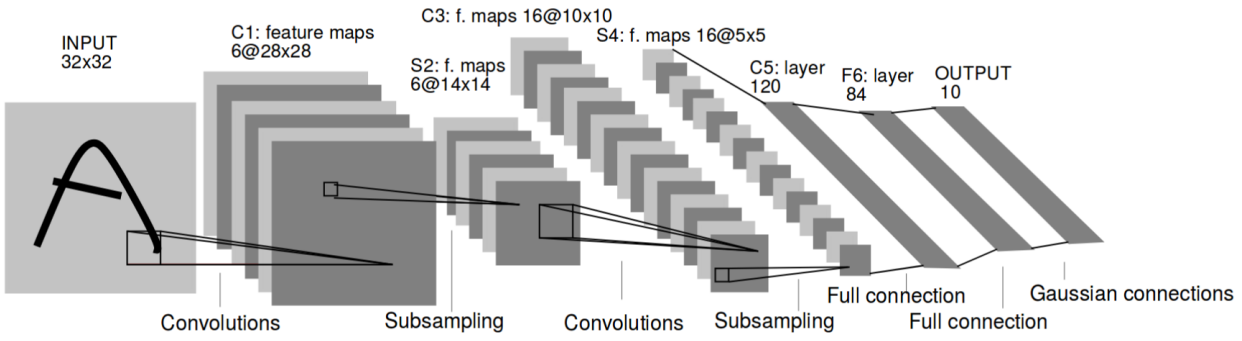

Figure 2.5: Architecture of LeNet-5, a CNN used for digits recognition [17].

In addition to fully connected layers, commonly found in ANNs, DL also encompasses various methods for creating network layers tailored to specific types of data. One such method is CNN, which incorporates convolution layers to process data with spatial connectivity. CNN was originally proposed as a method for classifying handwritten numbers from images [17], as illustrated in Figure 2.5 [17], and the advent of AlexNet [18] sparked the upsurge in the popularity of DL.

2.2 Reinforcement learning

RL addresses the challenge of enabling a decision-making agent to learn optimal actions within an environment, maximizing cumulative rewards. This problem is framed as a MDP, where the agent iteratively interacts with the environment over discrete time steps [19]. The agent's task is to continually refine its actions by discerning which yield favorable rewards.

In RL, the agent learns through trial-and-error, accumulating experiences as it engages with the environment. Figure 2.6 (left) illustrates the core concept: At time t, given the state $s_t \in \mathcal{S}$, the agent selects action $a_t \in \mathcal{A}$, resulting in the receipt of reward r_t, and subsequently transitions to the next state s_{t+1}. This interactive process between the agent and the environment effectively yields a series of transitions, namely trajectory:

$$s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \ldots$$

In the field of RL, there are two conventions regarding the timing of rewards, namely, representing the reward as r_t as shown on the left side of Figure 2.6, or as r_{t+1} as illustrated on the right side of Figure 2.6.

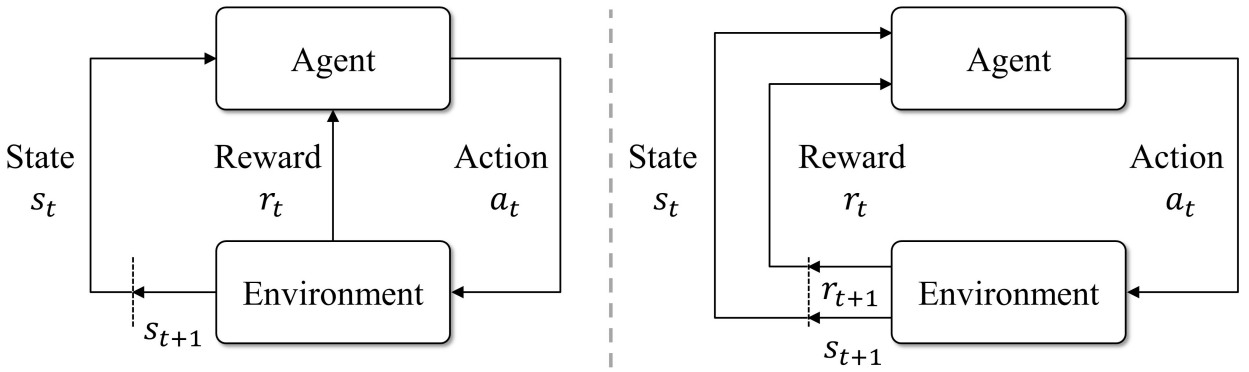

Figure 2.6: General concept of MDP.

- Given state s_t, taking action a_t, receiving reward r_t, and transitioning to the next state s_{t+1}.
- Given state s_t, taking action a_t, transitioning to the next state s_{t+1}, and receiving reward r_{t+1}.

As shown above, there are two approaches based on the timing of rewards. In this section, the former to elucidate the MDP is adopted.

2.2.1 Markov decision process

An MDP is characterized by a sequential decision process that is fully observable, operates in a stochastic environment, and possesses a transition model adhering to the Markov property [20]. The Markov property refers to a stochastic process where future developments depend exclusively on the current state, irrespective of the past state sequence. In other words, this property imparts a memory-less characteristic to the process, as the evolution of the system is influenced only by the current state. This property simplifies the representation of states, facilitating the modeling and solving of RL problems. The MDP can be concisely represented as a tuple $(\mathcal{S}, \mathcal{A}, p, r)$, where:

- $\mathcal{S}$ represents the set of all states called the state space.
- $\mathcal{A}$ denotes the actions available to the agent called the action space. Alternatively, the available actions are dependent on the current state, denoted as $\mathcal{A}(s)$ for $s \in \mathcal{S}$.
- $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ denotes the transition probability $p(s'|s, a)$, representing the likelihood that action a in state s will result in state s'. In scenarios where the environment is deterministic, the state transition can be expressed as a function

$s' = f(s, a)$.

- $r : \mathcal{S} \times \mathcal{A} \to [r_{min}, r_{max}]$ represents the bounded reward received after transitioning from state s to state s' as a result of action a.

The objective in an MDP is to determine a policy π, which models a probability distribution $\pi(a|s)$ over actions for each state, aiming to maximize a cumulative function of the rewards. Typically, this function is the expected discounted sum over a potentially infinite horizon, referred to as the return g_t in Equation (2.2.1).

$$g_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{i=t}^{\infty} \gamma^{i-t} r(s_i, a_i) \tag{2.2.1}$$

where $\gamma \in [0, 1]$ is the discount rate, causing rewards to exponentially decay over time. The introduction of the discount rate is primarily to prevent the return from becoming infinitely large and diverging in continuous tasks. Moreover, due to the discount rate, it becomes apparent that rewards closer to the present hold higher importance. Notably, with a slight rearrangement, it can be readily obtained $g_t = r_t + \gamma g_{t+1}$, indicating the relationship between returns at adjacent time steps.

Then, the expected return, also known as the state-value function $v_\pi(s)$, can be defined by the following expression:

$$v_\pi(s) = \mathbb{E}_\pi[g_t | s_t = s] \tag{2.2.2}$$

where the condition for the state-value function includes both the state s_t being s and the policy being π, as changes in the policy can lead to variations in the obtained rewards.

2.2.2 Bellman equation

Bellman equation is the most crucial equation in MDP and provides a fundamental basis for many RL algorithms. It provides the value of a decision problem in a specific state by taking into account the rewards from past decisions and the value of the remaining decision problem after those decisions have been made [21]. Utilizing the relationship $g_t = r_t + \gamma g_{t+1}$ and the linearity property of expectations, the value function for state s can be re-expressed by Equation (2.2.3).

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[g_t | s_t = s] \\ &= \mathbb{E}_\pi[r_t + \gamma g_{t+1} | s_t = s] \\ &= \mathbb{E}_\pi[r_t | s_t = s] + \gamma \mathbb{E}_\pi[g_{t+1} | s_t = s] \end{aligned} \tag{2.2.3}$$

The first part represents the expected value of rewards for potential transitions under state s, which can be given as follows.

$$\mathbb{E}_\pi[r_t|s_t = s] = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) r(s,a) \tag{2.2.4}$$

The second part represents the expectation of the next time step's return under current state being s. This is obtained by summing the expected returns from all possible transitions from the current time step to the next step, as illustrated in Equaiotn (2.2.5).

$$\begin{aligned} \mathbb{E}_\pi[g_{t+1}|s_t = s] &= \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) \mathbb{E}_\pi[g_{t+1}|s_{t+1} = s'] \\ &= \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) v_\pi(s') \end{aligned} \tag{2.2.5}$$

Substituting the two aforementioned equations into Equation (2.2.3), a new expression for the value function can be obtained.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[r_t|s_t = s] + \gamma \mathbb{E}_\pi[g_{t+1}|s_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) r(s,a) + \gamma \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) v_\pi(s') \\ &= \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) \left\{ r(s,a) + \gamma v_\pi(s') \right\} \end{aligned} \tag{2.2.6}$$

which is the Bellman equation, representing the recursive relationship between the value function for state s and the value function for the next state s'.

In RL, another crucial function is the action-value function $q_\pi(s,a)$, conventionally referred to as the Q-function. It has a definition similar to the state-value function $v_\pi(s)$ but includes the additional condition that the action at time step t a_t is a:

$$q_\pi(s,a) = \mathbb{E}_\pi[g_t|s_t = s, a_t = a] \tag{2.2.7}$$

which is the expected return of taking action a in state s and thereafter behaving according to the policy π. It is noteworthy that the action a is independent of the policy π; it is arbitrarily chosen. In other words, the Q-function is equivalent to the state-value function if the action a is chosen according to the policy π.

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a) \tag{2.2.8}$$

Following the same steps as those for the state-value function, the Bellman equation based on the Q-function can be derived as follows:

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi[r_t + \gamma g_{t+1} | s_t = s] \\ &= \mathbb{E}_\pi[r_t | s_t = s, a_t = a] + \gamma \mathbb{E}_\pi[g_{t+1} | s_t = s, a_t = a] \\ &= \sum_{s'} p(s'|s,a) r(s,a) + \gamma \sum_{s'} p(s'|s,a) \mathbb{E}_\pi[g_{t+1} | s_{t+1} = s'] \\ &= \sum_{s'} p(s'|s,a) \left\{ r(s,a) + \gamma v_\pi(s') \right\} \end{aligned} \tag{2.2.9}$$

Substituting Equation (2.2.8) into it results in:

$$q_\pi(s,a) = \sum_{s'} p(s'|s,a) \left\{ r(s,a) + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a') \right\} \tag{2.2.10}$$

Since MDP is designed to discover a latent optimal policy $\pi^*(a|s)$ that maximizes the expected return, which means the optimal policy is deterministic. For the state-value function $v_\pi(s)$, it aims to maximize the value function for all states. Therefore, the optimal Bellman equation for the state-value function is:

$$v_\pi^*(s) = \max_a \sum_{s'} p(s'|s,a) \left\{ r(s,a) + \gamma v_\pi^*(s') \right\} \tag{2.2.11}$$

Similarly, for the optimal Q-function with respect to the optimal policy π^* :

$$q_\pi^*(s,a) = \sum_{s'} p(s'|s,a) \left\{ r(s,a) + \gamma \max_{a'} q_\pi^*(s',a') \right\} \tag{2.2.12}$$

where, the next action a' in the computation is the one that maximizes the Q-function, as illustrated in Figure 2.7.

Up to this point, the Bellman equations have been derived, and the intuitive approach is to use these equations to establish a system of simultaneous equations for solving the value function. However, the practical complexity of real-world problems leads to an enormous expansion of the state and action spaces, rendering it impossible to achieve this through direct computation. Furthermore, it necessitates the complete knowledge of the environment's model, a condition that is often unattainable in most cases. In such scenarios, numerical solutions become imperative. Therefore, in this dissertation, the TD method will be employed

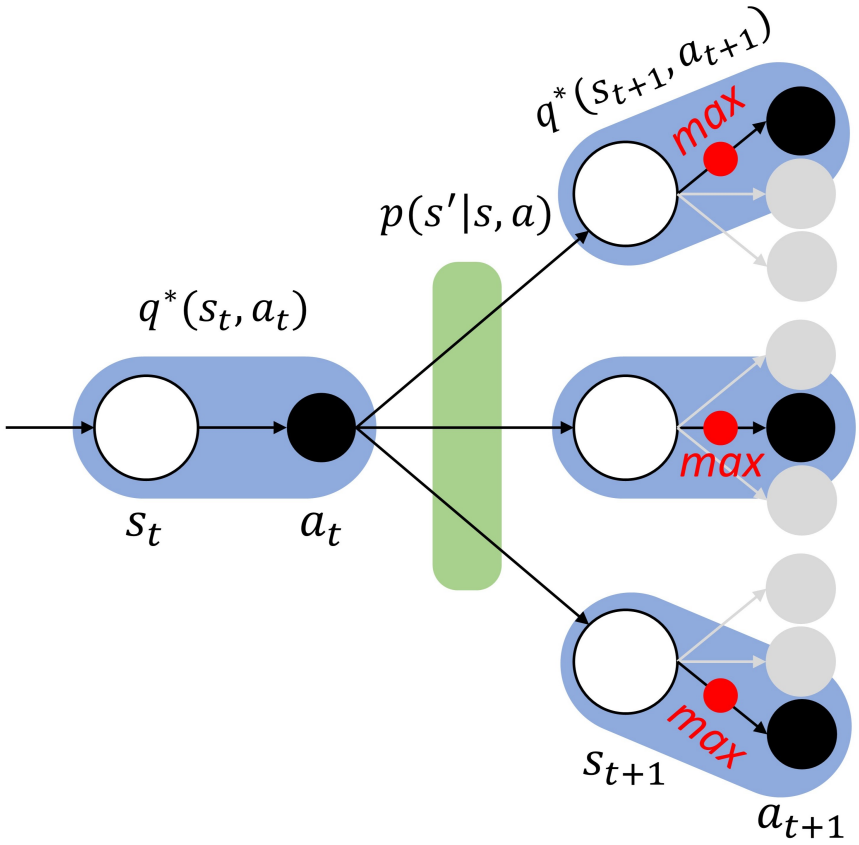

Figure 2.7: Diagram of the optimal Q-function.

to approximate the optimal value function.

2.2.3 Temporal difference learning

The TD method is an approach that doesn't require the use of an environmental model and allows for the updating of the value function after each action. It combines elements of Dynamic Programming (DP) and Monte Carlo (MC) methods. The core idea behind TD learning is to leverage experience for solving prediction problems [22, 23]. This implies that as the agent interacts with the environment, it accumulates experience and utilizes this experiential knowledge to estimate the value function. The re-expression of Equation (2.2.6) in the form of expectations is as follows:

$$\begin{aligned} v_\pi(s) &= \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) \left\{ r(s,a) + \gamma v_\pi(s') \right\} \\ &= \mathbb{E}_\pi[r_t + \gamma v_\pi(s_{t+1}) | s_t = s] \end{aligned} \tag{2.2.13}$$

where the term $r_t+\gamma v_\pi(s_{t+1})$ can be approximated using a mini-batch of samples. Subsequently, the TD update rule is provided in Equation (2.2.14).

$$V_\pi(s_t) \leftarrow V_\pi(s_t) + \alpha \left\{ r_t + \gamma V_\pi(s_{t+1}) - V_\pi(s_t) \right\} \tag{2.2.14}$$

where V_π represents the estimated value function, and uppercase letters are used to distinguish them from the true value function v_π. The target term $r_t + \gamma V_\pi(s_{t+1})$ is the estimated value one step ahead that the current estimate is updating toward. Similarly, the Q-function version is given as:

$$Q_\pi(s_t, a_t) \leftarrow Q_\pi(s_t, a_t) + \alpha \left\{ r_t + \gamma Q_\pi(s_{t+1}, a_{t+1}) - Q_\pi(s_t, a_t) \right\} \tag{2.2.15}$$

where, in both equations, α is a positive constant that adjusts the learning rate. Additionally, the concept of this TD method can be extended to n-step, where n denotes the number of future discounted returns to consider when computing the target value.

As one of the representative TD methods in RL, Q-Learning stands out as a widely adopted off-policy technique. At its essence, the algorithm strives to develop the Q-function serving as an estimator for the expected cumulative reward. This Q-function is conventionally depicted in a table known as the Q-table. The iterative process of the algorithm revolves around updating Q-values, taking into account observed rewards and the maximum Q-value of the subsequent state. The crucial Q-value update rule, is expressed as in Equation (2.2.16) and is graphically depicted in Figure 2.8.

$$Q_\pi(s_t, a_t) \leftarrow Q_\pi(s_t, a_t) + \alpha \left\{ r_t + \gamma \max Q_\pi(s_{t+1}, a) - Q_\pi(s_t, a_t) \right\} \tag{2.2.16}$$

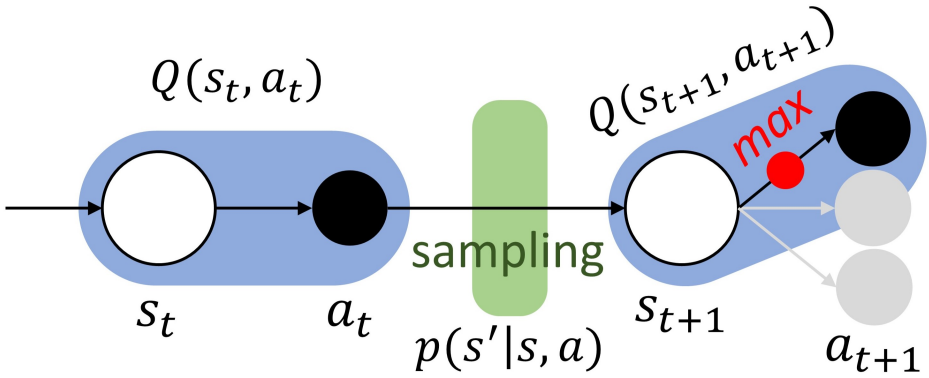

Figure 2.8: Diagram of Q-learning.

2.3 Deep reinforcement learning

Before the advent of DRL, the simplest representation of a policy is through a look-up table with one entry for each state or state-action pair, known as a tabular policy [24, 25]. However, this approach has its limitations, as outlined below:

- **Finite representation**. The table-based representation is limited to a finite number of state or state-action pairs, making it impractical for scenarios with an infinite number of possible states.
- **Infeasibility of visiting all states**. Given the vast number of possible states, it is not feasible to visit and record values for every state or state-action pair. This poses a significant challenge, especially in complex environments.
- **Memory constraints**. The size of the table is constrained by hardware memory limitations. In situations with large state spaces, storing values for all possible pairs becomes impractical and may lead to memory overflow.
- **Lack of knowledge sharing**. The table-based approach does not facilitate the sharing of knowledge about similar states. Each entry in the table is independent, hindering the model's ability to generalize across states that share common features.

To address the limitations mentioned above, a commonly adopted approach is to replace the table with a DNN as a function approximator. The DNN's ability to approximate non-linear functions and extract relevant features from raw inputs allows for generalization across previously unencountered states. DNNs have showcased impressive capabilities, excelling in tasks like defeating the world champion in the game of GO [26] and mastering intricate robotics manipulation tasks [27]. In this section, core technologies with strong relevance to this dissertation will be addressed.

2.3.1 Value-based method

The incorporation of Q-Learning with non-linear function approximation has faced challenges over the years, primarily due to issues related to unstable learning. This is attributed to the use of estimates that have not been properly determined, leading to updates in estimated values. In particular, for expressive function approximation methods such as ANNs, the issue of instability becomes more pronounced.

However, in 2013, the DeepMind group introduced an innovative solution known as the

Deep Q-Network (DQN) [28]. This approach achieved significant success by seamlessly combining model-free, off-policy Q-Learning with DNNs. Addressing the challenge of unstable learning, two key mechanisms were introduced: experience replay and target network. These mechanisms will be explained in more detail below.

- **Experience replay**

  Considering supervised learning as an example, a mini-batch approach is commonly employed, randomly selecting a subset of training data to update the parameters of a neural network. This method aims to prevent bias in the training data, such as exclusively using data from a single label.

  In contrast, in Q-learning, the Q-function is updated using a sequence of trajectories (s_t, a_t, r_t, s_{t+1}). This approach results in strong correlations among the generated experience data, potentially leading to overfitting during parameter learning. To address this issue, the concept of experience replay is introduced. Specifically, this method involves storing trajectories generated by the interaction between the agent and the environment in a buffer. During parameter updates, a random subset of past experience data is selected, mitigating the correlation among the experience data. Moreover, the ability to reuse experience data improves data utilization efficiency. It is illustrated in Figure 2.9.

  Additionally, experience replay is not limited to Q-learning; it has broad applicability in policy-off RL algorithms.

- **Target network**

  In Q-learning, the TD target is defined as $r_t + \gamma \max Q_\pi(s_{t+1}, a)$, and during Q-learning, the values of Q are updated towards this target. Unlike supervised learning, where the target label remains fixed, this target changes with the updates to Q, contributing to the instability in learning. The technique of using a target network involves employing a pair of neural networks with identical structures. While one is regularly updated online, the target network is periodically synchronized with the online network. This approach is employed to mitigate the variability in the TD target.

  However, completely refraining from updating the TD target would render the Q-function unable to learn. In addition to the periodic update approach, an alternative is to opt for continuous updates through a soft update method, wherein only a small fraction of the parameters is updated during each iteration.

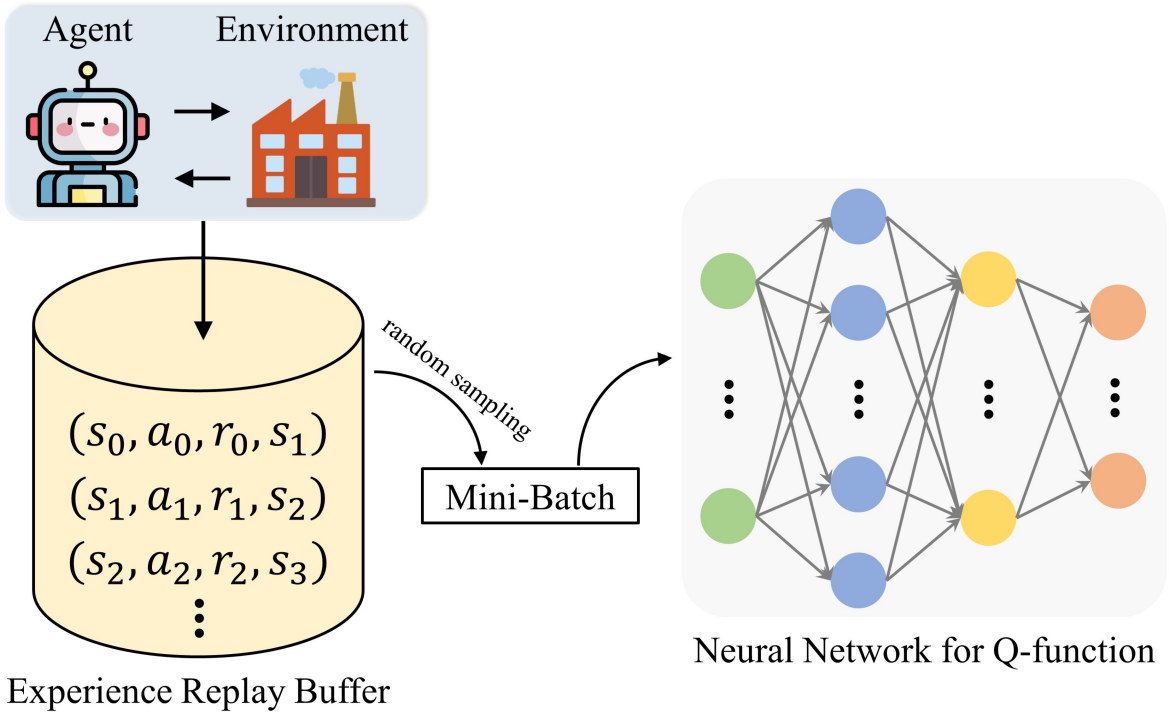

Figure 2.9: The process of Q-learning equipped with experience replay.

2.3.2 Policy-based method

Besides the value-based method, an alternative approach, which does does not rely on a value function but directly represents the policy $\pi(a|s)$, is known as the policy-based method. Among these methods, the most straightforward and pure form involves parameterizing the policy using ANN parameters—specifically, a vector of parameters ϕ. This approach utilizes gradient-based optimization to enhance the policy and is known as the Policy Gradient (PG) method [29, 30].

In the case of a stochastic, parameterized policy denoted as $\pi_\phi(a|s)$, the objective of the PG method is to maximize the expected return.

$$J_\pi(\phi) = \mathbb{E}_{\tau\sim\phi}[g(\tau)] \tag{2.3.1}$$

where $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_{T+1})$ is a finite trajectory. The gradient of $J_\pi(\phi)$ [31, 30] is given by

$$\Delta_\phi J_\pi(\phi) = \mathbb{E}_{\tau\sim\phi}[\sum_{t=0}^{T} g(\tau)\Delta_\phi \log \pi_\phi(a_t|s_t)] \tag{2.3.2}$$

The optimization of the policy is pursued through a gradient ascent approach, for instance,

$$\phi \leftarrow \phi + \alpha \Delta_\phi J_\pi(\phi) \tag{2.3.3}$$

where the gradient of policy performance, denoted as $\Delta_\phi J_\pi(\phi)$, is referred to as the PG. Algorithms that optimize the policy using this gradient are termed PG algorithms.

From Equation (2.3.2), it is apparent that the weight $g(\tau)$ associated with action a_t encompasses the entire temporal process, including both time before and after t. This contradicts the definition of value functions, where the value of action a_t should rely solely on the subsequent return after time t. Nevertheless, it introduces unrelated historical rewards. Taking this into consideration, the revised PG is [32]:

$$\Delta_\phi J_\pi(\phi) = \mathbb{E}_{\tau\sim\phi}[\sum_{t=0}^{T} g_t \Delta_\phi \log \pi_\phi(a_t|s_t)] \tag{2.3.4}$$

where $g_t = r_t + \gamma r_{t+1} + \ldots + \gamma^{T-t} r_T$.

A more advanced technique in PG method is given as

$$\begin{aligned} \Delta_\phi J_\pi(\phi) &= \mathbb{E}_{\tau\sim\phi}[\sum_{t=0}^{T} g_t \Delta_\phi \log \pi_\phi(a_t|s_t)] \\ &= \mathbb{E}_{\tau\sim\phi}[\sum_{t=0}^{T} (g_t - b(s_t)) \Delta_\phi \log \pi_\phi(a_t|s_t)] \end{aligned} \tag{2.3.5}$$

where g_t is replaced with $g_t - b(s_t)$, where $b(s_t)$ is an arbitrary function serving as the baseline, taking s_t as input.

2.3.3 Actor-critic

RL algorithms can be broadly categorized into two main types: value-based methods and policy-based methods. Additionally, there is a method that combines both approaches, known as the actor-critic method. The actor-critic algorithm is a RL approach that combines PG and TD learning [33]. In this context, the actor refers to the policy function $\pi_\phi(a|s)$, which aims to maximize returns. The critic refers to the value function V, which estimates the value of the current policy, effectively evaluating the performance of the actor. With the assistance of the value function, the actor-critic algorithm can perform single-step parameter updates without waiting for the completion of an entire episode.

Let θ denote the parameters of the neural network representing the value function. The gradient of the objective function is given by:

$$\Delta_\phi J_\pi(\phi) = \mathbb{E}_{\tau\sim\phi}[\sum_{t=0}^{T}(g_t - V_\theta(s_t))\Delta_\phi \log \pi_\phi(a_t|s_t)] \tag{2.3.6}$$

where the baseline $b(s_t)$ in previous section is a parametrized value function.

To address the limitation of being unable to update the policy and value function before the end of an episode, the gradient of the objective function for the TD learning version is:

$$\Delta_\phi J_\pi(\phi) = \mathbb{E}_{\tau\sim\phi}[\sum_{t=0}^{T}(r_t + \gamma V_\theta(s_{t+1}) - V_\theta(s_t))\Delta_\phi \log \pi_\phi(a_t|s_t)] \tag{2.3.7}$$

where the value of $V_\theta(s_t)$ is updated through mean squared error between itself and its TD target using gradient descent, while the policy is updated through gradient ascent. Both processes occur simultaneously in learning and updating.

References

[1] M. de Kamps and F. van der Velde, "From artificial neural networks to spiking neuron populations and back again," *Neural Networks*, vol. 14, no. 6-7, pp. 941–953, 2001.

[2] T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman, and G. Hinton, "Backpropagation and the brain," *Nature Reviews Neuroscience*, vol. 21, no. 6, pp. 335–346, 2020.

[3] A. Karpathy, "Cs231n convolutional neural networks for visual recognition." https://cs231n.github.io. Accessed: 2023-11-18.

[4] A. Géron, *Neural networks and deep learning*. O'Reilly, 2018.

[5] S. Sharma, S. Sharma, and A. Athaiya, "Activation functions in neural networks," *Towards Data Sci*, vol. 6, no. 12, pp. 310–316, 2017.

[6] B. Ding, H. Qian, and J. Zhou, "Activation functions and their characteristics in deep neural networks," in *2018 Chinese control and decision conference (CCDC)*, pp. 1836–1841, IEEE, 2018.

[7] V. Thost and J. Chen, "Directed acyclic graph neural networks," *arXiv preprint arXiv:2101.07965*, 2021.

[8] Q. Li, G. Chalvatzaki, J. Peters, and Y. Wang, "Directed acyclic graph neural network for human motion prediction," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3197–3204, IEEE, 2021.

[9] P. Cunningham, M. Cord, and S. J. Delany, "Supervised learning," in *Machine learning techniques for multimedia: case studies on organization and retrieval*, pp. 21–49, Springer, 2008.

[10] T. Hastie, R. Tibshirani, J. Friedman, T. Hastie, R. Tibshirani, and J. Friedman, "Overview of supervised learning," *The elements of statistical learning: Data mining, inference, and prediction*, pp. 9–41, 2009.

[11] J. Li, Y. Si, T. Xu, and S. Jiang, "Deep convolutional neural network based ecg classification system using information fusion and one-hot encoding techniques," *Mathematical problems in engineering*, vol. 2018, pp. 1–10, 2018.

[12] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[13] A. G. Baydin, B. A. Pearlmutter, A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *ArXiv*, vol. abs/1502.05767, 2015.

[14] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural Networks: Tricks of the Trade: Second Edition*, pp. 437–478, Springer, 2012.

[15] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," *arXiv preprint arXiv:1804.07612*, 2018.

[16] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[17] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, vol. 25, Curran Associates, Inc., 2012.

[19] M. L. Puterman, "Markov decision processes," *Handbooks in operations research and management science*, vol. 2, pp. 331–434, 1990.

[20] S. J. Russell and P. Norvig, *Artificial intelligence a modern approach*. London, 2010.

[21] E. Barron and H. Ishii, "The bellman equation for minimizing the maximum cost.," *Nonlinear Analysis: Theory, Methods & Applications*, vol. 13, no. 9, pp. 1067–1090, 1989.

[22] G. Tesauro *et al.*, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[23] G. Tesauro, "Practical issues in temporal difference learning," *Advances in neural information processing systems*, vol. 4, 1991.

[24] C. Szepesvári, *Algorithms for reinforcement learning*. Springer Nature, 2022.

[25] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[26] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[27] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1334–1373, 2016.

[28] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[29] S. M. Kakade, "A natural policy gradient," *Advances in neural information processing systems*, vol. 14, 2001.

[30] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *International conference on machine learning*, pp. 387–395, Pmlr, 2014.

[31] T. Degris, M. White, and R. S. Sutton, "Off-policy actor-critic," *arXiv preprint arXiv:1205.4839*, 2012.

[32] O. S. Up, "Extra material: Proof for don't let the past distract you." https://

spinningup.openai.com/en/latest/spinningup/extra_pg_proof1.html. Accessed: 2023-11-16.

[33] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," *Advances in neural information processing systems*, vol. 12, 1999.

CHAPTER 3

Problem formulation and modeling

This chapter addresses the challenge of path following for nonholonomic wheeled mobile robots in a planar environment. The problem is presented in a universal manner, ensuring applicability across various vehicle models. Further details, including the iterative updating of path parameters and preprocessing techniques like arc-length parameterization, are discussed. The mobile robot is modeled as a typical two-wheeled differentially driven mobile robot, limited to the degrees of freedom for forward movement and rotation. The chapter concludes by introducing the pure pursuit algorithm, with a focus on the improved version employed in this dissertation.

3.1 Path following

The problem of path following can be expressed as a tracking problem on a vehicle-independent plane. In this section, this problem is described in a more generalized way, rendering it applicable to a wide range of vehicle models. Considering a controlled nonlinear system in the form below,

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(t)), \quad \mathbf{x}(t_0) = \mathbf{x}_0, \tag{3.1.1}$$

where $\mathbf{x} \in \mathcal{X} \subseteq \mathbb{R}^n$ and $\mathbf{u} \in \mathcal{U} \subseteq \mathbb{R}^m$ define state and input constraints, the objective of path following is to design a controller, such that the system follows a parametrized reference path [1],

$$\mathcal{P} = \{\mathbf{p} \in \mathbb{R}^2 | \mathbf{p} = \mathbf{p}_r(\lambda), \forall \lambda \geq 0\}. \tag{3.1.2}$$

For any given parameter λ, a local reference coordinate system $\{R\}$ centered at $\mathbf{p}_r(\lambda)$ can be defined, denoted by the subscript r. The relative angle δ_r between the fixed coordinate system $\{O\}$ in a two-dimensional plane and the local reference coordinate system $\{R\}$ can be calculated with Equation (3.1.3).

$$\delta_r(\lambda) = \text{atan2}(y_r^{'}(\lambda), x_r^{'}(\lambda)) \tag{3.1.3}$$

where the function atan2 used here is the four-quadrant version of arctan, which calculates the angle between the positive x-axis and the vehicle position $[x_r, y_r]^T$ in the Cartesian plane, with a positive counterclockwise direction. $x_r^{'}$ and $y_r^{'}$ are the first order derivatives. Moreover, it is clear that the parametrized reference path must exhibit continuous differentiability.

Considering the vehicle's pose at time t as $[x(t), y(t), \psi(t)]$, the error in path following, commonly referred to as the cross-track error, is determined by Equation (3.1.4), which is a cross product between two vectors [2, 3]. The control objective is to guarantee that the path following error converges such that $\lim_{t\to\infty} e_p(t) = 0$.

$$e_p(t) := d_y\hat{t}_x - d_x\hat{t}_y \tag{3.1.4}$$

where $\mathbf{d} = (d_x, d_y)$ is the tracking error vector and $\hat{\mathbf{t}} = (\hat{t}_x, \hat{t}_y)$ is the unit tangent vector to the reference path at $\lambda(t)$ as defined in Equations (3.1.5) and (3.1.6), respectively.

$$\mathbf{d} = (x(t), y(t)) - (x_r(\lambda(t)), y_r(\lambda(t))) \tag{3.1.5}$$

$$\hat{\mathbf{t}} = \frac{(x_r^{'}(\lambda(t)), y_r^{'}(\lambda(t)))}{\|(x_r^{'}(\lambda(t)), y_r^{'}(\lambda(t)))\|} \tag{3.1.6}$$

The orientation error $\psi_e(t)$ between the vehicle and the reference path at time t is determined by Equation (3.1.7), which is frequently incorporated and is typically treated as a secondary objective, or used to assist in the elimination of the cross-track error. It indicates moving towards or away from the direction of the path. In practical applications, normalization techniques are utilized to confine the value of $\psi_e(t)$ within the range of $[-\pi, \pi]$. While the trigonometric operations remain unchanged, constraining the range to this specific interval aids in reducing the observation space.

$$\begin{aligned} \psi_e(t) &= \psi(t) - \delta_r(\lambda(t)) \\ \psi_e(t) &= \text{atan2}(\sin(\psi_e(t)), \cos(\psi_e(t))) \end{aligned} \tag{3.1.7}$$

Distinguishing from path following, the goal of trajectory tracking is to develop a controller so that the system state $\mathbf{x}(t)$ follows a time-varying reference trajectory $\mathbf{r}(t)$ which is generated by a trajectory planning algorithm in some cases. The tracking problem can be represented as a stabilization problem, where the objective is to achieve $\lim_{t\to\infty}(\mathbf{x}(t) - \mathbf{r}(t)) = 0$.

3.1.1 Nearest point calculation

To determine the point along the reference path for calculating the cross-track error, the point nearest to the vehicle is selected [4, 5, 3]. It leads to an optimization problem of finding the parameter λ that minimizes the distance between the vehicle position and the reference path. The preference for the squared Euclidean distance is based on its equivalence with the original optimization problem and its computational convenience. The optimization problem can be expressed as follows:

$$\min_{\lambda} f(\lambda) = \min_{\lambda} \|(x(t), y(t)) - (x_r(\lambda), y_r(\lambda))\|^2 \tag{3.1.8}$$

A common approach for updating the path variable λ involves iteratively computing the value that minimizes the distance between the vehicle and the reference path. This process can be accomplished through the application of the conjugate gradient method as depicted in Equation (3.1.9), which is highly efficient when solving quadratic convex optimization problems, often converging to the optimal solution within a finite number of steps [6]. Unlike certain other optimization methods, such as the Newton's method, the conjugate gradient method does not require the storage of large Hessian matrices and only relies on first-order derivatives being continuous [7, 8]. Simultaneously, the feature of guaranteeing only a local optimum serves to prevent abrupt jumps in the path parameter, fostering stability in the optimization process. This is achieved by employing the previous path variable value as the initial guess.

$$\lambda_{k+1} = \lambda_k - \eta_k d_k \tag{3.1.9}$$

where $k \in \mathbb{N}$ denotes the iteration, and the step size η_k is typically determined through a line search to ensure the best descent along the current search direction d_k. The specific formula for computing η_k is represented in Equation (3.1.10).

$$\eta_k = \frac{\langle \nabla f(\lambda_k), \nabla f(\lambda_k) \rangle}{\langle \nabla f(\lambda_k), \mathbf{d}_k \rangle} \tag{3.1.10}$$

where the first order derivative $\nabla f(\lambda)$ of the objective function is given as follow.

$$\nabla f(\lambda) = -2(x(t) - x_r(\lambda))x_r^{'}(\lambda) - 2(y(t) - y_r(\lambda))y_r^{'}(\lambda) \tag{3.1.11}$$

The direction of gradient descent, denoted as d_k, is chosen to be a linear combination of the negative gradient $-\nabla f(\lambda_k)$, which represents the steepest descent direction for the objective function [9], and the previous direction d_{k-1}, and can be written as:

$$d_k = -\nabla f(\lambda) + \beta_k d_{k-1} \tag{3.1.12}$$

where the scalar β_k is to be determined by the requirement that d_{k-1} and d_k are conjugate, and it is computed as follows.

$$\beta_k = \frac{\langle \nabla f(\lambda_{k+1}), \nabla f(\lambda_{k+1}) \rangle}{\langle \nabla f(\lambda_k), \nabla f(\lambda_k) \rangle} \tag{3.1.13}$$

This entire computation process is detailed in Algorithm 1. By employing the conjugate gradient method, the optimal value of λ is iteratively computed until the estimate approaches the desired ideal value with sufficient closeness.

Algorithm 1 Nearest Point Calculation with Conjugate Gradient Method

Initialize λ_0

Choose initial search direction $d_0 = -\nabla f(\lambda_0)$

for $k = 0, 1, 2, \ldots$ **do**

Choose step size: $\eta_k = \frac{\langle \nabla f(\lambda_k), \nabla f(\lambda_k) \rangle}{\langle \nabla f(\lambda_k), \mathbf{d}_k \rangle}$

Update parameter: $\lambda_{k+1} = \lambda_k + \eta_k d_k$

Compute gradient: $\nabla f(\lambda_{k+1})$

if Tolerance for termination is satisfied **then**

Terminate the loop

end if

Compute β_k for the next search direction:

$\beta_k = \frac{\langle \nabla f(\lambda_{k+1}), \nabla f(\lambda_{k+1}) \rangle}{\langle \nabla f(\lambda_k), \nabla f(\lambda_k) \rangle}$

Update search direction: $d_{k+1} = -\nabla f(\lambda_{k+1}) + \beta_k d_k$

end for

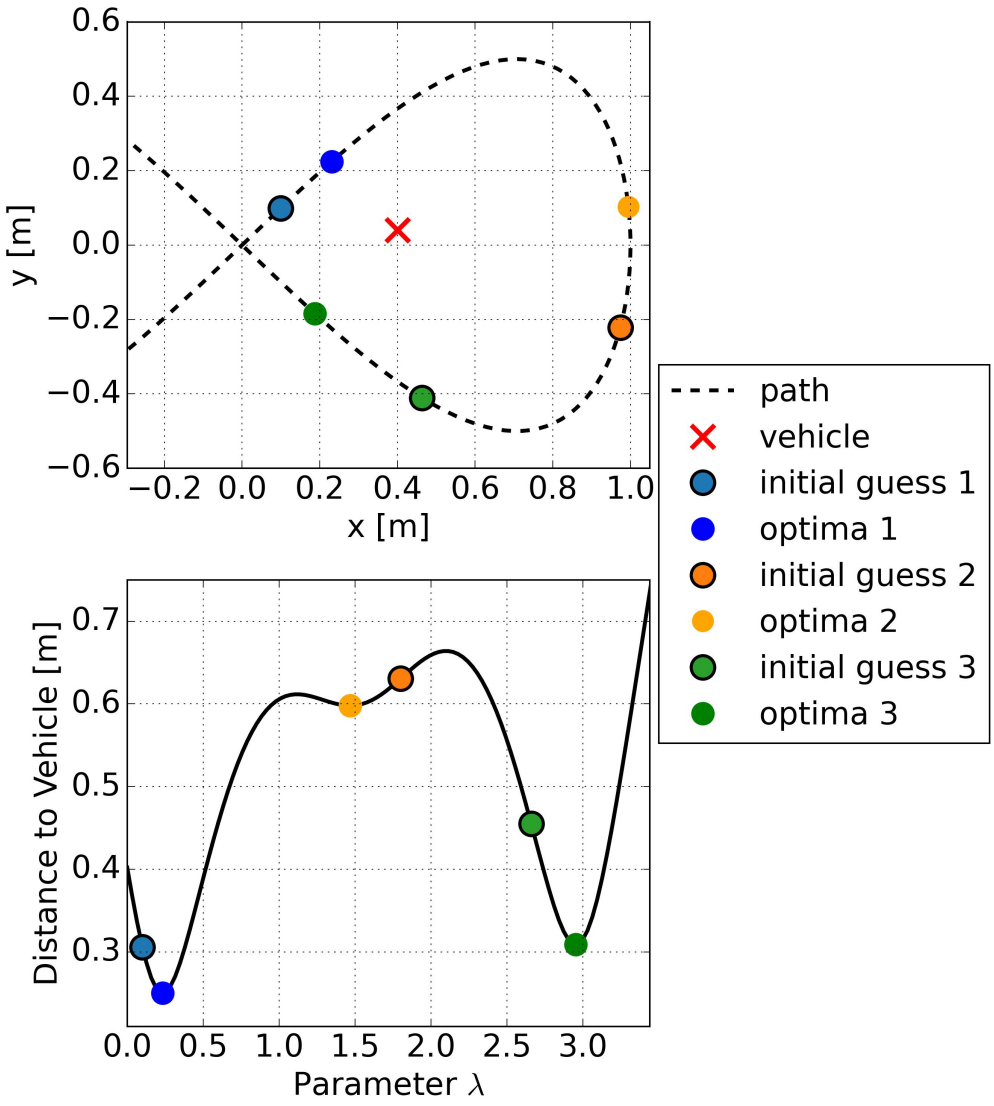

Figure 3.1: A schematic representation of the parameter λ calculation for the nearest point.

The termination tolerance is determined by the proximity of the derivative to zero. In practice, the initial guess before tracking the path can be based on a bold estimate of the vehicle's position. For vehicles starting from a location near the beginning of the path, zero can be used as the initial estimate for the entire path following process. Using the parameter estimated from the previous iteration as the starting point for finding a new nearest point ensures that the optimal solution is the nearest local minimum which additionally guarantees smooth path following and prevents sudden parameter jumps. This phenomenon is illustrated in Figure 3.1. Due to the careful selection of initial parameter guess and the fact that only small increments are made in each iteration, the parameter do not abruptly transition from one segment of the path to another, even in the presence of multiple local optima.

3.1.2 Path arc length parameterization

In this dissertation, the path $\mathcal{P}$ is a continuous function with an independent parameter λ. The transformation of the parameter into arc length offers specific advantages, enabling a convenient representation of the distance traveled by a vehicle on the path through variations in the parameter. Thus, this section provides an introduction on how any first-order differentiable continuous function can be transformed into an arc length-based path function.

When dealing with a given function, whether it is a naturally occurring function or one derived through interpolation using multiple waypoints, it is possible to determine the arc length between the starting point of the path and any arbitrary point along the curve using the following formula:

$$\Delta\lambda = \int_0^c \sqrt{(\frac{\mathrm{d}x_r}{\mathrm{d}\lambda})^2 + (\frac{\mathrm{d}y_r}{\mathrm{d}\lambda})^2}\ \mathrm{d}\lambda \tag{3.1.14}$$

where, c is within the range $[0, \hat{\lambda}]$ with $\hat{\lambda}$ representing the path's endpoint [10, 11]. This process transforms the original variable λ into a new parameter, $\Delta\lambda$, based on arc length. Subsequently, the newly parameterized function can be obtained through reinterpolation using the corresponding path points with respect to λ. The path, once parameterized by arc length, can be expressed as follows.

$$\mathcal{P} = \{\mathbf{p} \in \mathbb{R}^2 | \mathbf{p} = \mathbf{p}_r(\Delta\lambda), \forall \Delta\lambda \geq 0\} \tag{3.1.15}$$

Evident in Figure 3.2, a substantial disparity arises in the discrete path points before and after arc length parameterization. The upper figure displays results derived from the original parameters, while the lower figure illustrates the outcomes after arc length parametrization, both comprising 30 discrete points. Pre-parameterization, the discretization reveals non-uniformity, characterized by extended arc lengths between points along straight segments and a more concentrated distribution around curved segments. Post-arc length parameterization, the discretization of path points attains uniformity across diverse segments of the path.

This phenomenon becomes apparent upon examining the correlation between parameters and arc length, graphically represented in Figure 3.3. Arc length parameterization has transformed the initially non-uniform relationship into a one-to-one linearization. In the subsequent chapters, the distinction between between λ and $\Delta\lambda$ will no longer be maintained. For the sake of convenience and uniformity, λ will be used exclusively for representation, and it should be noted that the default path has already been arc length parameterized.

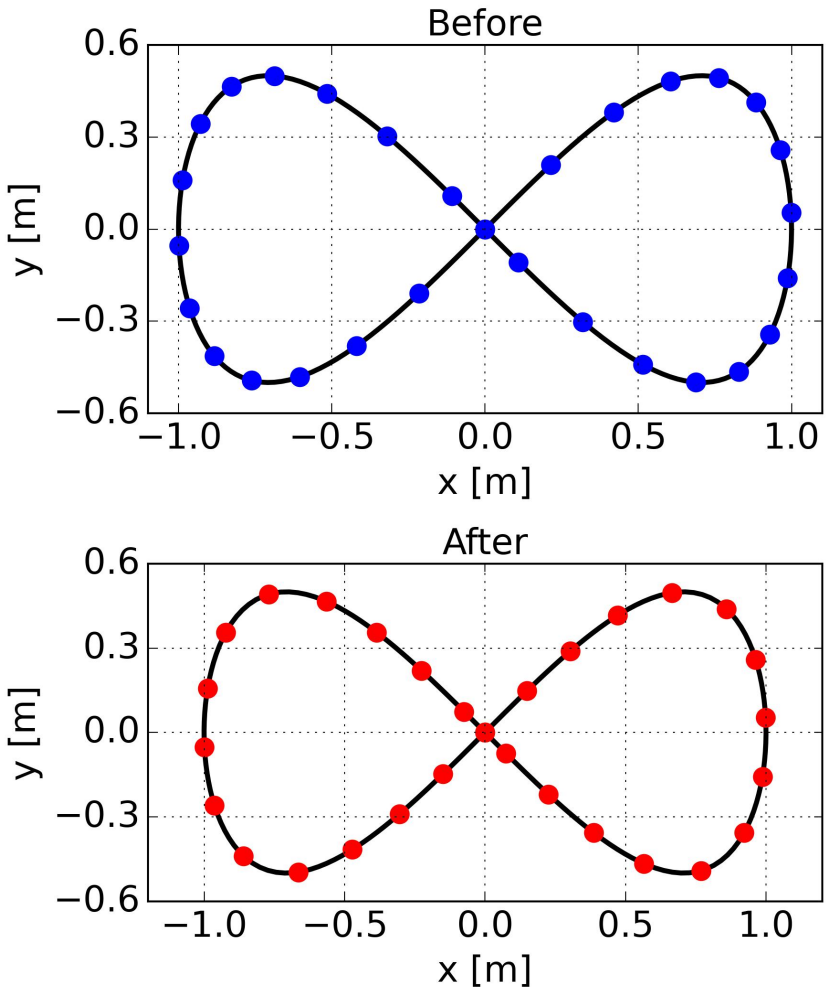

Figure 3.2: The comparison of path points corresponding to uniformly spaced parameters before and after arc length parameterization.

3.2 Kinematics

Kinematic modeling is a critical aspect of mobile robotics, as it serves as the foundation for designing effective path planning algorithms and motion control strategies [12]. Nonholonomic mobile robots are a class of mobile robots whose mobility is constrained, unlike typical holonomic mobile robots, which can move freely in a plane. Nonholonomic mobile robots often have certain motion restrictions, such as being able to move only along specific trajectories or execute specific types of motion [13].

This type of robot is particularly useful in certain applications, such as specific industrial tasks or environments, where they can be precisely designed to perform particular jobs without requiring complete freedom of movement [14]. The design and control of non-holonomic mobile

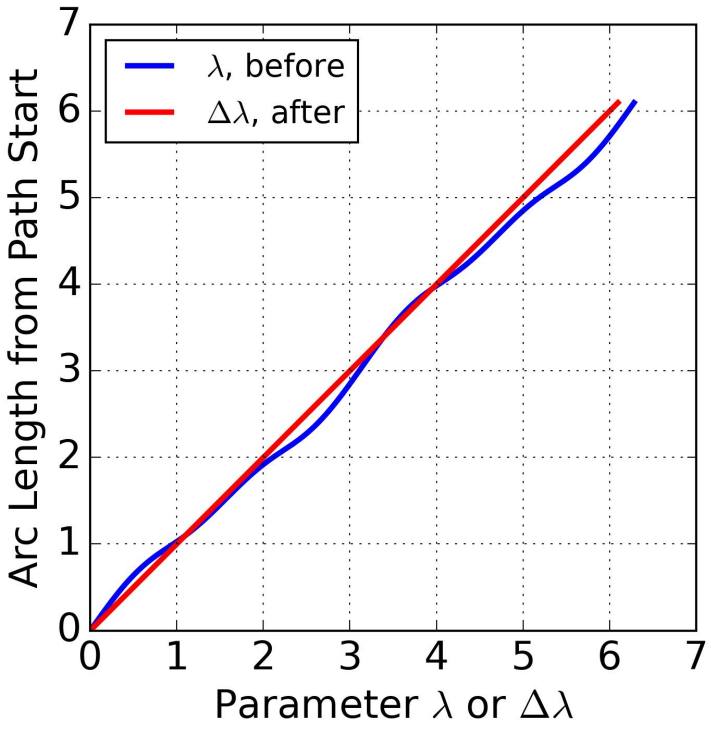

Figure 3.3: The arc length variation from the starting point to endpoint along the path.

robots typically take these motion constraints into consideration. Steering in nonholonomic mobile robots is achieved by independently controlling the speed of the wheels on each side of the vehicle. When the speeds of the wheels on both sides are not equal, the vehicle will turn. Basic differential steering robots are equipped with two driven wheels and a front and rear caster for added stability as illustrated in Figure 3.4.

3.2.1 Modeling

In the context of a two-wheeled independently driven mobile robot as depicted in Figure 3.4, $\{O\}$ represents a fixed absolute coordinate system within a two-dimensional plane, while $\{Q\}$ denotes a coordinate system fixed on the robot itself. The origin is positioned at the midpoint of the axis connecting both wheels, and the forward direction, as illustrated in Figure 3.4, is defined as the X-axis, perpendicular to the axis. In the absolute coordinate system, the robot has three degrees of freedom in its posture, which are indicated by its posture,

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ \psi \end{bmatrix} \tag{3.2.1}$$

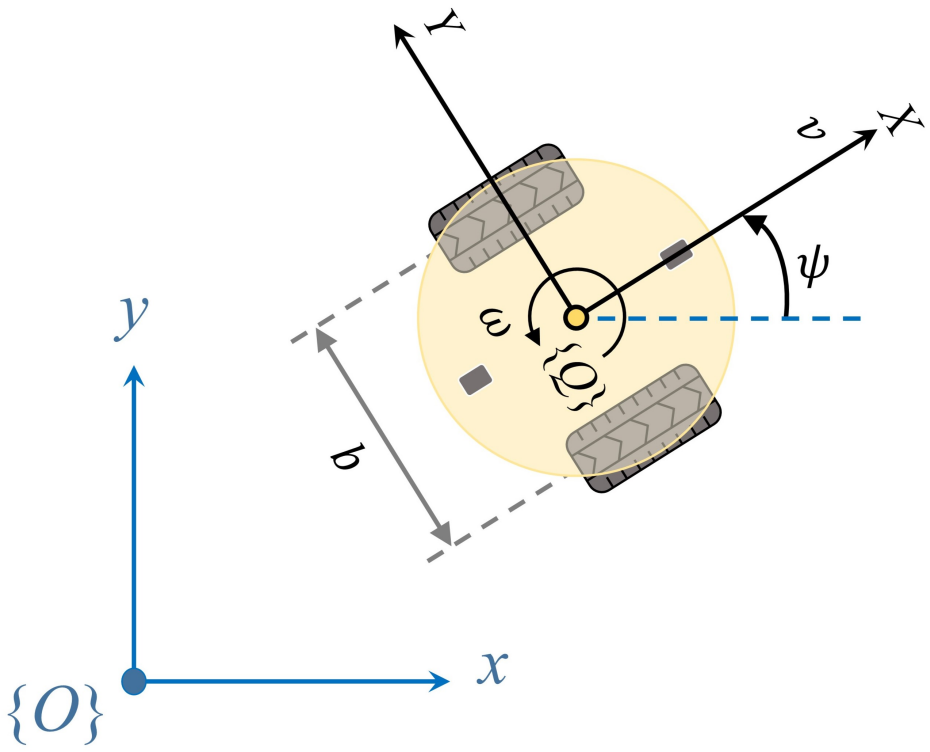

Figure 3.4: A two-wheeled independently driven mobile robot.

where (x, y) represents the robot's current position, and ψ represents the heading, measured counterclockwise from the x-axis.

The mobile robot's motion is controlled by its linear velocity v and rotational velocity ω, as their positive directions are defined in Figure 3.4. The mobile robot's kinematics is then defined as follows [12, 13, 15, 16],

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} \cos\psi & 0 \\ \sin\psi & 0 \\ 0 & 1 \end{bmatrix} \mathbf{u} \tag{3.2.2}$$

where $\mathbf{u} = [v, \omega]^T \in \mathcal{U} \subset \mathbb{R}^2$ define input constraints. Path following of such a nonholonomic wheeled mobile robot involves the design of algorithms to generate reference commands for $\mathbf{u}$.

3.2.2 Determining maximum velocities

To further elaborate, the motion of the wheeled mobile robot is derived from the wheels independently driven by motors, and these wheels contribute to both linear and angular velocity. It can be modeled in both forward as represented by Equation (3.2.3) and in reverse

as represented by Equation (3.2.4).

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{b} & -\frac{1}{b} \end{bmatrix} \begin{bmatrix} v_r \\ v_l \end{bmatrix} \tag{3.2.3}$$

$$\begin{bmatrix} v_r \\ v_l \end{bmatrix} = \begin{bmatrix} 1 & \frac{b}{2} \\ 1 & -\frac{b}{2} \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{3.2.4}$$

where $v_r \in \mathbb{R}$ and $v_l \in \mathbb{R}$ are the linear velocities of the right and left wheels, respectively, and $b \in \mathbb{R}^+$ denotes the wheelbase, which is the distance between the centers of the wheels.

Let v_{max} represent the maximum linear velocity achievable by the mobile robot. It is crucial to account for the condition that, even when the robot simultaneously moves forward at a linear velocity v and rotates at an angular velocity ω, the velocity of the outer wheel should not exceed the maximum allowed velocity [15]. Therefore, the following constraint is applied:

$$\hat{v} + \frac{b}{2}\hat{\omega} \leq v_{max} \tag{3.2.5}$$

where $\hat{v}$ and $\hat{\omega}$ denote the set maximum velocities in practical use. The experimental robot in the dissertation is tested to move at a maximum velocity v_{max} of 0.5 m/s. Through the constraint outlined in Equation 3.2.5, the maximum values are determined and presented in Table 3.1.

Table 3.1: Mobile robot parameters.

Symbol	**Description**	**Value**
$\hat{v}$	Maximum linear velocity	0.4 m/s
$\hat{\omega}$	Maximum angular velocity	1.0 rad/s
b	Wheelbase	0.172 m

Now, the path following control system can be formally defined as depicted in Figure 3.5. In the context of practical applications, the velocity saturation and the transfer function from velocity commands to actual velocities are also incorporated into the block diagram. Generally, the path following algorithm takes both path information and model state information as inputs, yielding linear and angular velocity commands as outputs. Due to the imposition of velocity constraints, it is possible to design the two velocities independently.

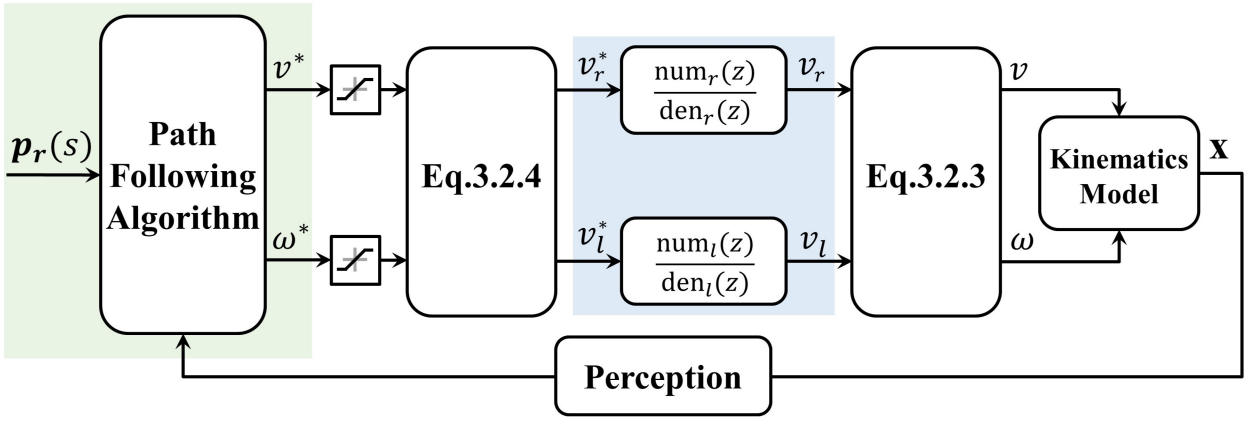

Figure 3.5: Structure of path following control system.

3.3 Pure pursuit control

The pure pursuit algorithm, originally designed for recalculating the arc needed to reposition a robot onto its path, has gained popularity due to its simplicity and effectiveness in real-time control applications [2]. The core principle of the pure pursuit controller involves fitting a circle through the vehicle's current position, often defined as the rear wheel position in the case of a car or, in the case of nonhomogeneous mobile robots, as the midpoint of the axis connecting both wheels. This circle is then aligned with a point on the path located ahead of the vehicle at a predetermined look-ahead distance, commonly denoted as L.

3.3.1 Fundamental control law

The point on the reference path located at a distance of one look-ahead distance from the robot is defined as per Equation (3.3.1) [17, 18]. As there are usually multiple points on the reference path at a distance of one look-ahead distance from the robot, the control is uniquely defined by selecting the point with the greatest value of the parameter λ.

$$\|(x, y) - (x_r(\lambda), y_r(\lambda))\| = L \tag{3.3.1}$$

In Figure 3.6, the geometry is depicted. The circle is defined as passing through the mobile robot's current position and the point on the path located one look-ahead distance ahead of the robot, with the circle being tangent to the robot's heading. The curvature of the circle, which is the reciprocal of the circle's radius, is calculated as described in Equation (3.3.2).

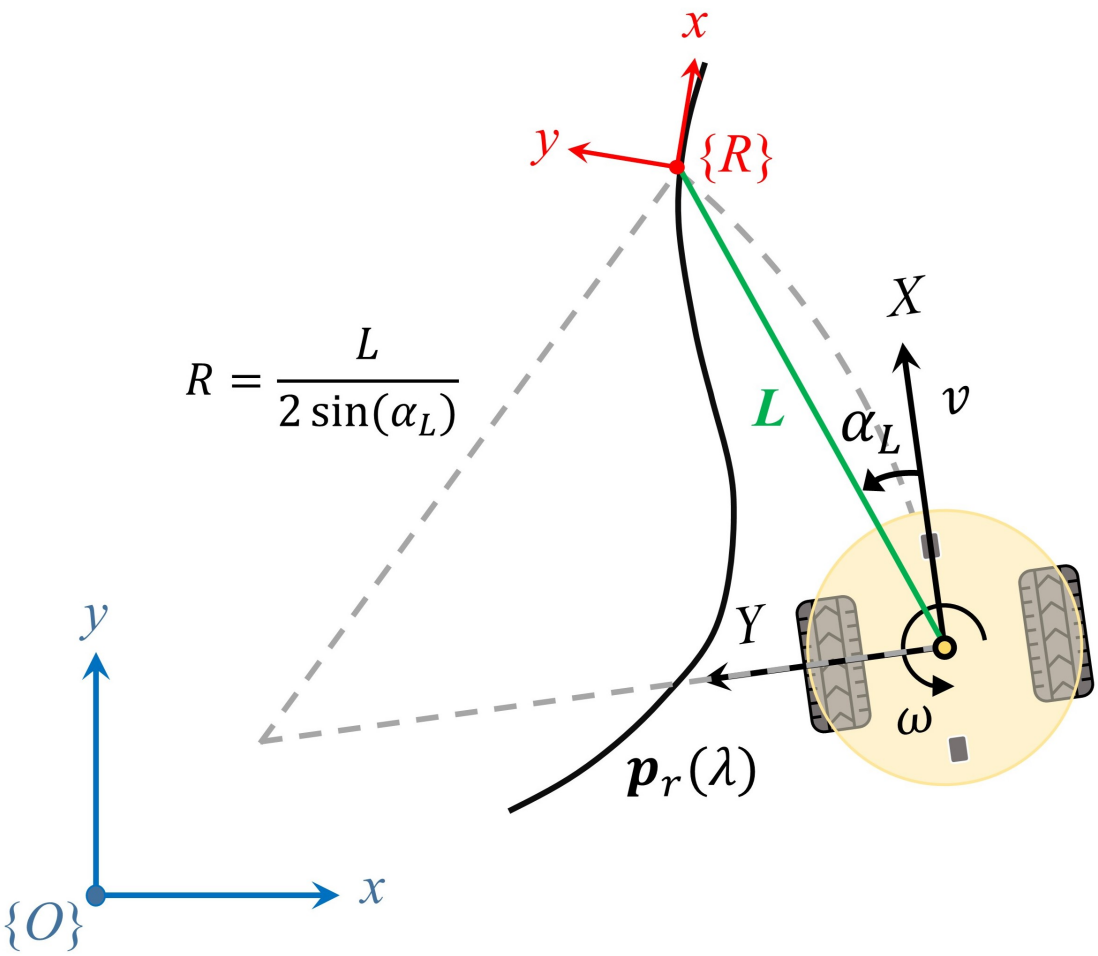

Figure 3.6: Geometry of the pure pursuit algorithm.

$$k = \frac{2\sin(\alpha_L)}{L} \tag{3.3.2}$$

where the look-ahead angle α_L is given by

$$\alpha_L = \arctan(\frac{y_r(\lambda) - y}{x_r(\lambda) - x}) - \psi \tag{3.3.3}$$

In essence, the pure pursuit algorithm allows the robot to pursue a desired path by continuously adjusting its heading to intersect with the selected point on the path, effectively following the desired trajectory. This straightforward approach has found widespread utility in various applications that demand real-time control. It offers an elegant solution for maintaining a vehicle's course along a predefined path, making it a reliable and robust choice, especially when dealing with scenarios involving a linear path and constant vehicle velocity.

Eventually, the commanded heading rate for a robot traveling at velocity v is defined as per Equation (3.3.4).

$$\omega^* = \frac{2v\sin(\alpha_L)}{L} \tag{3.3.4}$$

3.3.2 Drawbacks and improvements

The pure pursuit controller is particularly useful when the reference path has no curvature, and the robot maintains a constant velocity. However, this method has significant drawbacks. In cases where the robot's distance to the path exceeds L, the controller output is not well-defined. When the robot exceeds this distance, there is a risk of the look-ahead point and the nearest point becoming too close, leading to the possibility of excessive steering. Furthermore, in situations where the reference path exhibits significant curvature, there can be abrupt variations in the parameter λ corresponding to the look-ahead distance. This can be highly detrimental to achieving smooth and precise control.

To address the limitation associated with the look-ahead distance, an improvement can be achieved by selecting a point that is located at an arc length of d forward along the reference path from the point nearest to the robot's current position relative to the path, denoted as $[x_r(\lambda), y_r(\lambda)]$. This newly selected point, $[x_r(\lambda + d), y_r(\lambda + d)]$, is then designated as the look-ahead point as illustrated in Figure 3.7.

Due to the groundwork laid in the preceding sections, the implementation of this aspect can be readily accomplished. In this case, the new look-ahead distance is defined as follows.

$$L = \|(x, y) - (x_r(\lambda + d), y_r(\lambda + d))\| \tag{3.3.5}$$

By doing this, a corresponding control strategy is established, regardless of the robot's relative position to the path. It's important to note that, for robots moving at a constant velocity, this strategy involves a larger look-ahead distance during straight-line motion and a smaller look-ahead distance when there is curvature. This pattern is in line with typical human driving habits. Moreover, due to the smooth variation of the parameter λ, the look-ahead point also exhibits a smooth transition, preventing abrupt changes. In the subsequent utilization of the pure pursuit algorithm, an improved version incorporating these modifications will be employed, with d set to the value specified in Table 3.2.

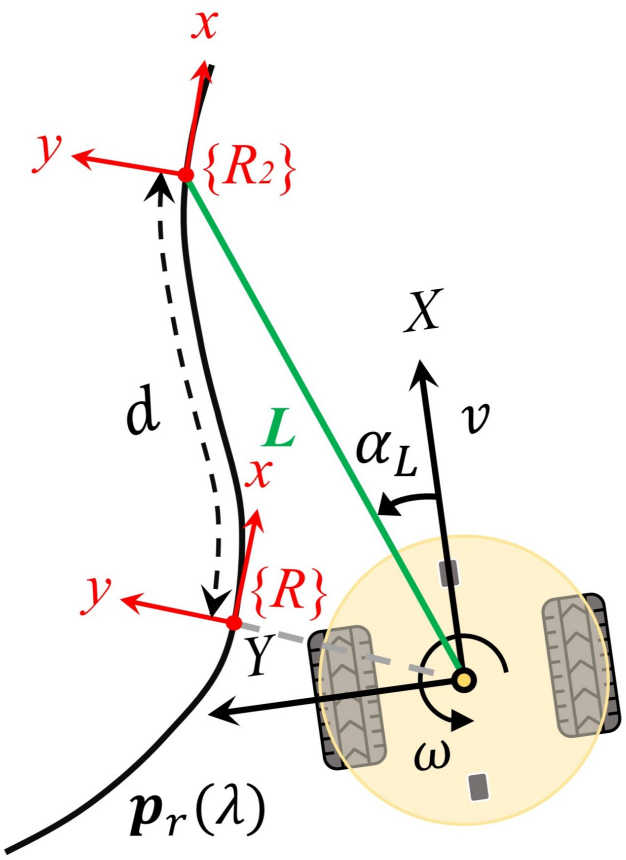

Figure 3.7: Geometry of the improved pure pursuit algorithm.

Table 3.2: Arc length forward along the path.

Symbol	Value
d	0.2 m

The limitations of the pure pursuit algorithm become evident when dealing with curved paths. While it performs well on paths with small curvature within the maximum speed range, challenges arise on segments with significant curvature. Due to the constraints of constant speed control, it is likely to deviate from the path, exceeding the look-ahead distance specified in the original algorithm.

Clearly, maintaining a constant speed is not an optimal control strategy. Manually assigning a reference linear velocity to each path point or parameter λ is one approach, but it introduces the challenge of repetitive design for different scenarios. Therefore, in this dissertation, the

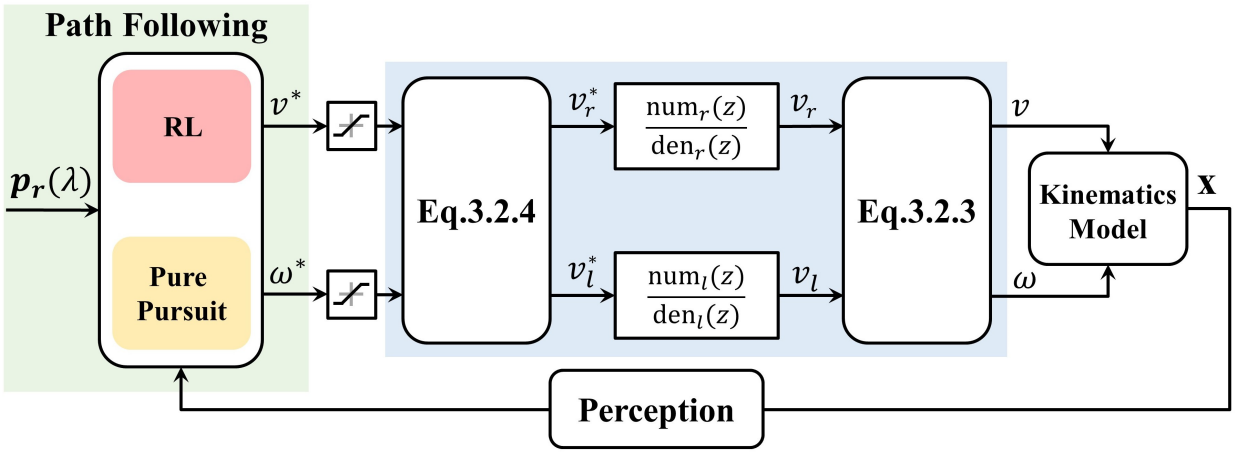

Figure 3.8: Structure of proposed path following control system.

use of RL to interact with environments formed by various curved paths is investigated. While maintaining the responsibility of steering control within the pure pursuit algorithm, the objective is to learn an adaptive velocity control strategy to achieve the goal of path following. The proposed control framework is illustrated in Figure 3.8. The reason for not using RL simultaneously for both aspects is because it was found that this actually decreased tracking performance. Due to the intervention of the look-ahead point, a cheating phenomenon occurs when RL learns steering control concurrently, where the steering is directly reduced for the sake of speed improvement.

References

[1] T. Faulwasser, B. Kern, and R. Findeisen, "Model predictive path-following for constrained nonlinear systems," in *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*, pp. 8642–8647, IEEE, 2009.

[2] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," *IEEE Transactions on intelligent vehicles*, vol. 1, no. 1, pp. 33–55, 2016.

[3] C. Samson, "Path following and time-varying feedback stabilization of a wheeled mobile robot," *Second International Conference on Automation, Robotics and Computer Vision*, vol. 3, 1992.

[4] A. B. Martinsen and A. M. Lekkas, "Curved path following with deep reinforcement learning: Results from three vessel models," in *OCEANS 2018 MTS/IEEE Charleston*, pp. 1–8, 2018.

[5] J. D. Rounsaville, J. S. Dvorak, and T. S. Stombaugh, "Methods for calculating relative cross-track error for asabe/iso standard 12188-2 from discrete measurements," *Transactions of the ASABE*, vol. 59, no. 6, pp. 1609–1616, 2016.

[6] S. P. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[7] W. W. Hager and H. Zhang, "A survey of nonlinear conjugate gradient methods," *Pacific journal of Optimization*, vol. 2, no. 1, pp. 35–58, 2006.

[8] B. T. Polyak, "The conjugate gradient method in extremal problems," *USSR Computational Mathematics and Mathematical Physics*, vol. 9, no. 4, pp. 94–112, 1969.

[9] H. D. Sherali and O. Ulular, "Conjugate gradient methods using quasi-newton updates with inexact line searches," *Journal of Mathematical Analysis and Applications*, vol. 150, no. 2, pp. 359–377, 1990.

[10] F.-C. Wang and D. Yang, "Nearly arc-length parameterized quintic-spline interpolation for precision machining," *Computer-Aided Design*, vol. 25, no. 5, pp. 281–288, 1993.

[11] H. Wang, J. Kearney, and K. Atkinson, "Arc-length parameterized spline curves for real-time simulation," in *Proc. 5th International Conference on Curves and Surfaces*, vol. 387396, 2002.

[12] R. González, F. Rodríguez, and J. L. Guzmán, "Autonomous tracked robots in planar off-road conditions," *Modeling, Localization, and Motion Control—Springer*, 2014.

[13] P. I. Corke, W. Jachimczyk, and R. Pillat, *Robotics, vision and control: fundamental algorithms in MATLAB*, vol. 73. Springer, 2011.

[14] M. B. Alatise and G. P. Hancke, "A review on challenges of autonomous mobile robot and sensor fusion methods," *IEEE Access*, vol. 8, pp. 39830–39846, 2020.

[15] Y. Kanayama, Y. Kimura, F. Miyazaki, and T. Noguchi, "A stable tracking control method for an autonomous mobile robot," in *Proceedings., IEEE International Conference on Robotics and Automation*, pp. 384–389, IEEE, 1990.

[16] W. Dong and K.-D. Kuhnert, "Robust adaptive control of nonholonomic mobile robot

with parameter and nonparameter uncertainties," *IEEE Transactions on Robotics*, vol. 21, no. 2, pp. 261–266, 2005.

[17] R. C. Coulter *et al.*, *Implementation of the pure pursuit path tracking algorithm*. Carnegie Mellon University, The Robotics Institute, 1992.

[18] O. Amidi and C. E. Thorpe, "Integrated mobile robot control," in *Mobile Robots V*, vol. 1388, pp. 504–523, SPIE, 1991.

CHAPTER 4

Design and implementation

In this chapter, building upon the content of the previous two chapters, the design and implementation of DRL in path following are discussed. Specifically, the SAC algorithm is employed in a stochastic path following environment. In Section 4.1, standard RL is extended to MERL, and the SAC algorithm is introduced. In Section 4.2, this algorithm is utilized to design a training process, including the design of the observation space, action space, and reward function. Section 4.3 provides a detailed exploration of additional nuances during the training process.

4.1 Maximum entropy reinforcement learning

Entropy is a measure used to quantify the randomness of a random variable, and in practical calculations, it is directly related to the probability distribution that the variable follows. For instance, to compute the entropy of a variable X following a distribution P, the entropy $\mathcal{H}(X)$ is given by

$$\mathcal{H}(X) = \mathbb{E}_{x\sim P}[-\log P(x)] \tag{4.1.1}$$

where in RL, the degree of stochasticity in the policy π under a specific state can be represented by $\mathcal{H}(\pi(\cdot|s))$, which is the expected value of the logarithmic selection probability for actions.

$$\mathcal{H}(\pi(\cdot|s)) = \sum_a -\pi(a|s)\log\pi(a|s) = \mathbb{E}_{a\sim\pi}[-\log\pi(a|s)] \tag{4.1.2}$$

In other words, when the policy entropy is high, it means that there is a higher likelihood of different actions a being adopted for the same input state s. Therefore, by incorporating

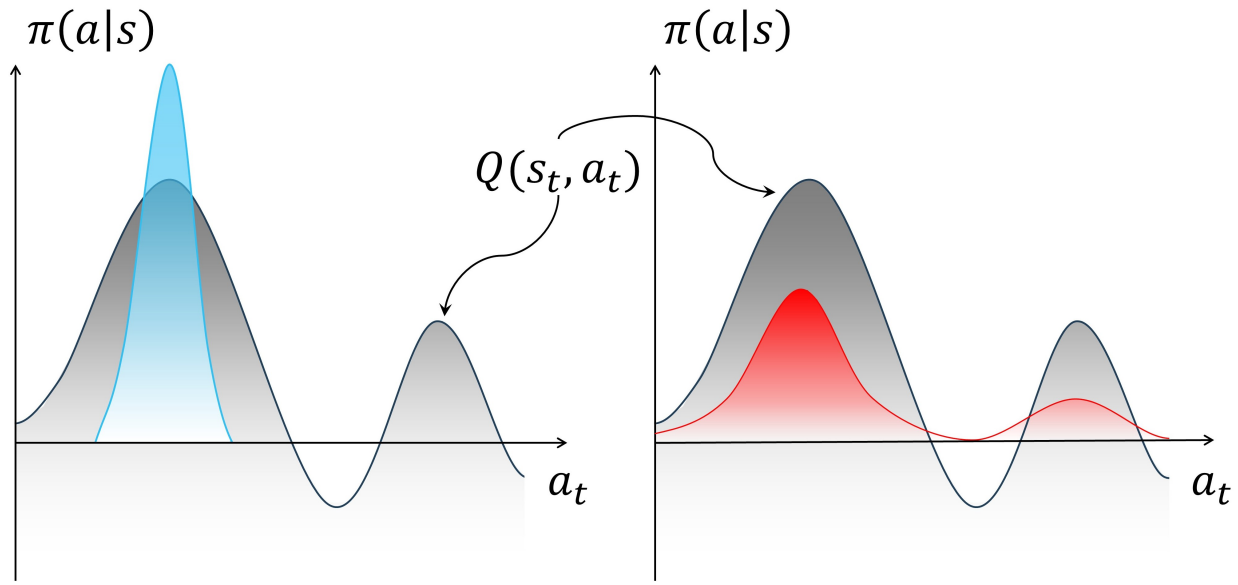

Figure 4.1: Comparison between standard RL (left) and MERL (right).

this policy entropy term into the objective function, it is expected to naturally address the exploration problem in Q-learning.

The MERL generalizes the objective of standard RL by introducing a regularization term in the form of entropy [1, 2], such that the optimal policy π^* simultaneously maximizes expected return and entropy,

$$\pi^* = \arg\max_{\pi} \sum_{t} \mathbb{E}_{(s_t,\ a_t)\sim\rho^{\pi}} \left[r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t)) \right] \tag{4.1.3}$$

where ρ^{π} represents the distribution of state-action pairs that the agent will encounter under the control of policy π, and the parameter α represents the temperature, influencing the trade-off between the entropy term and the reward in determining the optimal policy's stochasticity. The larger α is, the stronger the exploratory behavior, facilitating accelerated subsequent policy learning and reducing the likelihood of the policy getting stuck in suboptimal local optima. While the maximum entropy objective diverges from the conventional maximum expected return objective in standard RL, the latter can be reclaimed in the limit as α approaches zero.

In Figure 4.1, considering the definition of $Q(s_t, a_t)$ as the expected cumulative reward following the execution of action a at state s, the standard RL approach involves defining a policy distribution centered around the maximum Q-value. This distribution incorporates exploration noise that spans neighboring actions, depicted by a blue distribution. However,

this conventional approach may result in the overestimation of Q-values, a phenomenon frequently reported. In contrast, the MERL objective ensures effective exploration for the Q-function, as illustrated by a red distribution.

4.1.1 Soft actor-critic algorithm

The Bellman equation, a fundamental recursive relationship widely employed in the field of RL, takes the following form in soft Q-learning,

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim p}[V(s_{t+1})] \tag{4.1.4}$$

where p denotes the state transition probability, and the soft state-value function $V(s_t)$ based on MERL is defined by

$$V(s_t) = \mathbb{E}_{a_t \sim \pi}[Q(s_t, a_t) - \alpha \log \pi(a_t|s_t)] \tag{4.1.5}$$

In the SAC algorithm [3, 4], optimization is performed for function approximators of both the soft Q-function and the policy. The soft Q-function is parameterized by $\theta \in \mathbb{R}^n$, representing a vector of n parameters, and can be effectively modeled using a DNN. The optimization of the soft Q-function is achieved by employing a policy evaluation algorithm, such as TD learning. Similarly, the policy, parameterized by $\phi \in \mathbb{R}^m$, is modeled as a Gaussian with mean and covariance determined by a DNN. The parameters of the soft Q-function can be optimized by minimizing the mean squared loss given by Equation (4.1.6).

$$L_Q(\theta) = \mathbb{E}_{s_t, a_t \sim \mathcal{D}}[(Q_\theta(s_t, a_t) - y_t)^2] \tag{4.1.6}$$

where y_t given in Equation (4.1.7) is the TD target that the soft Q-function is updating towards. The update makes use of a target soft Q-function with parameters $\bar{\theta}$, where $i = 1, 2$ denotes the double Q-function approximators, referred to as the clipped double-Q trick [5].

$$y_t = r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim p}[\min_{i=1,2} Q_{\bar{\theta}_i}(s_{t+1}, a'_{t+1}) - \alpha \log \pi(a'_{t+1}|s_{t+1})] \tag{4.1.7}$$

The objective for updating the policy parameters is defined by maximizing the expected future return plus future entropy, equivalent to minizing the expected divergence in Equation (4.1.8). Due to the challenges in directly sampling latent actions and computing gradients, the policy is reparameterized such that $a_t = f_\phi(\xi_t; s_t) = \tanh(\mu_\phi(s_t) + \sigma_\phi(s_t) \odot \xi_t)$ within

a finite bound. Here, μ_ϕ and σ_ϕ represent the mean and standard deviation of the actions, respectively, and $\xi_t \sim \mathcal{N}(0, I)$ is an input noise vector. The reparameterized sample is thus differentiable.

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim \mathcal{D},\ \xi_t \sim \mathcal{N}}[\alpha \log \pi_\phi(f_\phi(\xi_t; s_t)|s_t) - Q_\theta(s_t, f_\phi(\xi_t; s_t))] \tag{4.1.8}$$

where π_ϕ is implicitly defined in relation to f_ϕ.

4.1.2 Automating entropy adjustment

In the work of Haarnoja et al. [3], the temperature parameter α is treated as a fixed constant, despite the necessity for tuning it across various tasks. Simply imposing a fixed entropy value is an inadequate solution because the policy should retain the freedom to explore more in regions where the optimal action is uncertain, while maintaining a more deterministic nature in states where the distinction between good and bad actions is clear.

In their subsequent research [4], the authors introduced a constrained optimization problem. Here, the average entropy of the policy is constrained, allowing for variability in entropy at different states. This approach provides a more nuanced way of addressing the exploration-exploitation trade-off in RL. The objective of discovering a stochastic policy with maximal expected return, satisfying a minimum expected entropy constraint, is presented in Equation (4.1.9).

$$J(\alpha) = \mathbb{E}_{a_t \sim \pi_t}[-\alpha \log \pi_t(a'_t|s_t) - \alpha\bar{\mathcal{H}}] \tag{4.1.9}$$

where the entropy target $\bar{\mathcal{H}}$ is set to be the negative of the action space dimensiondim($\mathcal{A}$).

4.2 Path following with soft actor-critic

The path-following controller, rooted in the SAC algorithm, employs a pair of neural networks—one specifically designed for learning the policy and the other for learning the value function. To mitigate bias in estimation, target networks are introduced, incorporating delayed updates. Moreover, it integrates an experience replay buffer $\mathcal{D}$ for storing and replaying samples, effectively reducing sample correlation, enhancing sample efficiency, and improving the learning capability of the algorithm. The overall architecture is depicted in Figure 4.2. Notably, a' represents actions resampled from the current policy rather than from prior experiences.

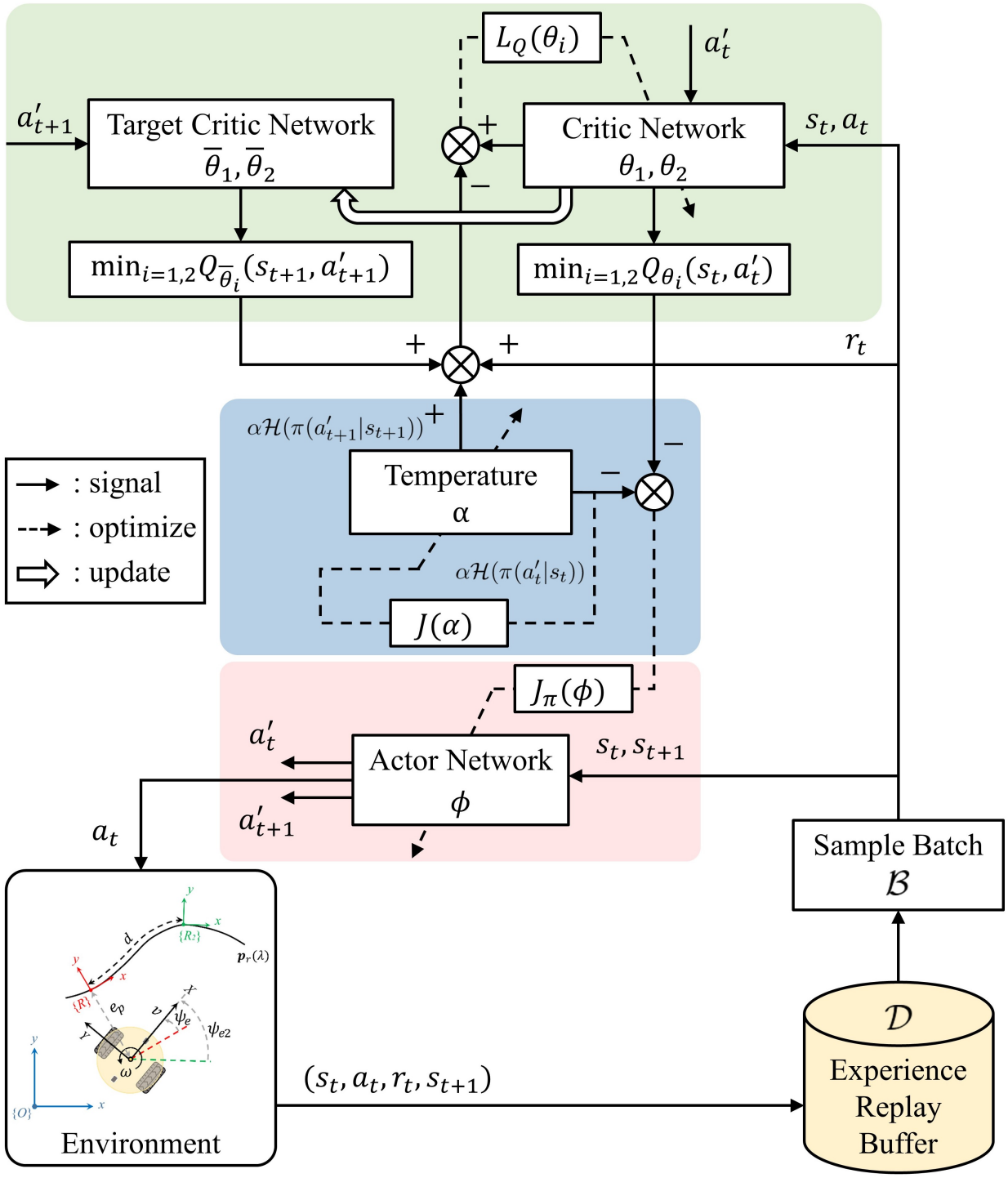

Figure 4.2: The optimization process of SAC-based path-following controller at each time-step.

4.2.1 Observation space and action space

As the considered MDP is partially observable, the term "observation" is used instead of "state" to denote the information the agent relies on when taking actions. In this context, the agent, represented by the mobile robot, performs actions in the environment and observes the resulting changes in the environment's state. The observation s, serving as the input to the path-following controller, is selected in an intuitive and low-dimensional manner:

$$s = \{e_p, \psi_e, v, \omega, \psi_{e2}\} \tag{4.2.1}$$

where e_p is the cross-track error, $\psi_e \in [-\pi, \pi]$ is the normalized orientation error between the path and the mobile robot, v and ω are the linear velocity and rotational velocity of the robot, respectively. ψ_{e2} selected from the look-ahead point as discussed in section 3.3, functions as an augmented observation that provides information about the curvature of the path in the future.

Additional details regarding the observation are provided in Table 4.1, and a graphical explanation is presented in Figure 4.3. It is worth noting that the absolute value of the cross-track error is not limited due to the success of the pure pursuit controller, which can always pull the robot back to the path and keep the observation in cross-track error within a range that is not excessively large.

Table 4.1: Observation space.

Symbol	**Description**	**Min**	**Max**
e_p	Cross-track error	$-\infty$ m	∞ m
ψ_e	Orientation error	$-\pi$ rad	π rad
v	Linear velocity	0.0 m/s	0.4 m/s
ω	Rotational velocity	-1.0 rad/s	1.0 rad/s
ψ_{e2}	Orientation error of the look-ahead point	$-\pi$ rad	π rad

The action, denoted as $a(t) = \pi(s(t)|\phi)$, represents the rate of linear velocity $\dot{v}$ normalized by the velocity itself. This selection is adopted to mitigate undesired rapid changes. Adopting an incremental control input for the robot makes it easier to achieve smooth motions without the need for additional rewards or penalties for excessive velocity changes. Furthermore,

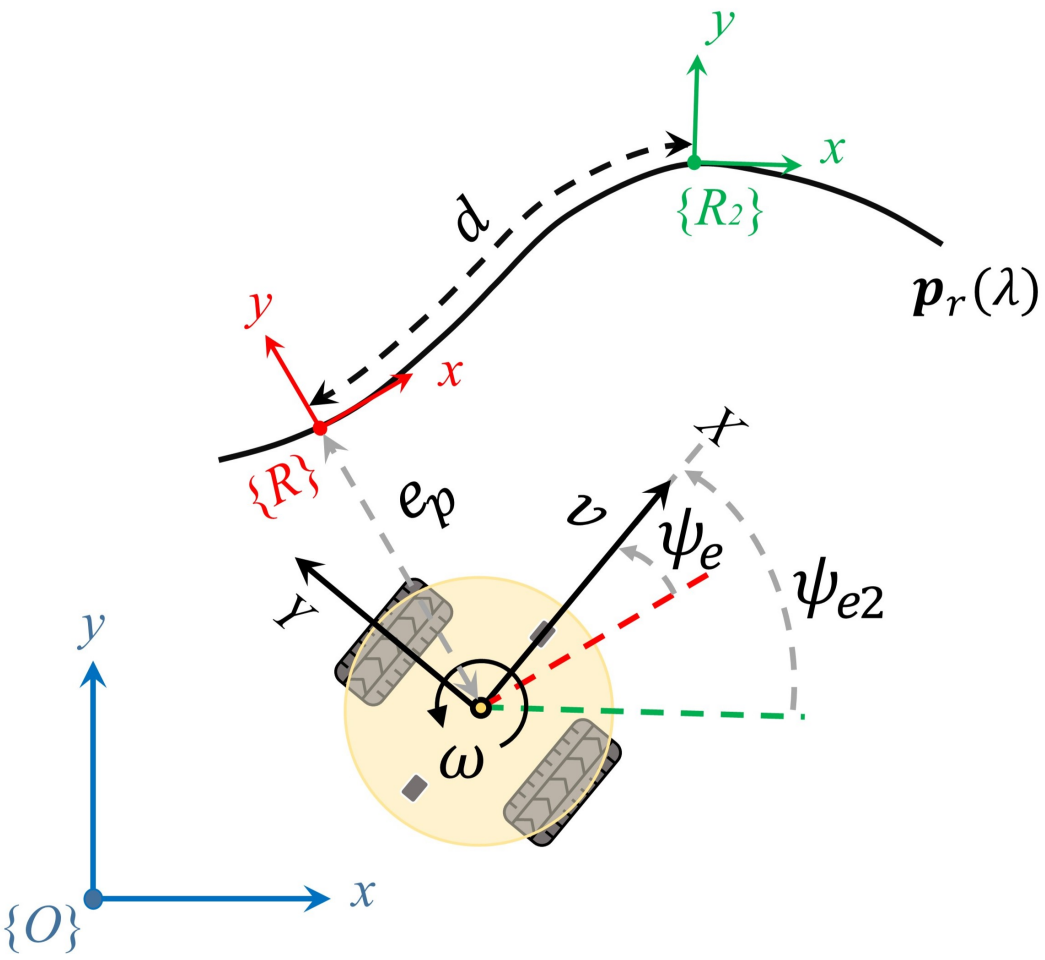

Figure 4.3: Observations concerning a predetermined reference path.

constraining the velocity rate within a specified range, i.e., $\dot{v} \in [\dot{v}_{min}, \dot{v}_{max}]$, provides a better stability. The linear velocity command $v^*(t)$ after saturation operation for the mobile robot at each time step is calculated using Equation (4.2.2) as discussed in the previous section.

$$\begin{aligned} a(t) &= \tanh(\mu_\phi(s(t)) + \sigma_\phi(s(t)) \odot \xi(t)) \\ \dot{v}^*(t) &= ka(t) + b \\ v^*(t) &= \text{clip}(v(t) + \dot{v}^*(t)\Delta t, v_{min}, v_{max}) \end{aligned} \tag{4.2.2}$$

where $k = (\dot{v}_{max} - \dot{v}_{min})/2$ and $b = (\dot{v}_{max} + \dot{v}_{min})/2$ are the scale and bias, respectively, to recover the normalized action to the range of the desired action. The range of action is listed in Table 4.2. The linear velocity rate, that is, acceleration, is employed to achieve smooth velocity control. The exploration space for deceleration is slightly larger than that for acceleration, addressing situations requiring urgent braking.

Table 4.2: Action space.

Symbol	Description	Min	Max
$\dot{v}$	Linear acceleration	-0.5 m/s^2	0.3 m/s^2

As depicted in Figure 3.8, the steering control is governed by the pure pursuit algorithm, as defined in section 3.3. In addition to the advantages mentioned within the section 3.3, this design approach avoids repeated optimization calculations, requiring only the computation of the nearest point.

4.2.2 Rewards

Considering the policy's objective is to maximize long-term returns, the meticulous design of a reward function is crucial for achieving satisfactory performance. In the realm of path following, an intuitive approach involves incentivizing the agent to minimize the cross-track error concerning the desired path. In the study by Rubí et al. [6], rewards are granted for staying on the path and moving forward, with penalties imposed through an absolute value function for deviations. Meyer et al. [7] employ an exponential reward function, rewarding the agent for incremental improvements to an unsatisfactory location.

With careful consideration, the reward function is designed to penalize the robot when it deviates from the path, while rewarding the robot's velocity as much as possible, as depicted in Equation (4.2.3).

$$r(t) = -k_1|e_p(t)| + k_2 v(t)\left(1 - \frac{1}{e_{tol}}|e_p(t)|\right) - k_3 F(t) \tag{4.2.3}$$

where k_1, k_2 and k_3 are positive constants that define the importance of each term. e_{tol} is the tolerance for lateral deviation within which the robot receives positive velocity rewards. This entire concept aims to incentivize the robot to stay close to the desired path while simultaneously encouraging the robot to achieve a potential maximum velocity. Based on intuitive considerations, it is desirable for the robot to decrease its velocity when deviating from the path to prevent further error expansion. To achieve this, a segmented penalty approach is introduced. When the cross-track error exceeds a critical threshold, the penalty on velocity increases accordingly. This design ensures that the policy receives velocity rewards only when the lateral deviation is within the critical threshold. The reward function with the

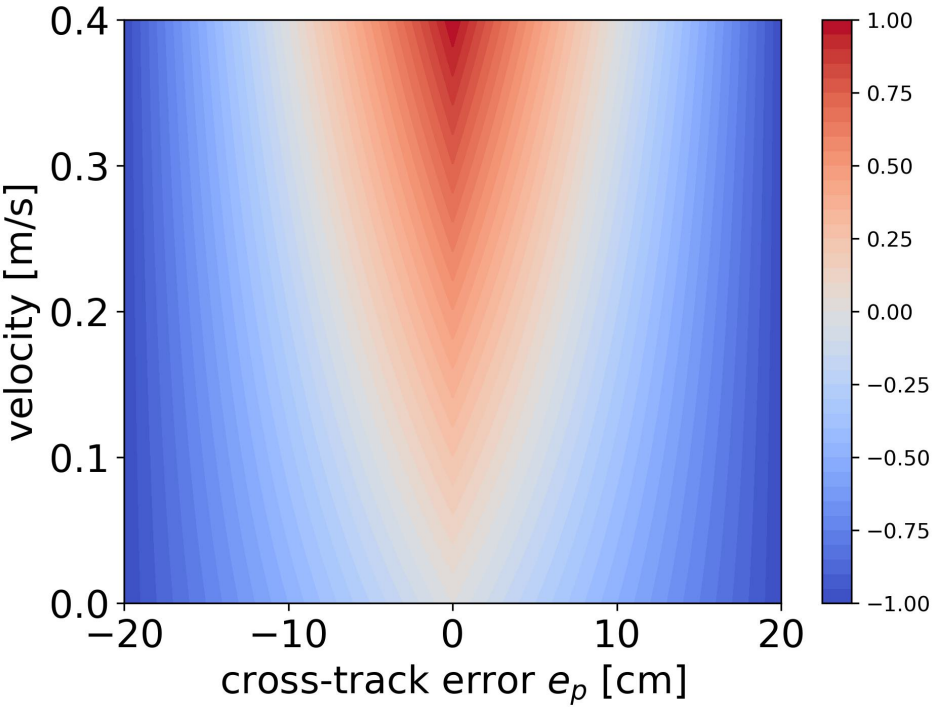

Figure 4.4: Reward function for path-following within the e_p range of $[-e_{tol}, e_{tol}]$.

first two terms is visualized in Figure 4.4, with the range of $[-e_{tol}, e_{tol}]$.

Additionally, it has been observed in experiments that the agent may choose to discontinue forward movement at challenging turns to avoid potential penalties. To address this situation, a flag F defined as

$$F(t) = \begin{cases} 1 & \text{if } v(t) < \epsilon, \\ 0 & \text{otherwise.} \end{cases}$$

indicating a stationary state has been introduced, where ϵ is an extremely small value such as 1×10^{-6}. Parameters of the reward function are detailed in Table 4.3.

Table 4.3: Reward function parameters.

Symbol	**Value**
k_1	5.0
k_2	2.5
k_3	0.2
e_{tol}	0.2 m

Within this range, the reward at each step ranges from a maximum value of 1, indicating perfect tracking of the path at maximum speed, to a minimum value of -1. Due to enhancements in the refined pure pursuit algorithm, the mobile robot can consistently track the point ahead of the path at any lateral distance. In this scenario, it is sufficient to solely investigate the reward associated with the robot traveling along the path, which is $\psi_e \in [-\frac{\pi}{2}, \frac{\pi}{2}]$.

4.2.3 Training environment

The operational context within which the agent is anticipated to operate is characterized by a predefined path. It is imperative to incorporate the kinematics of the mobile robot into the training environment, given that this parameter lies beyond our direct influence. Furthermore, ensuring the adaptability of the agent to diverse challenges is of paramount importance, as it fosters a capacity for handling generalized scenarios, thereby mitigating the risk of over-fitting to specific trajectories. In light of these considerations, an algorithm is proposed to generate stochastic reference paths, as delineated in Algorithm 2. This algorithm aims to equip the agent with the ability to handle a variety of situations it has not encountered before, fostering a strong and adaptable skill set. A visual representation of paths randomly generated utilizing this algorithm is presented in Figure 4.5.

Algorithm 2 Stochastic Path Generator

Generated waypoint counter $n \leftarrow 1$

Starting waypoint $\mathbf{p}_1 \leftarrow [0, 0]$

Number of path waypoints $N_w \in \mathbb{N}$

Range of length between waypoints $[L_{min}, L_{max}] \in \mathbb{R}^+$

while $n \leq N_w$ **do**

  Sample L_w from $\mathcal{U}(L_{min}, L_{max})$

  Sample δ_w from $\mathcal{U}(0, 2\pi)$

  New waypoint $\mathbf{p}_{n+1} \leftarrow \mathbf{p}_n + L_w[\cos(\delta_w), \sin(\delta_w)]^T$

  $n \leftarrow n + 1$

end while

Create arc length parameterized path $\mathbf{p}_r(s) = [x_r(\lambda), y_r(\lambda)]^T$ using Cubic Spline Interpolator as described in 3.1.2

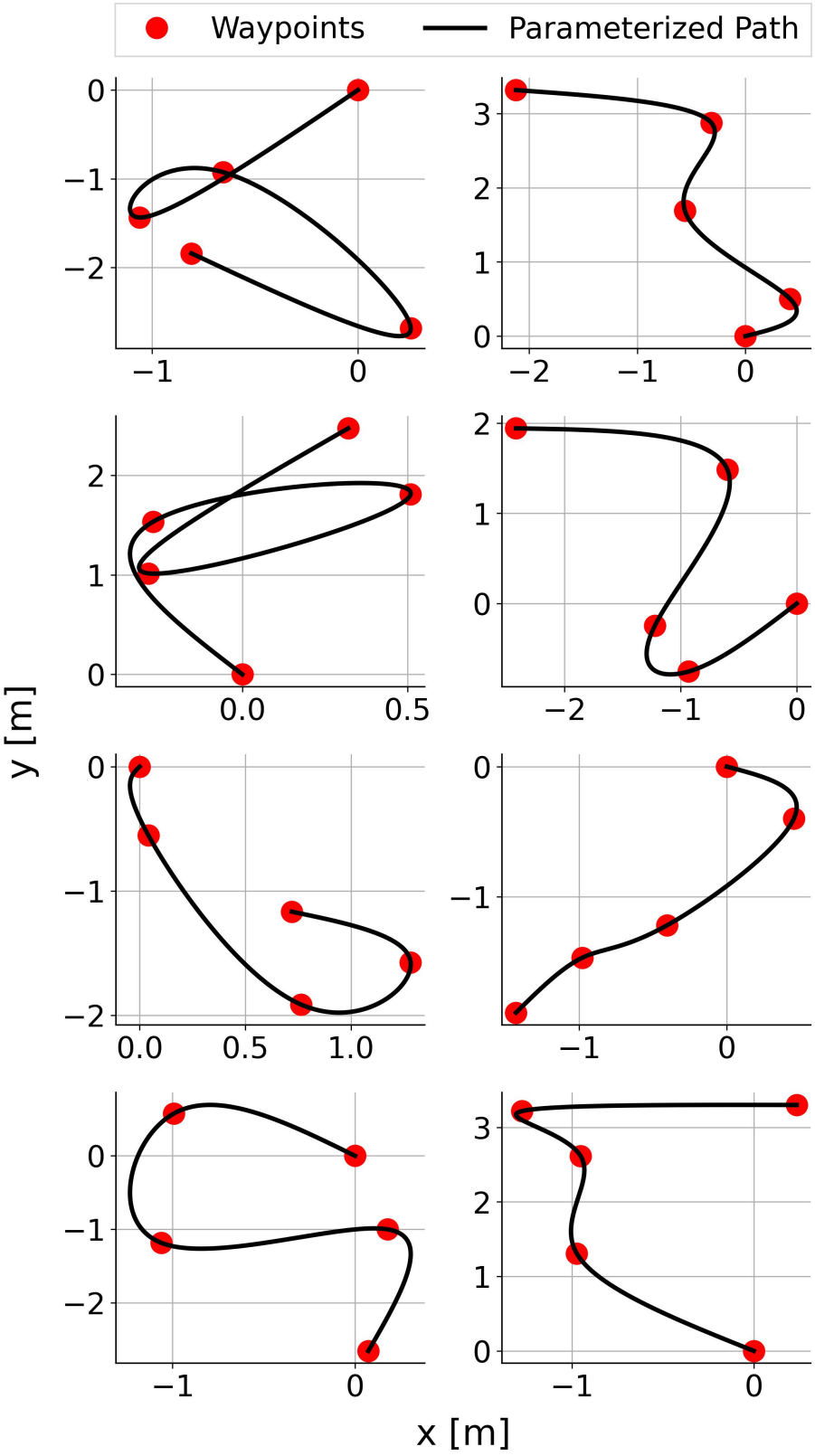

Figure 4.5: Example of randomly generated reference paths.

In the context of this dissertation, the parameterization of the stochastic path generation algorithm is defined with $N_w = 5$, $L_{min} = 0.5$ m and $L_{max} = 2.0$ m. It is noteworthy that the algorithm selectively generates curved paths for the purpose of training. In practical training, straight paths are also introduced into the training every ten episodes, achieved by setting $N_w = 2$ and $L_w = 2.5$ m. The incorporation of stochasticity in the generation of reference paths is deliberate, as it introduces an element of unpredictability, mirroring the uncertainties inherent in real-world applications.

The deliberate emphasis on curved paths in the training dataset is motivated by a strategic intent to intricately associate velocity adjustments with path curvature. This deliberate selection aims to foster a more nuanced understanding of the intricate interplay between velocity modulation and the varying curvatures encountered during robotic navigation. By exclusively utilizing curved paths in the training process, the trained agent is expected to develop a heightened sensitivity to curvature-dependent velocity adjustments, ultimately enhancing its adaptability in negotiating complex and dynamically evolving environments. This methodological choice aligns with the overarching objective of cultivating a robust and versatile autonomous navigation system, wherein the agent's responsiveness to path curvature is prioritized as a key element in its decision-making process.

4.3 Implementation details

The actor and critic neural networks are both structured with two hidden layers, as depicted in Figures 4.6 and 4.7. Each layer is equipped with ReLU activation function, featuring 256 neurons in both hidden layers. The actor's final layer outputs the mean $\mu_\phi(s_t)$ and standard deviation $\sigma_\phi(s_t)$ of a distribution, facilitating the sampling of a valid action. Subsequently, the action undergoes a tanh transformation to confine its range, as outlined in Equation (4.2.2). In the critic network, the action and state are concatenated to form an input. Notably, the action in the optimization process is resampled instead of being retrieved from the experience buffer replay. This strategic adjustment ensures a dynamic and responsive adaptation of the critic network to the evolving exploration-exploitation dynamics, thereby contributing to the overall learning stability of the RL algorithm.

The weights are initialized following the method outlined in [8], with a slight modification: uniform distribution of $[-3 \times 10^{-4}, 3 \times 10^{-4}]$ is employed to initialize the final layers of the actor and critic networks. This adjustment is implemented to alleviate output saturation during the initial stages of training. For the optimization of neural networks, the Adam

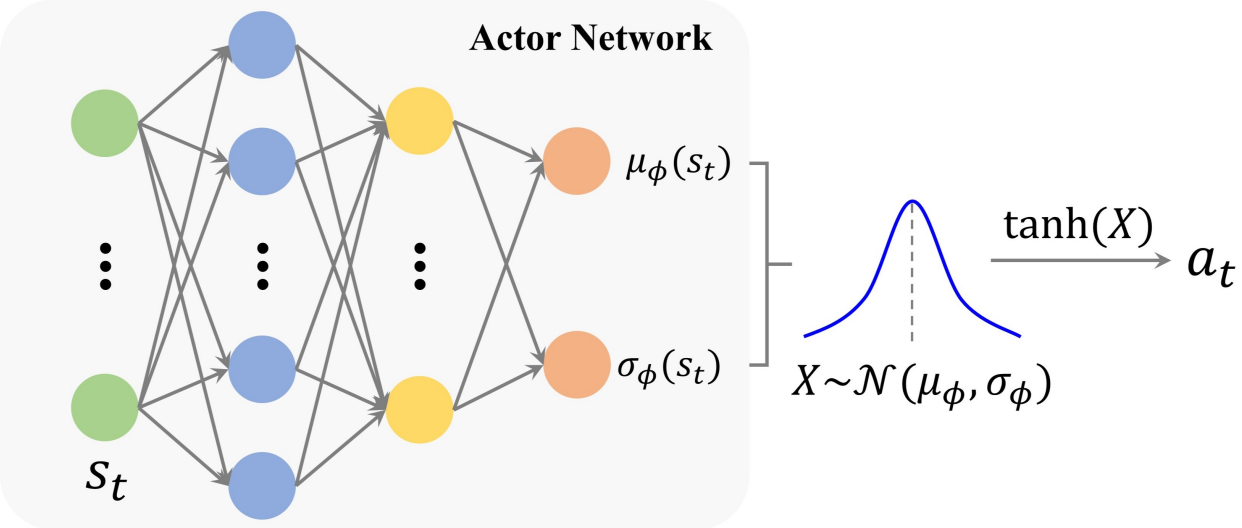

Figure 4.6: Structure of the actor network.

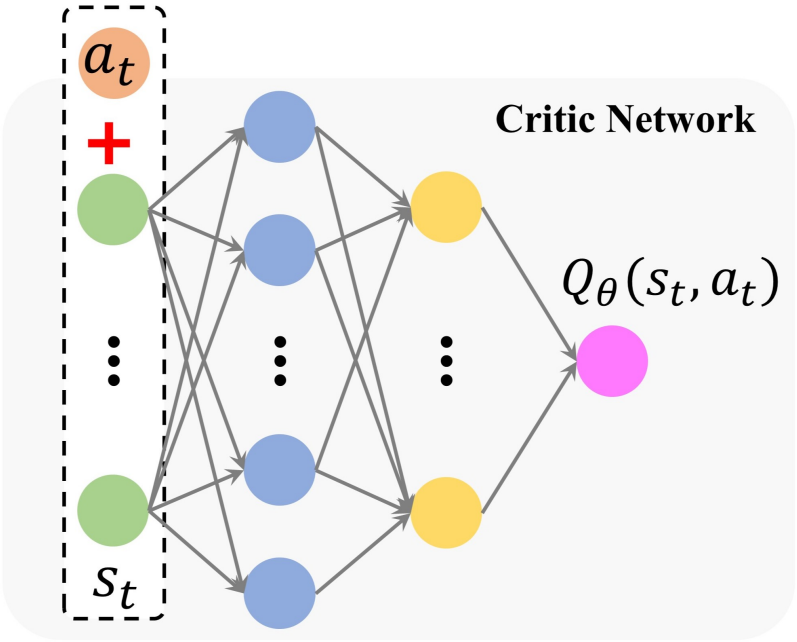

Figure 4.7: Structure of the critic network.

optimizer [9] is utilized with a minibatch size of 256. Subsequently, training is conducted on five different random seeds, allowing for an assessment of the algorithm's effectiveness and stability. Hyperparameters are summarized in Table 4.4.

Table 4.4: Hyperparameters of neural network approximators for both actor and critic networks for path following.

Description	**Value**
Number of hidden layers	2
Number of neurons per layers	256
Activation function	ReLU
Optimizer	Adam
Learning rate	3×10^{-4}
Minibatch size	256

Before each training episode, a stochastic reference path is generated and the initial pose of the mobile robot is randomly sampled from a uniform distribution, with a position range of $[-0.1, 0.1]$ meters and a heading error range of $[-0.0873, 0.08723]$ radians, which is approximately $\pm$ 5 degrees. A warm-up strategy is also implemented to gather completely random experiences. During the initial training phase, the agent takes random actions uniformly sampled from the action space. At each time step, the actor network generates an action based on the observed state, transitions to the next state, computes the corresponding reward, and stores it in the experience replay buffer as a tuple. Subsequently, the optimization process is initiated only if there are a sufficient number of tuples in the buffer to form a minibatch, and this occurs after the warm-up phase has concluded, as depicted in Figure 4.2.

The loss required for updating the parameters of the critic and actor networks can be computed using Equations (4.1.6) and (4.1.8). Following the optimization of the networks, the parameters of the target networks undergo an update using a soft update strategy, denoted by Equation (4.3.1). Specifically, a fraction of the updated network parameters is blended with the target network parameters. This soft update strategy is instrumental in stabilizing the learning process and mitigating oscillations.

$$\bar{\theta} = \tau\theta + (1 - \tau)\bar{\theta} \tag{4.3.1}$$

where parameter τ indicates how fast the update is carried on and the update is performed at each step after optimizing the online critic networks. The relevant parameters and their values used during training are presented in Table 4.5.

Table 4.5: Hyperparameters in training the SAC for path following.

Description	**Value**
Sampling period	0.05 s
Target soft update rate	0.005
Discount factor	0.99
Entropy target	-1
Maximum time steps per episode	4×10^2
Warm-up time steps	5×10^3
Maximum time steps	5×10^5
Experience replay buffer size	5×10^5

4.3.1 Tools and libraries

The implementation of the SAC algorithm relied on the utilization of the PyTorch library [10]. This library, known for offering a comprehensive DL framework, played a key role in constructing and training neural networks for the dissertation. In the course of the implementation, PyTorch's tensor operations and high-level modules, particularly the torch.nn module, to configure the actor and critic networks were used. Additionally, PyTorch's optimization algorithms to iteratively adjust the weights of the networks throughout the training process were incorporated. The training of the agent unfolded within a customized path-following environment crafted using Python. This environment enabled the simulation of diverse scenarios, allowing for a thorough evaluation of the agent's performance across various conditions.

For the convenient deployment of the trained policy, the post-training neural network has been converted to the Open Neural Network Exchange (ONNX) format for subsequent experimental testing. In each control cycle, the measured and computed observations serve as inputs for the ONNX model. The resulting velocity commands are then published at a frequency of 20 Hz. Table 4.6 summarizes the parameter count, Floating-Point Operations

(FLOPs), and the inference time during actual operation of the neural network. The results highlight the remarkable lightweight nature and computational efficiency of the neural network, perfectly fulfilling the control objectives.

Table 4.6: Computational cost in real-time, where the inference time is an average of 100 inferences.

Parameters	**FLOPs**	**Inference Time**
67,842	67,328	97.8 μs

References

[1] B. D. Ziebart, *Modeling purposeful adaptive behavior with the principle of maximum causal entropy*. Carnegie Mellon University, 2010.

[2] T. Haarnoja, H. Tang, P. Abbeel, and S. Levine, "Reinforcement learning with deep energy-based policies," in *International conference on machine learning*, pp. 1352–1361, PMLR, 2017.

[3] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *arXiv preprint arXiv:1801.01290*, 2018.

[4] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, *et al.*, "Soft actor-critic algorithms and applications," *arXiv preprint arXiv:1812.05905*, 2018.

[5] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International conference on machine learning*, pp. 1587–1596, PMLR, 2018.

[6] B. Rubí, B. Morcego, and R. Pérez, "Deep reinforcement learning for quadrotor path following with adaptive velocity," *Autonomous Robots*, vol. 45, no. 1, pp. 119–134, 2021.

[7] E. Meyer, H. Robinson, A. Rasheed, and O. San, "Taming an autonomous surface vehicle for path following and collision avoidance using deep reinforcement learning," *IEEE Access*, vol. 8, pp. 41466–41481, 2020.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*, Santiago, Chile, 07-13 December 2015.

[9] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[10] "Pytorch." https://pytorch.org/. Accessed: 2023-11-16.

CHAPTER 5

Results and analysis

In this chapter, the results based on the methodologies discussed in the preceding chapters are examined. The initial focus lies on scrutinizing the evolution of the policy throughout the training process. Subsequently, a quantitative evaluation of the path following performance in simulation is conducted, leveraging mathematical statistics, for both the conventional pure pursuit method and the innovative approach proposed in this dissertation. This evaluation encompasses a significant number of randomly generated paths.

Following this comprehensive assessment, simulation and experimental results obtained under two distinct test paths are presented. These selected paths include straight segments as well as curves with varying degrees of curvature, representing commonly encountered scenarios in real-world applications.

- **Figure-Eight Curve**

$$\begin{cases} x &= a\sin(\lambda) \\ y &= a\sin(\lambda)\cos(\lambda) \end{cases} \tag{5.0.1}$$

  where a is a constant that dictates the size and shape of the curve, set as 1.0.

- **Lane-Change Curve**

$$\begin{cases} x &= \lambda \\ y &= \frac{b-a}{1+e^{-k(\lambda-c)}} \end{cases} \tag{5.0.2}$$

  where the parameters are specified as: $a = 0.0, b = 1.5, c = 1.5$ and $k = 15.0$, representing the starting point, end point, center of the curve and the steepness of the sigmoid function, respectively.

Notably, despite the provided parameterization equation for the path here, it is still necessary to undergo arc length parameterization, as described in Section 3.1.2. Moreover, based on the Algorithm 2 for generating random paths and multiple tests, as exemplified in Figure 4.5, these two paths are highly unlikely to occur in the training environment.

5.1 Training process

The learning curve and average velocity for the path-following problem are depicted in Figures 5.1 and 5.2, respectively. The solid line represents the average of the 5 trials, while the shaded area corresponds to the confidence interval represented by the standard deviation. In the initial stage, the average velocity learned by the policy is high, but the reward is low. This suggests that the learning strategy prioritizes speed improvement while neglecting the reduction of cross-track error.

Subsequently, a noticeable decrease in average velocity and an increase in reward occur. This is attributed to the influence of pure pursuit, where cross-track error only arises during curve stages. It indicates that the policy begins to learn how to decelerate in response to curves, followed by instances of excessive deceleration.

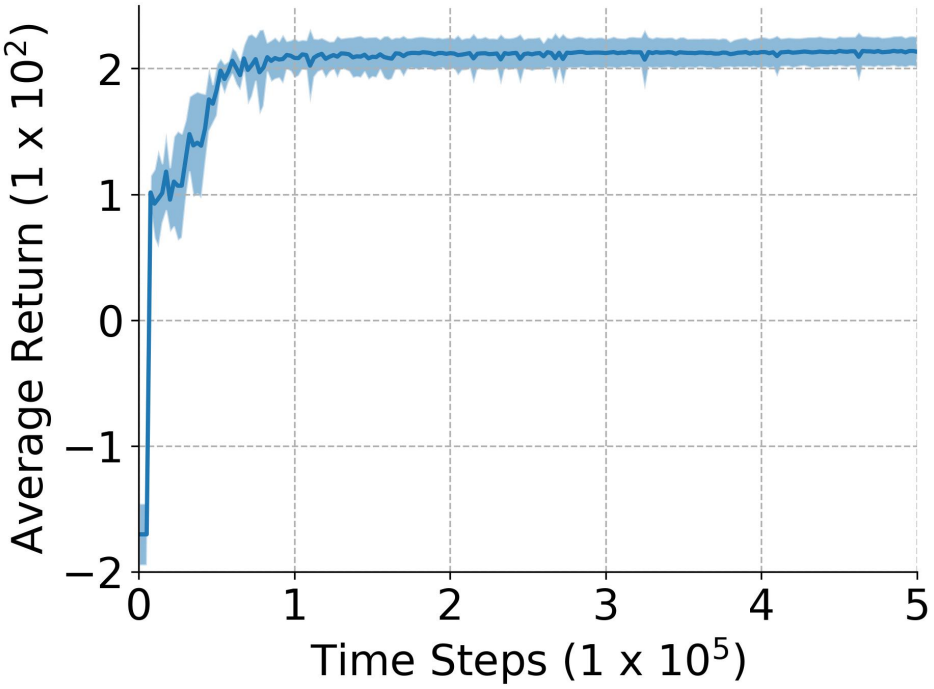

Figure 5.1: Learning curve for path-following over 5 trials.

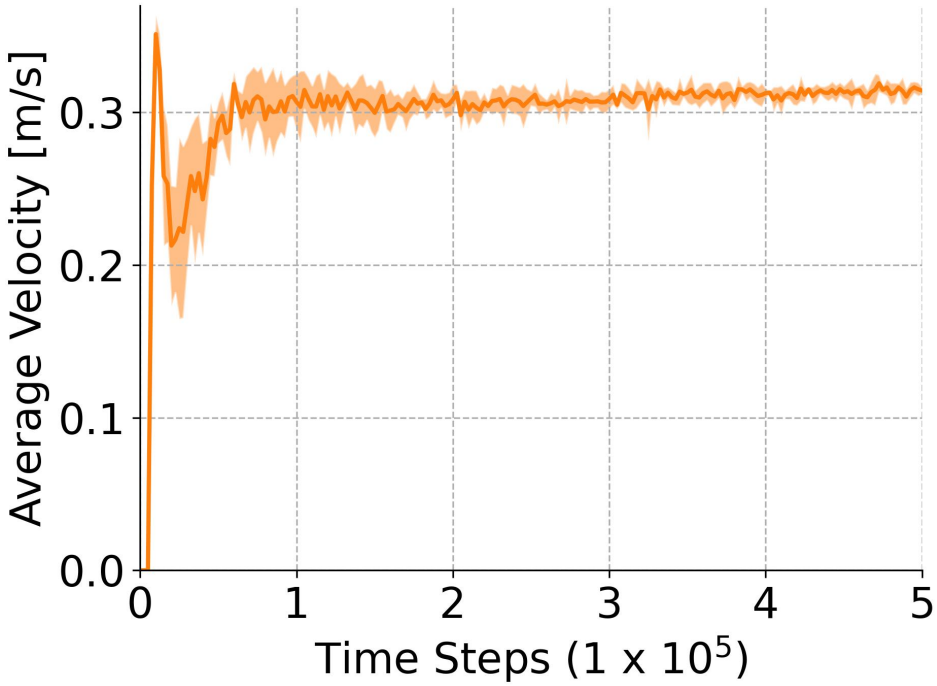

Figure 5.2: Average velocity during training over 5 trials.

Nevertheless, after progressing halfway through the learning process, a more suitable strategy is identified that balances both speed and cross-track error. For all five experiments, the overall training tends to stabilize.

5.2 Simulation results

5.2.1 Quantitative evaluation

In this section, the performance of both the pure pursuit algorithm and the proposed method presented in this dissertation is evaluated across 1000 paths generated using the approach outlined in Algorithm 2. Performance is evaluated by determining whether the cross-track error exceeds a predefined threshold within a limited number of time steps, which is set at 400 steps, consistent with the duration used for training the SAC. In each path-following task, if this threshold is exceeded, the task is marked as a failure, and the execution of the task is terminated.

$$\text{failure rate} = \frac{N_{failure}\ [\text{sample}]}{N_{total}\ [\text{sample}]} \tag{5.2.1}$$

where $N_{failure}$ represents the number of failed samples. Additionally, the overall completion of the path is evaluated at that specific moment.

With the path parameterized by arc length, the completion rate is defined as the ratio of the path parameter at which the robot concludes its trajectory to the parameter of the path's endpoint, as expressed in Equation 5.2.2. This parameterization allows for a meaningful measure of how much of the path has been covered when the robot finishes its trajectory.

$$\text{completion rate} = \frac{\lambda_n \text{ [m]}}{\lambda_{end} \text{ [m]}} \tag{5.2.2}$$

where $\lambda_n = \arg\min_\lambda \|(x(t), y(t)) - (x_r(\lambda), y_r(\lambda))\|^2$ denotes the arc length parameter of the nearest point, as elaborated in Section 3.1.1, while λ_{end} represents the arc length parameter of the path's endpoint.

Firstly, the performance of the pure pursuit algorithm is evaluated. The results for three different thresholds of cross-track error (0.1, 0.2, and 0.3 meters) are presented in Figures 5.3 to 5.5. For each velocity, the failure rate is the ratio of failed samples to the total number of samples and path completion rate represent the averaged outcomes over 1000 paths. The generation of these 1000 paths is ensured with a fixed random seed, maintaining consistency across evaluation environments for each velocity.

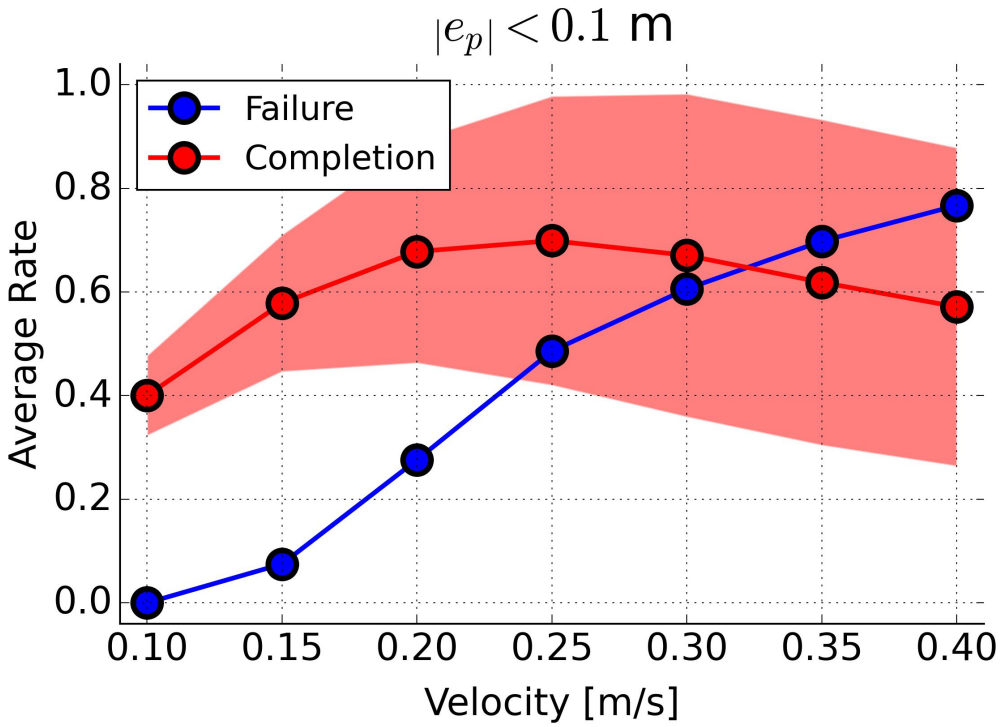

Figure 5.3: The convergence and completion performance of the pure pursuit algorithm over 1000 randomly generated paths. The threshold of cross-track error is 0.1 meter.

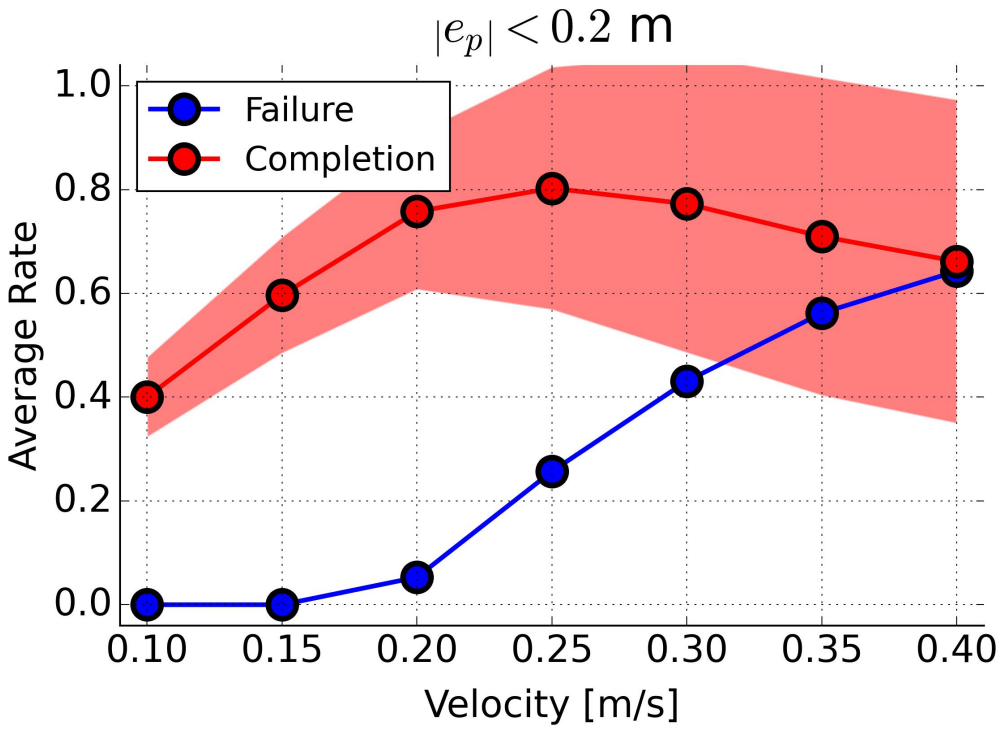

Figure 5.4: The convergence and completion performance of the pure pursuit algorithm over 1000 randomly generated paths. The threshold of cross-track error is 0.2 meter.

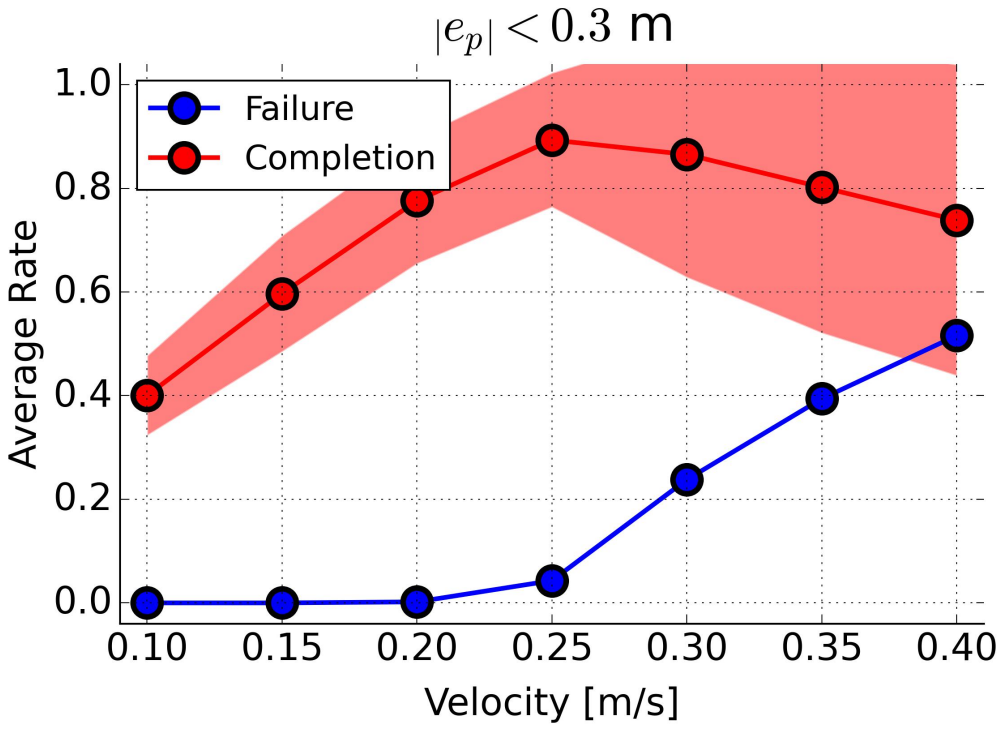

Figure 5.5: The convergence and completion performance of the pure pursuit algorithm over 1000 randomly generated paths. The threshold of cross-track error is 0.3 meter.

From the perspective of failure rate, it is certain that a higher threshold leads to a lower failure rate for any given velocity. Within the same threshold criteria, as velocity increases, the failure rate also increases, primarily due to poor performance at high velocities in turns. Conversely, from the perspective of completion rate, lower velocities may result in minimal failure in path following, but the overall completion rate is not high. Increasing velocity is associated with an improvement in completion rate, but beyond a certain velocity, the completion rate decreases due to premature failures caused by excessive velocity. Moreover, the higher the velocity, the greater the variation in completion rates across different paths. The detailed results can be found in Table 5.1 and Table 5.2.

Table 5.1: Failure rates of pure pursuit controller at various velocities for three cross-track error thresholds.

	0.1 m	**0.2 m**	**0.3 m**
0.10 m/s	0.000	0.000	0.000
0.15 m/s	0.075	0.000	0.000
0.20 m/s	0.276	0.053	0.002
0.25 m/s	0.486	0.257	0.043
0.30 m/s	0.606	0.431	0.238
0.35 m/s	0.698	0.562	0.394
0.40 m/s	0.767	0.643	0.516

Table 5.2: Average completion rates of pure pursuit controller at various velocities for three cross-track error thresholds.

	0.1 m	**0.2 m**	**0.3 m**
0.10 m/s	0.400 ± 0.077	0.400 ± 0.077	0.400 ± 0.077
0.15 m/s	0.578 ± 0.132	0.597 ± 0.112	0.597 ± 0.112
0.20 m/s	0.678 ± 0.215	0.758 ± 0.150	0.776 ± 0.122
0.25 m/s	0.699 ± 0.279	0.803 ± 0.234	0.893 ± 0.130
0.30 m/s	0.671 ± 0.312	0.773 ± 0.287	0.865 ± 0.237
0.35 m/s	0.619 ± 0.314	0.710 ± 0.307	0.802 ± 0.281
0.40 m/s	0.571 ± 0.307	0.662 ± 0.312	0.739 ± 0.300

The five policies trained using the SAC algorithm are evaluated on the same set of 1000 paths. The cross-track error threshold is finely divided into intervals of 0.025 m within the range of 0.1 to 0.3 for better visualization. In Figure 5.6, the solid line represents the average of five policies, and the shaded region indicates half a standard deviation from the average.

The trained policies exhibit significant improvements in both failure rate and completion rate. Within the 0.15 m threshold, there is a low failure rate, and despite failures, the completion rate reaches as high as 0.87. Beyond the 0.15 m threshold, the absence of failures and the approaching path completion rate to 1 imply that under SAC training, the velocity control ensures a reduction in cross-track error while maximizing speed in regions with low curvature. In other words, it allows for higher speeds when possible and slows down where necessary. The reason for not reaching 1 is that some randomly generated paths have a substantial arc length, and they cannot be fully completed within the 400-step limit. At the same time, it is evident that the performance among the five policies is quite similar, indirectly indicating the stability of the learning process and outcomes. More detailed results are summarized in Table 5.3 and Table 5.4.

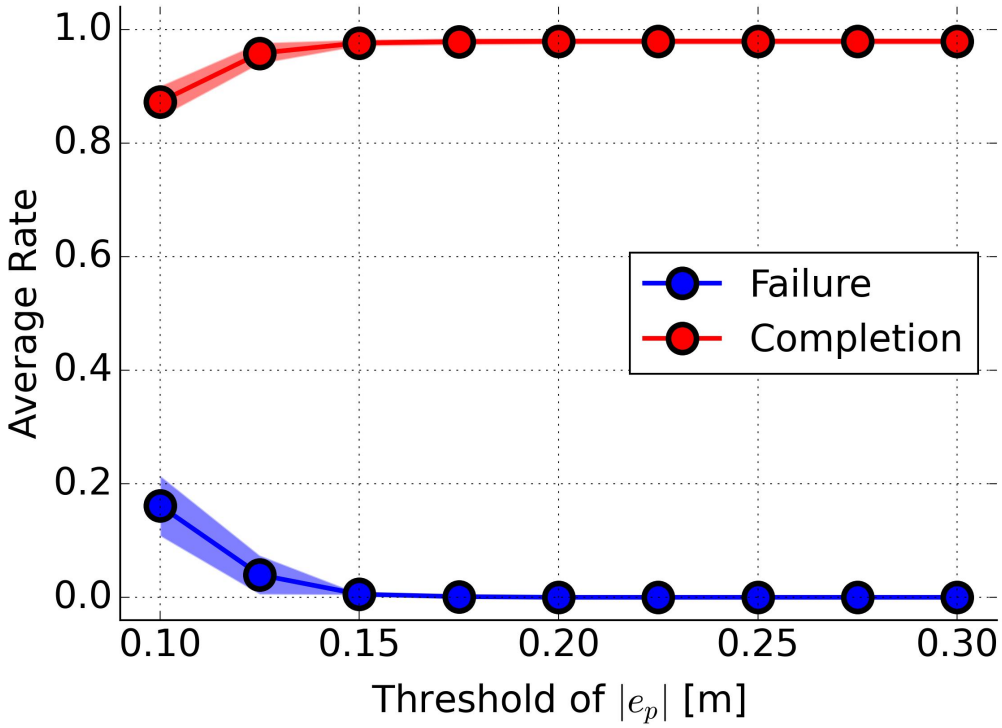

Figure 5.6: The convergence and completion performance of the 5 trained policies over 1000 randomly generated paths.

Table 5.3: Failure rates of SAC controllers for three cross-track error thresholds.

	0.1 m	**0.2 m**	**0.3 m**
policy 1	0.155	0.004	0.000
policy 2	0.114	0.005	0.000
policy 3	0.157	0.005	0.000
policy 4	0.118	0.005	0.000
policy 5	0.262	0.008	0.000
Avg ± Std	0.161 ± 0.053	0.005 ± 0.001	0.000 ± 0.000

Table 5.4: Average Completion rates of SAC controllers for three cross-track error thresholds.

	0.1 m	**0.2 m**	**0.3 m**
policy 1	0.883 ± 0.284	0.984 ± 0.070	0.987 ± 0.070
policy 2	0.891 ± 0.285	0.972 ± 0.124	0.975 ± 0.124
policy 3	0.880 ± 0.286	0.981 ± 0.080	0.985 ± 0.080
policy 4	0.892 ± 0.277	0.975 ± 0.099	0.978 ± 0.099
policy 5	0.817 ± 0.326	0.968 ± 0.140	0.971 ± 0.140
Avg ± Std	0.873 ± 0.028	0.976 ± 0.006	0.979 ± 0.006

The above findings clearly indicate that within the framework of the same pure pursuit control, maintaining a constant speed proves to be suboptimal when confronted with a multitude of unknown paths. While setting a fixed speed for specific paths or designing a path-based speed command is feasible, it necessitates redesigning for each new path. Furthermore, real-world testing and experiments add further challenges.

In addressing these issues, the proposed method in this dissertation proves to be a robust solution, demonstrating significant improvements in both failure rate and completion rate. This method not only overcomes the limitations associated with maintaining a static speed but also showcases its adaptability to diverse and dynamic path scenarios. This adaptability positions the proposed method as a promising solution for autonomous navigation in complex environments. Next, the proposed strategy on two distinct paths is evaluated with simulation results analyzed first, followed by corresponding experimental results.

5.2.2 Path convergence and adaptive velocity

Due to the SAC's reward function being set to encourage velocity maximization, in the comparison, the case where pure pursuit is instructed with the maximum velocity is evaluated. Meanwhile, as indicated by the results in the previous section, a fixed velocity is not optimal for unknown paths. Here, these specific paths are still considered as unknown, highlighting that the choice of velocity command for pure pursuit may not be crucial. The emphasis is on analyzing the cross-track error and velocity performance of the proposed method in this dissertation within the path.

(a) Figure-Eight Curve

The trajectories and velocities of the pure pursuit algorithm and the pure pursuit algorithm for steering control combined with the SAC algorithm for speed control, as proposed in this dissertation for tracking the figure-eight curve, are illustrated in Figure 5.7 and 5.8. The initial pose is set to a randomly generated value of $[0.009, -0.044, 0.736]$.

The results of the pure pursuit algorithm show a velocity command at the maximum value of 0.4 m/s, indicating that even though pure pursuit involves anticipatory turning based on a look-ahead viewpoint, its performance in curved sections is poor. On the other hand, the proposed method, specifically policy 4 in the figure, due to adaptive velocity adjustment, can turn with minimal error, significantly improving the success rate of path following and achieving a high overall average velocity.

Specific comparative results are summarized in Table 5.5, where $\bar{e_p}$ represents the root mean squared cross-track error, demonstrating consistent path convergence performance across the five trained policies. The following section of the dissertation follows this presentation.

Table 5.5: Results for one lap of the figure-eight path in simulation.

	$\bar{e_p}$ **[m]**	$\lvert e_p \rvert_{max}$ **[m]**	$\bar{v}$ **[m/s]**
pure pursuit	0.0593	0.1311	0.4000
policy 1	0.0117	0.0385	0.2868
policy 2	0.0118	0.0385	0.2793
policy 3	0.0119	0.0384	0.2819
policy 4	0.0115	0.0385	0.2688
policy 5	0.0121	0.0385	0.2958

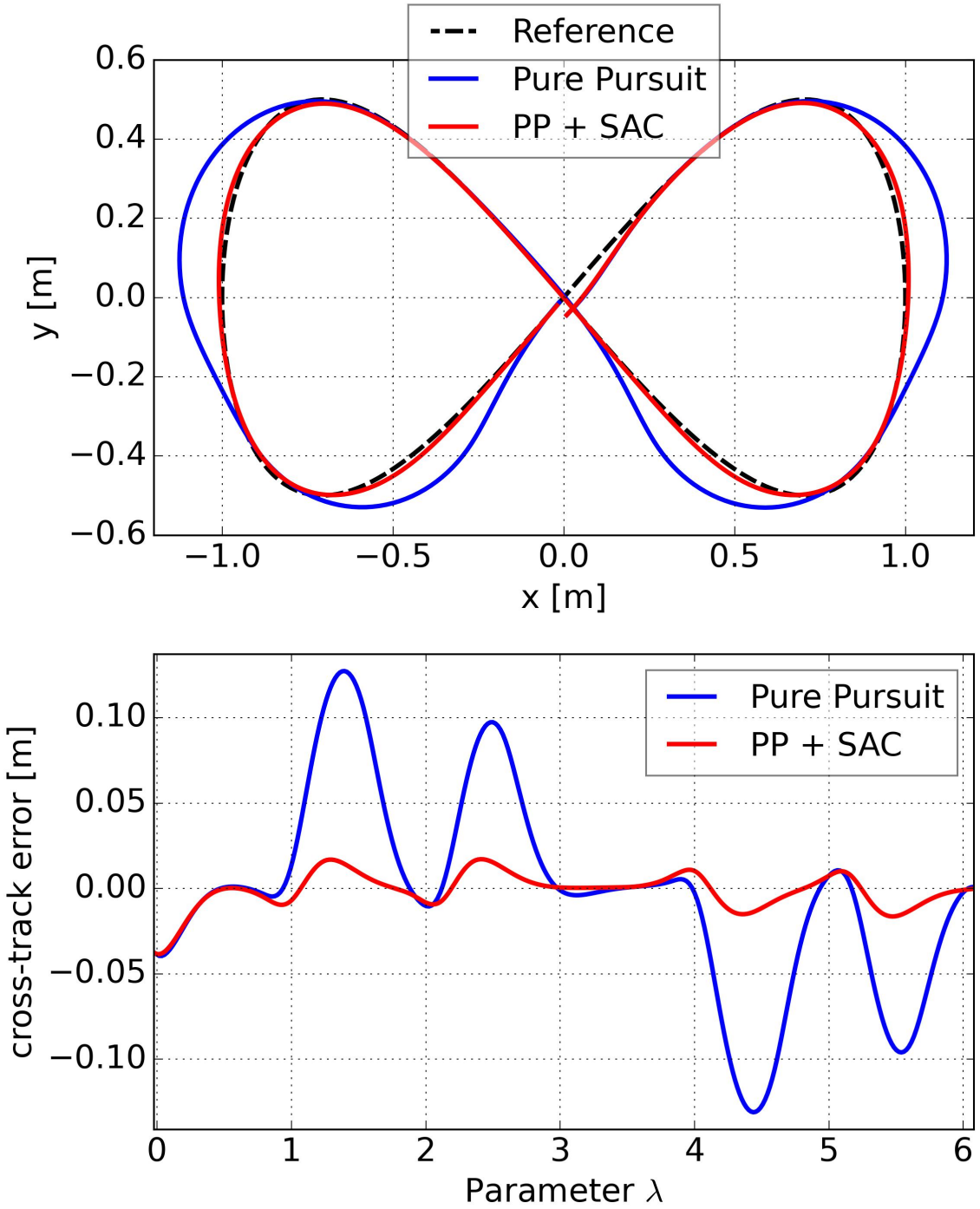

Figure 5.7: Trajectory and cross-track error comparison for the figure-eight path in simulation.

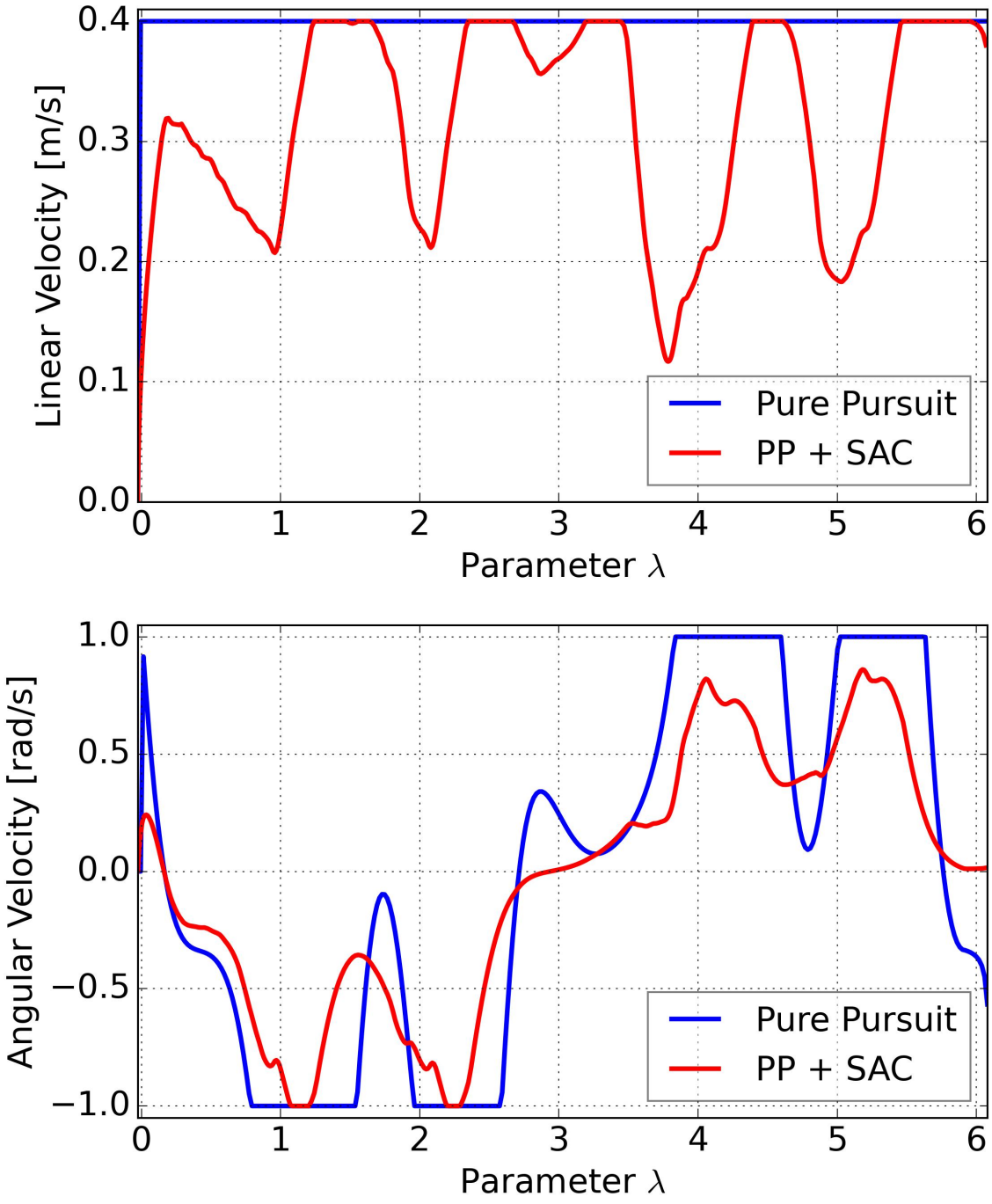

Figure 5.8: Velocity comparison for the figure-eight path in simulation.

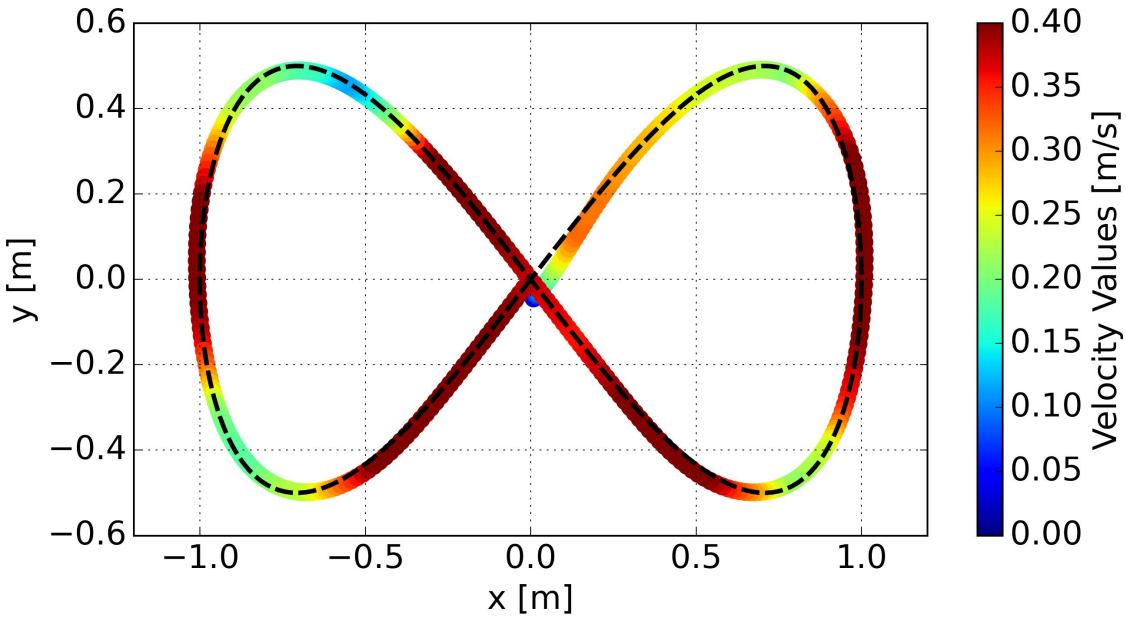

Figure 5.9: Linear velocity variation along the trajectory for the figure-eight path in simulation.

The comparison of linear velocity and angular velocity corresponding to Figure 5.7 is depicted in Figure 5.8. It is evident that the proposed method can adaptively adjust velocity in a smooth way, decelerating before entering a curve, accelerating when exiting a curve rapidly, and maintaining maximum velocity on relatively straight segments. A more intuitive visualization is presented in a trajectory scatter plot plotted using velocity magnitude, as shown in Figure 5.9. Smooth variations in linear velocity are evident in this representation. In contrast to the angular velocity of the pure pursuit algorithm, the proposed method avoids the saturation region of angular velocity.

(b) Lane-Change Curve

In this part, following the same evaluation criteria as the previous path, another specific trajectory is assessed. The trajectories and velocities of both approaches for tracking the lane-change curve are illustrated in Figure 5.10 and 5.11. The initial pose is set to a randomly generated value of $[0.090, -0.055, -0.034]$.

The fixed velocity of the pure pursuit algorithm exhibits noticeable overshooting when encountering sharp turns and requires some time to return to the path. On the other hand, the proposed method, specifically policy 1 in the figure, handles sharp turns gracefully without deviating from the path, ensuring the success of the pure pursuit algorithm responsible for steering control. Specific comparative results are summarized in Table 5.6.

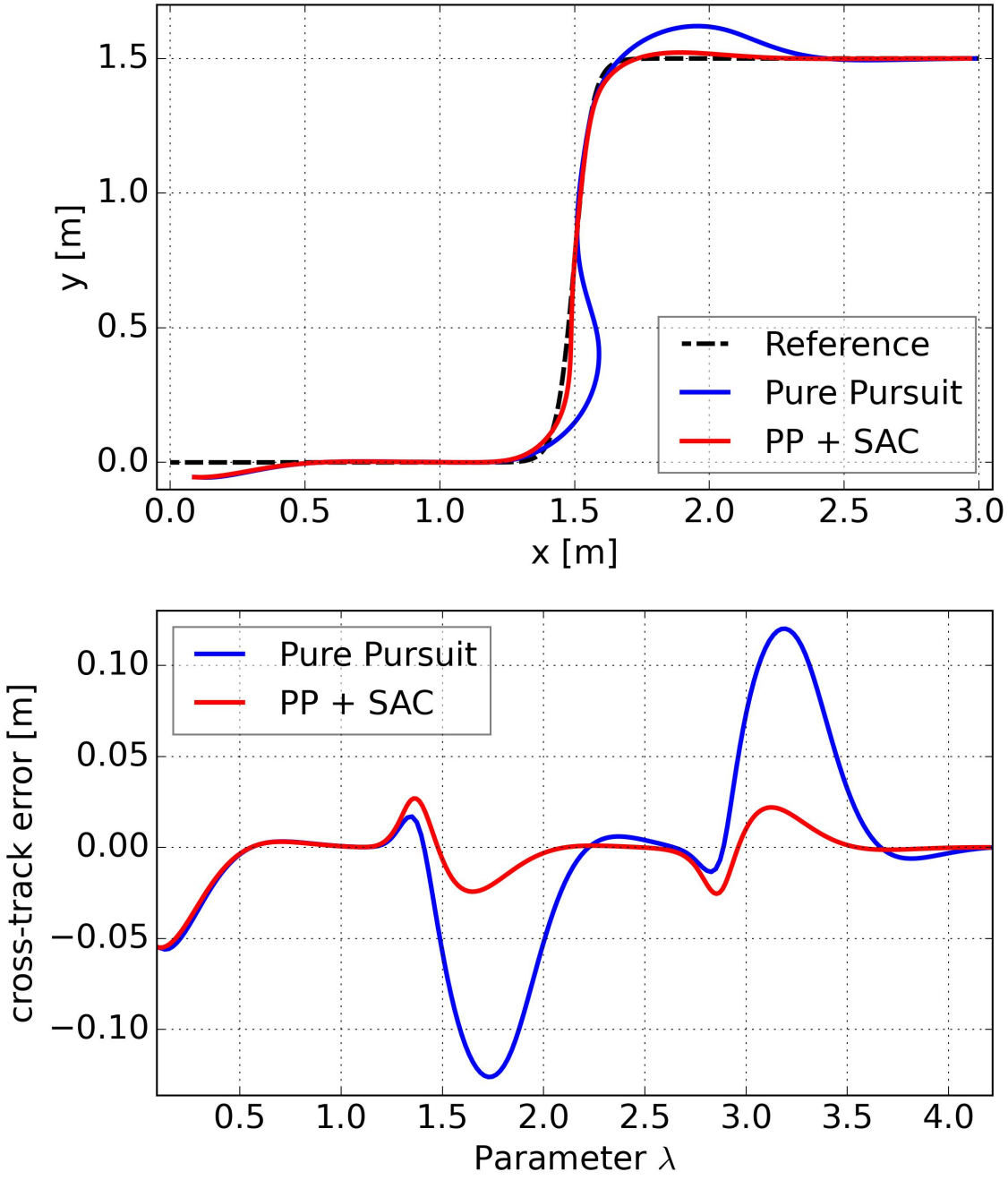

Figure 5.10: Trajectory and cross-track error comparison for the figure-eight path in simulation.

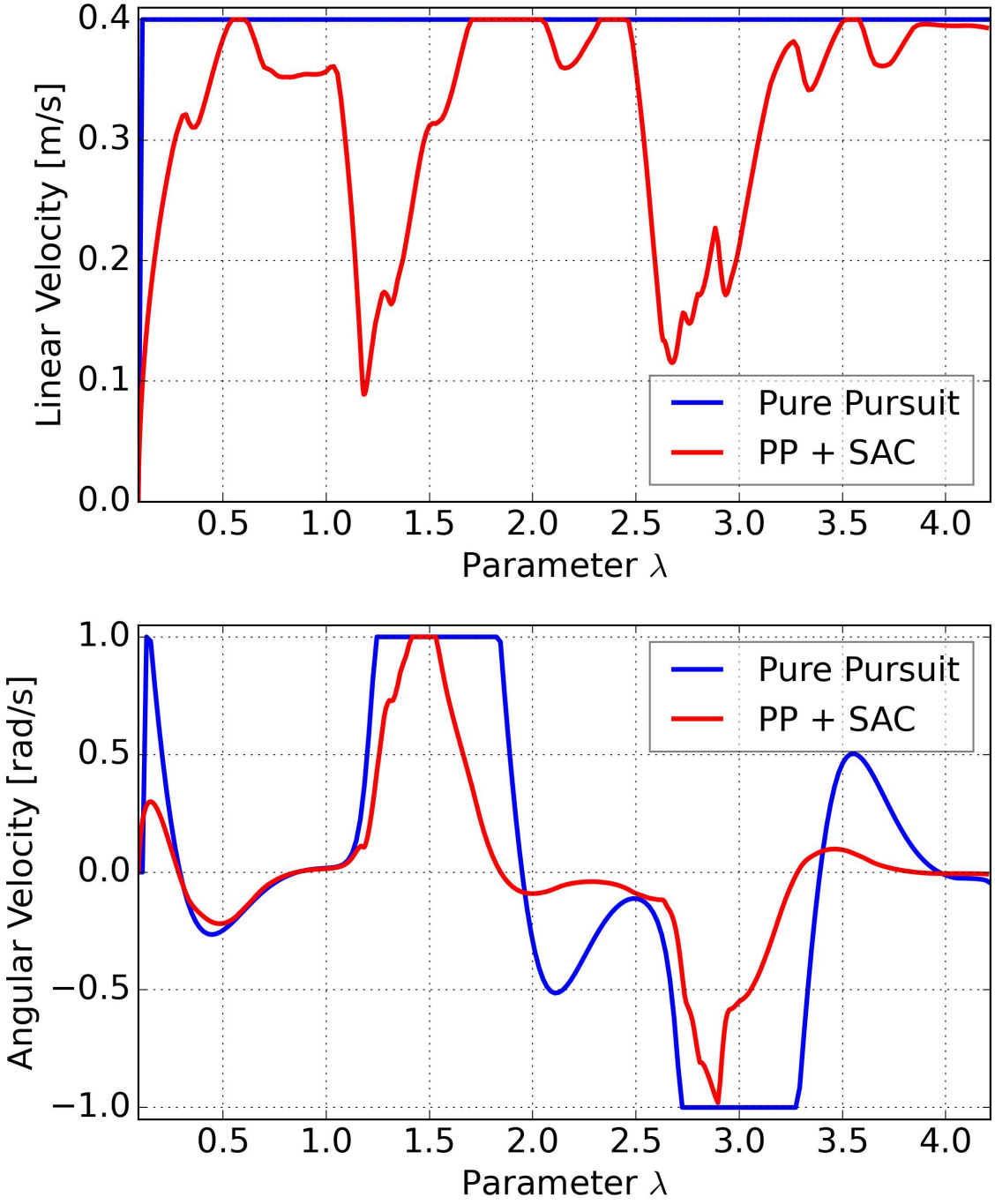

Figure 5.11: Velocity comparison for the lane-change path in simulation.

Table 5.6: Results of the lane-change path in simulation.

	$\bar{e_p}$ [m]	$\|e_p\|_{max}$ [m]	$\bar{v}$ [m/s]
pure pursuit	0.0525	0.1262	0.4000
policy 1	0.0181	0.0551	0.2854
policy 2	0.0184	0.0551	0.2844
policy 3	0.0185	0.0551	0.2896
policy 4	0.0187	0.0551	0.2762
policy 5	0.0186	0.0551	0.2875

The comparison of linear velocity and angular velocity corresponding to Figure 5.10 is illustrated in Figure 5.11. It can be observed that, for sharp turns, the trained policy is capable of decelerating to very low values, in this case, approximately 0.06 m/s. With minimal cross-track error, it navigates through the sharp turn and quickly returns to the maximum velocity. A clearer visualization of velocity variation is presented in a trajectory scatter plot plotted using velocity magnitude, as depicted in Figure 5.12. Even under sharp turns, smooth variations in linear velocity can be achieved, due to the policy's action range. Similarly, the proposed method avoids the saturation region of angular velocity.

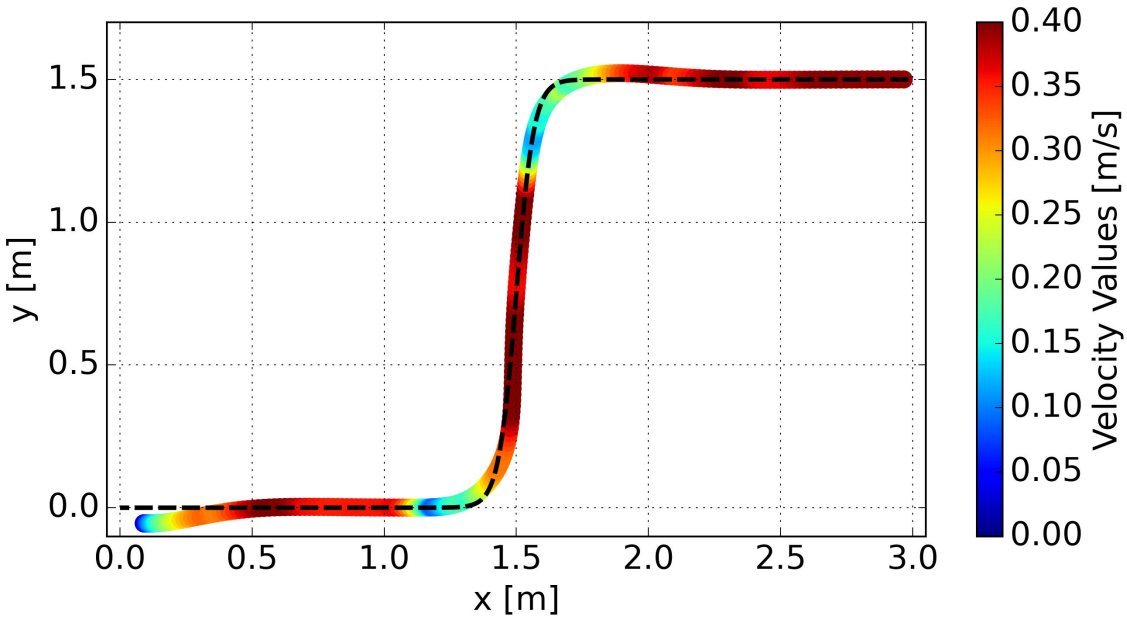

Figure 5.12: Linear velocity variation along the trajectory for the lane-change path in simulation.

5.3 Experimental results

In this section, the evaluation focuses on assessing the performance of the trained policies in real-world path following. The evaluation is conducted on two specific paths, mirroring the paths discussed in the previous section for consistency. The configuration of the robot used in the experiment is introduced first, followed by a brief discussion of the implementation of mapping and localization, with a focus on the results of path-following performance.

5.3.1 Experimental setup

The experimental nonholonomic wheeled mobile robot, as depicted in Figure 5.13, features the placement of the active wheel at the center of the chassis, with an additional omnidirectional wheel at both the front and rear. This configuration is intended to both enhance maneuverability and simplify control. With an active wheel strategically positioned in the center, the robot gains the capability to rotate more freely in narrow spaces, making it well-suited for applications that necessitate frequent steering or operation within confined environments, such as indoor navigation and robot obstacle avoidance. Moreover, the design effectively minimizes the impact of wheel spacing on the vehicle's movement, thereby enhancing stability in complex environments. The wheelbase, measured and calibrated to 0.172 m, corresponds precisely to the parameter employed in simulation. The overall dimensions of the robot measure $216 \times 216 \times 171$ mm, and it weighs approximately 2.5 kg.

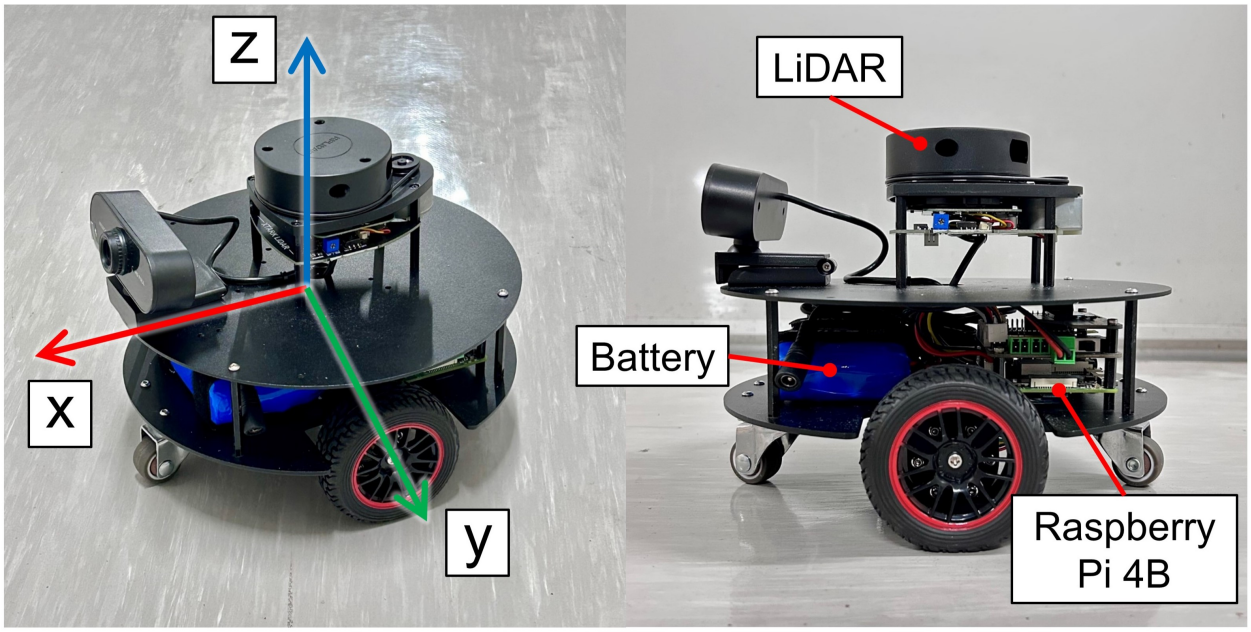

Figure 5.13: Configuration of the experimental nonholonomic wheeled mobile robot.

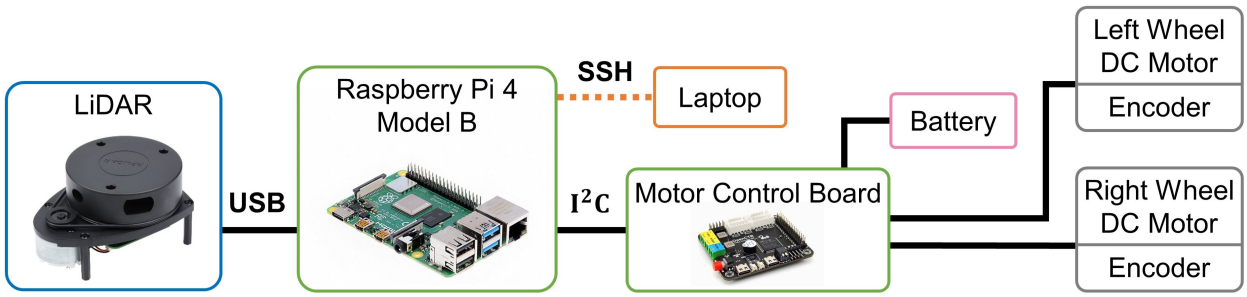

Figure 5.14: Diagram of the robot's connectivity and communication.

At the top layer of the robot, a 360 Degree Laser Scanner, utilizing Laser imaging, Detection, and Ranging (LiDAR), is configured for subsequent mapping and localization functions. LiDAR is a method for determining ranges by targeting an object or a surface with a laser and measuring the time for the reflected light to return to the receiver. This setup serves the purpose of perceiving the robot's pose. The middle layer comprises the main control Raspberry Pi 4 Model B [1], motor control module, and battery. Although an RGB camera is also configured, it was not utilized in this dissertation.

The Raspberry Pi operates on the Ubuntu 20.04 system [2] to support the execution of the Robot Operating System (ROS) [3], a set of software libraries and tools designed to facilitate the development of robot applications. This content specifically utilizes the ROS Noetic version. The robot's connectivity and communication are illustrated in Figure 5.14, where communication with the robot's main body is established through the Secure Shell Protocol (SSH). This necessitates that the laptop and the Raspberry Pi system be within the same Local Area Network (LAN) for effective communication, with the ROS facilitating the connection and interaction.

The primary focus of this dissertation is path following; therefore, the methods for obtaining the real-world pose of the robot are briefly summarized. Given that a robot equipped with LiDAR has already been configured, the experiments are conducted indoors. The mapping component utilizes Gmapping [4], a Simultaneous Localization and Mapping (SLAM) algorithm based on 2D laser range data that employs Rao-Blackwellized Particle Filters (RBPF) for constructing 2D grid maps. The advantages of Gmapping include real-time construction of indoor environment maps, low computational requirements for small scenes, high map accuracy, and a relatively low demand on the scanning frequency of the LiDAR. The experimental site and the constructed grid map are shown in Figure 5.15. After obtaining the map, during the execution of the path following algorithm, the robot's pose is determined

Figure 5.15: Experimental site and mapping results.

using Adaptive Monte Carlo Localization (AMCL) [5], which employs a particle filter to track the robot's pose against a known map. Due to the algorithm's favorable performance in small-scale environments, it is confident in asserting that the localization is relatively accurate.

5.3.2 Sim-to-real transfer

Due to limitations in collecting real-world data, a simulation environment is utilized to train the path following policy. The simulation environment not only offers the possibility of infinite data sources but also avoids safety issues associated with real robots. However, the gap between simulation and the real world weakens the performance of the policy on actual robots. Possible solutions include establishing a realistically accurate simulator through system identification methods or highly randomizing the simulation to cover the data distribution of the real world [6].

In this dissertation, to avoid the complexities and inaccuracies of system identification, perfect identical velocity tracking is assumed, where wheel speed equals the commanded speed, enabling rapid convergence during training. Additionally, systems with time delay effects can lead to incorrect and untimely rewards for actions, resulting in non-convergence or failure of training [7].

Here, a simple approach to compensate for the gap between simulation and reality by

treating wheel speed as a first order lag system with delay is discussed. The velocities under different conditions during the tracking of the figure-eight path are illustrated in Figures 5.16 and 5.17, where τ represents the time constant in second and k represents the delayed samples. In the figure, the policies are the same; the difference lies in the wheel speed transfer characteristics of the robot.

As the time constant and delay increase, it can be observed that not only does the response slow down, but the average speed also decreases, making it difficult to reach the maximum speed. It is conceivable that, given a scenario where the mobile robot maintains low cross-track error and does not require steering adjustments, there would be sufficient time to reach maximum speed. However, in cases like the figure-eight path in this example, the path length is insufficient for this purpose. This phenomenon is also mentioned in [8], and one straightforward approach is to scale the action size to compensate. As shown in Figure 5.18, compensation result under one condition is presented, where the action output by the policy, i.e., the magnitude of acceleration, is scaled up by a factor of 2.5. The main purpose is to achieve the maximum speed. It can be seen that this method is simple and effective, allowing for an expanded range of achievable speeds.

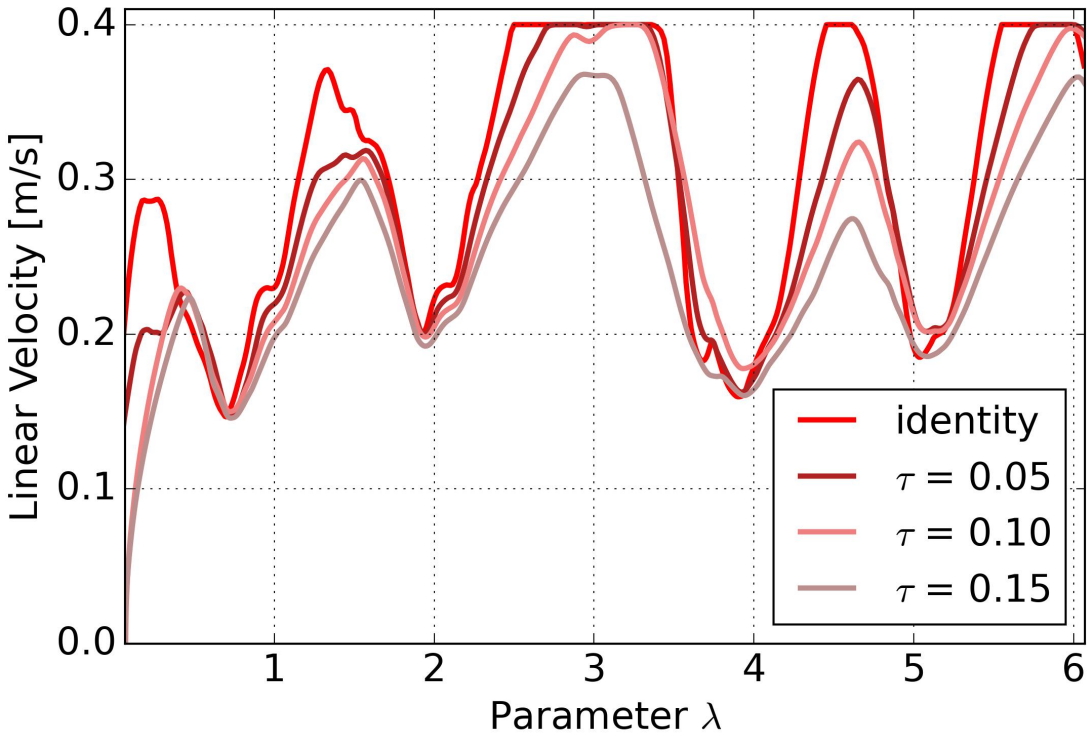

Figure 5.16: Comparisons of velocities under different time constants.

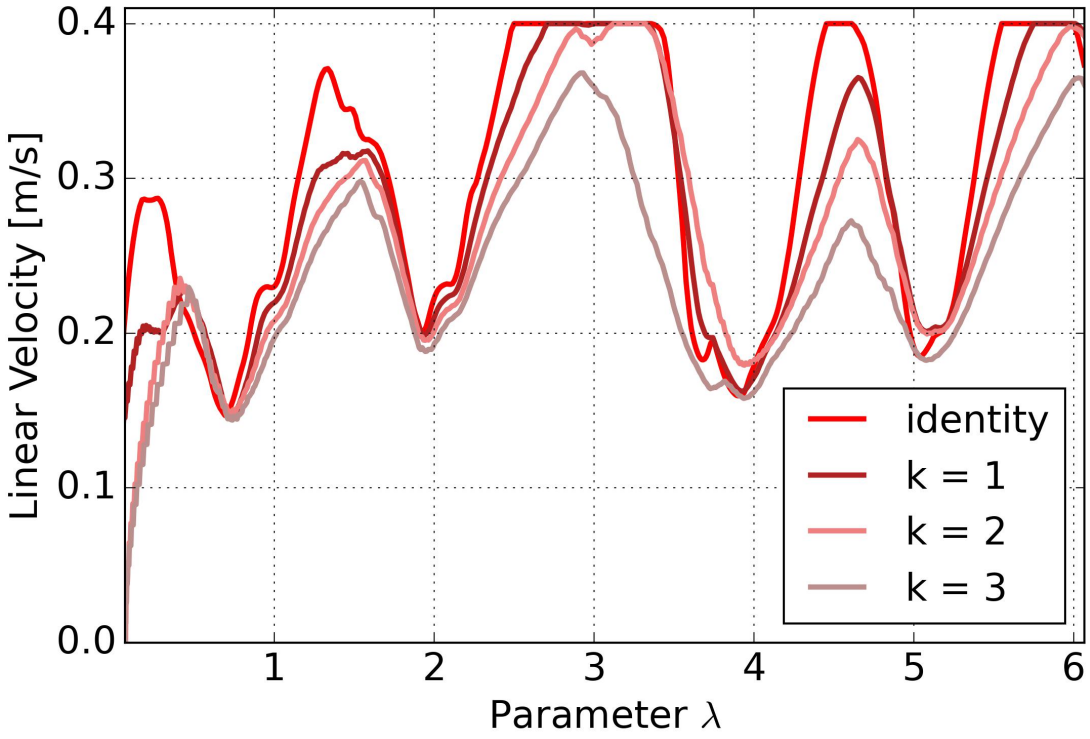

Figure 5.17: Comparisons of velocities under different time delays.

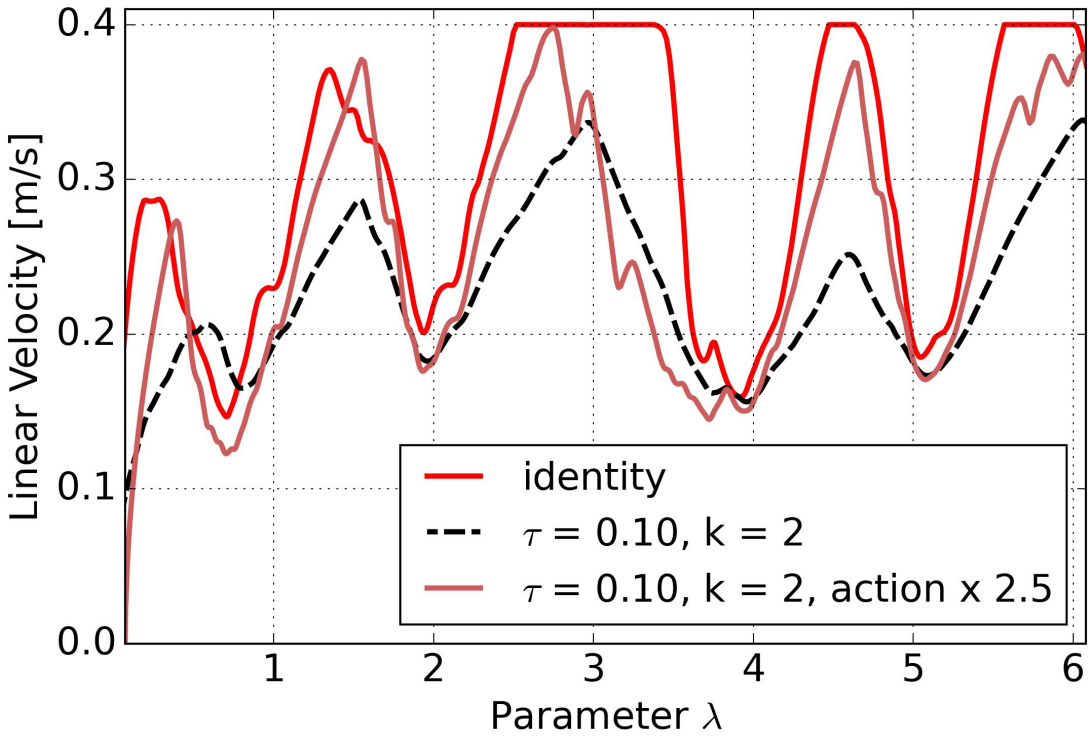

Figure 5.18: Compensation example with action scaling.

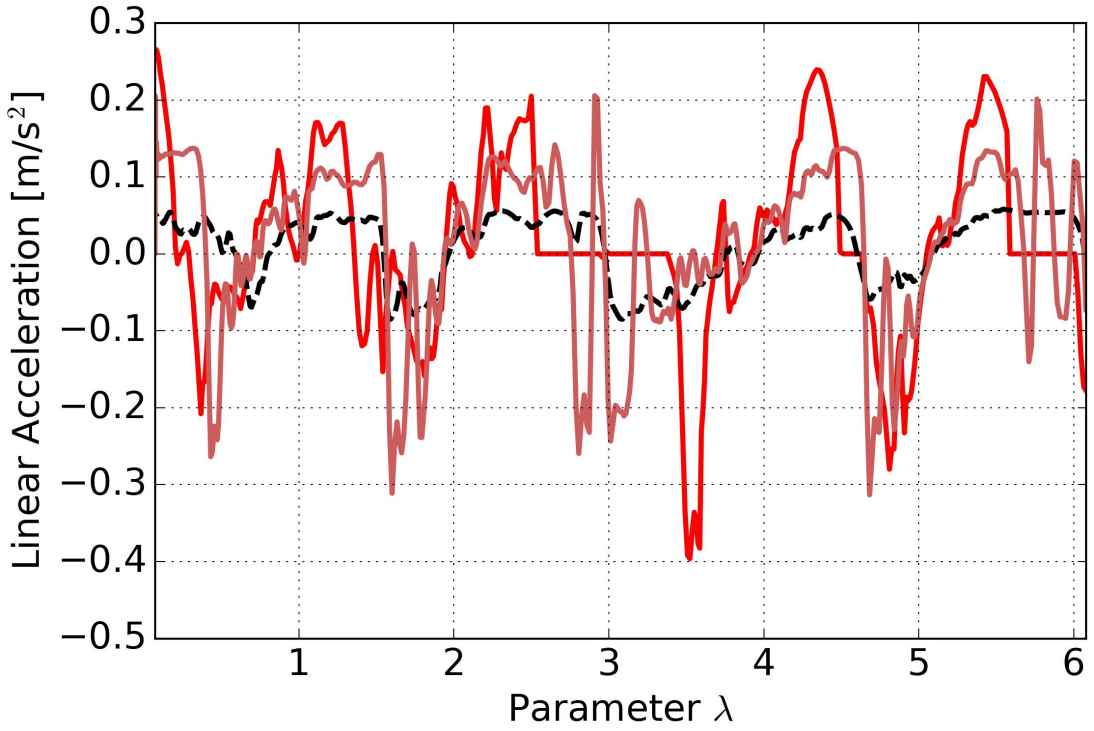

Figure 5.19: The accelerations corresponding to Figure 5.18.

Furthermore, due to delay and slow response, the actual acceleration after scaling the action did not exceed the expected range during the training, as depicted in Figure 5.19. Highlighting the significance, it should be emphasized that overly aggressive scaling may induce speed oscillations, as observed by the trials.

The trajectories, errors, and velocities of the directly used policy are meticulously compared with those of pure pursuit and illustrated in Figures 5.20 and 5.21. The depicted results specifically correspond to policy 1. It is worth noting that analogous patterns to the aforementioned phenomena are evident. Despite the close resemblance in path following, braking, and acceleration behaviors to the simulation, the overall speed is noticeably lower due to response time and a inherent delay. Notably, in sections with curves, the policy exhibits a cautious behavior, decelerating to an extremely low velocity, and it refrains from maintaining high speeds even on straight segments.

Through experimentation, it has been determined that a scaling factor of 2.2 is a reasonable value to make the speed performance more closely resemble the simulation. The results of the next section's experiments for both paths are based on this coefficient.

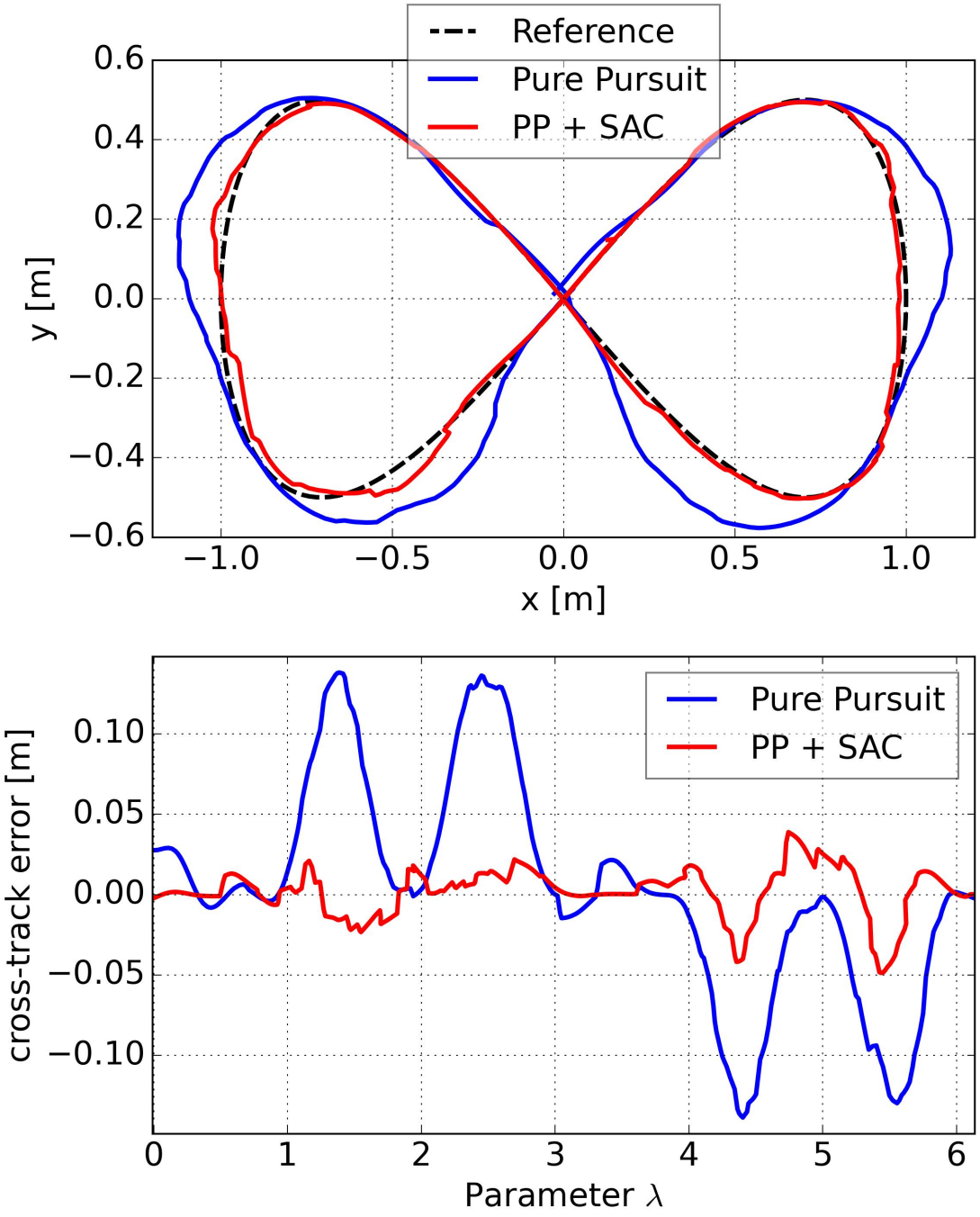

Figure 5.20: Comparisons of velocities under different time constants.

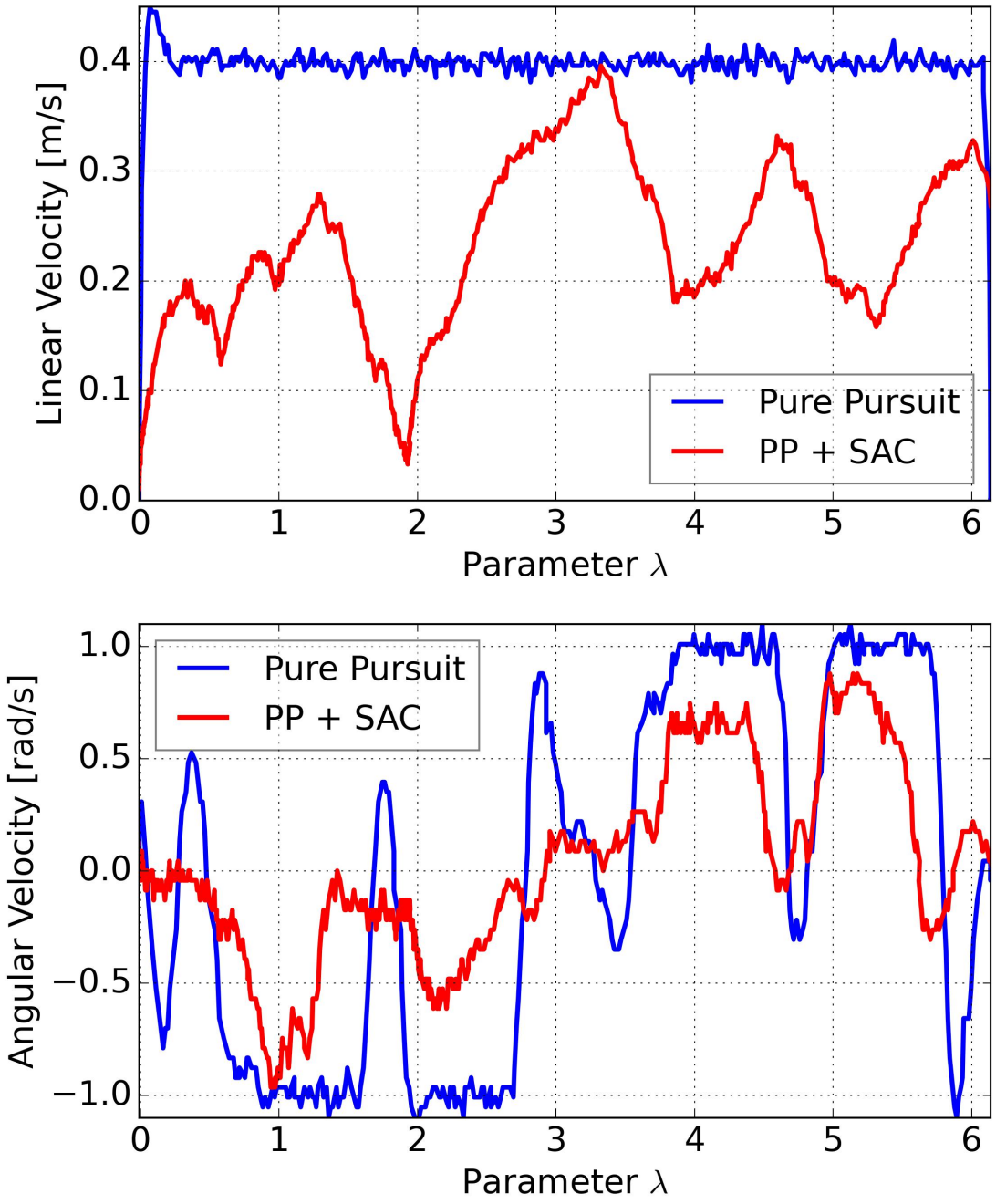

Figure 5.21: Comparisons of velocities under different time delays.

5.3.3 Path convergence and adaptive velocity

(a) Figure-Eight Curve

The trajectory results of the experiment for tracking the eight-shaped path are shown in Figure 5.22, demonstrating consistent results with the simulation. Specifically, the results in the figure represent the outcomes of policy 2. Relatively speaking, the acceleration of the policy in the experiment appears more cautious, attributed to the transfer relationship and delay between input and output of motor wheels. Setting aside this aspect, the consistency of the policy has been validated, and a noticeable deceleration effect is observed in the curved segments. The results of the five experiments are summarized in Table 5.7. The experimental results for the five policies consistently outperform traditional pure pursuit in both RMSE and maximum cross-track error. However, due to the slower velocity on straight segments in the middle portion compared to the simulated speed, there is a slight overall decrease in average velocity.

Table 5.7: Results for one lap of the figure-eight path in experiment.

	$\bar{e_p}$ **[m]**	$\|e_p\|_{max}$ **[m]**	$\bar{v}$ **[m/s]**
pure pursuit	0.0674	0.1386	0.3949
policy 1	0.0207	0.0493	0.2317
policy 2	0.0146	0.0410	0.2257
policy 3	0.0108	0.0314	0.2444
policy 4	0.0134	0.0376	0.2097
policy 5	0.0117	0.0368	0.2090

The comparison between linear velocity and angular velocity is depicted in Figure 5.23. It is noticeable that, during sharp turns, the trained policy exhibits the ability to decelerate to remarkably low values. However, on relatively straight segments, it doesn't fully reach the maximum velocity as observed in the simulation, mainly due to input-output delay. The changes in acceleration corresponding to Figure 5.23 are illustrated in Figure 5.24, with the acceleration slightly surpassing the expected range. A more explicit representation of velocity variation is showcased in a trajectory scatter plot generated using velocity magnitude, illustrated in Figure 5.25. Likewise, the proposed approach steers clear of the saturation region of angular velocity, mirroring the behavior observed in the simulation.

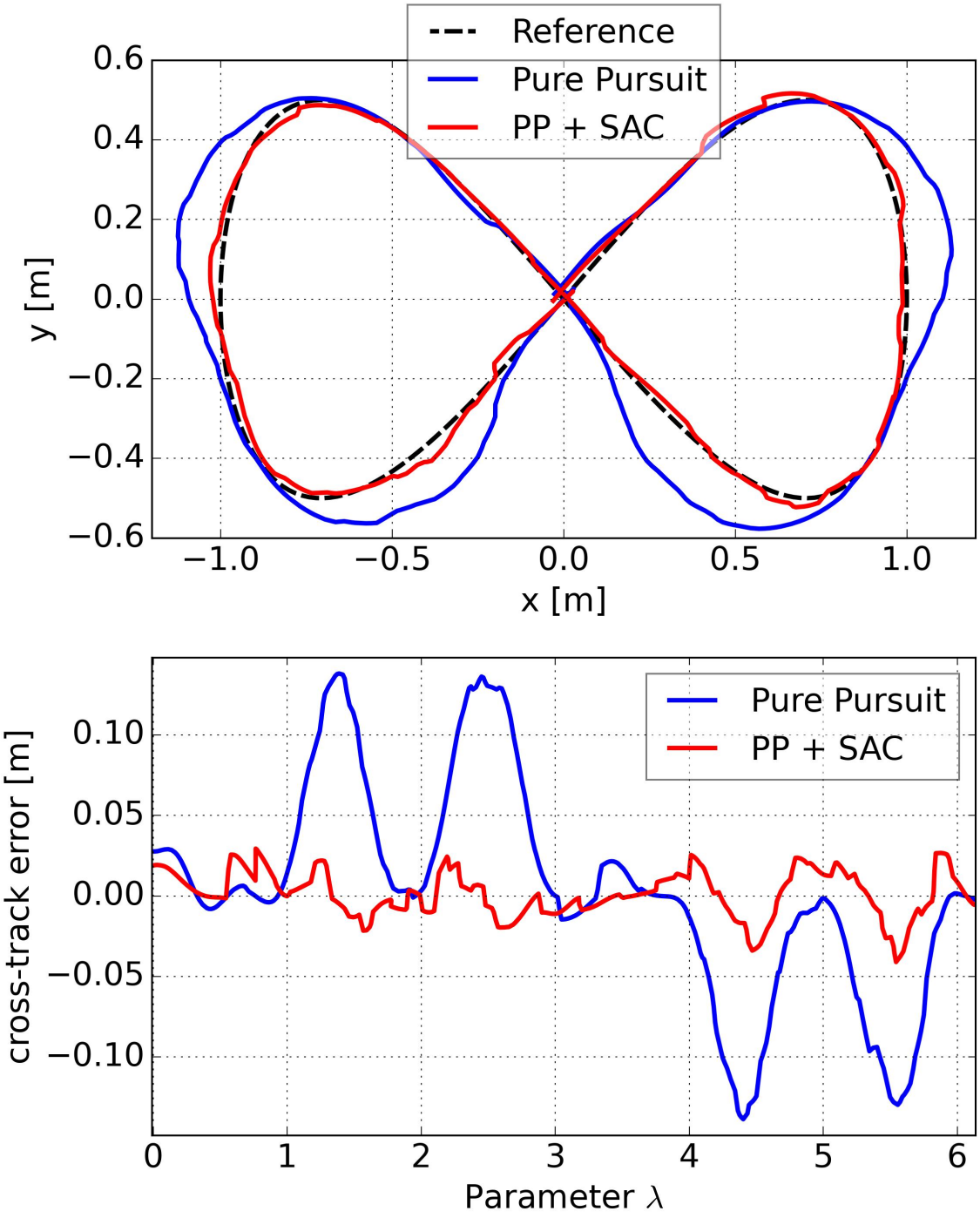

Figure 5.22: Trajectory and cross-track error comparison for the figure-eight path in experiment.

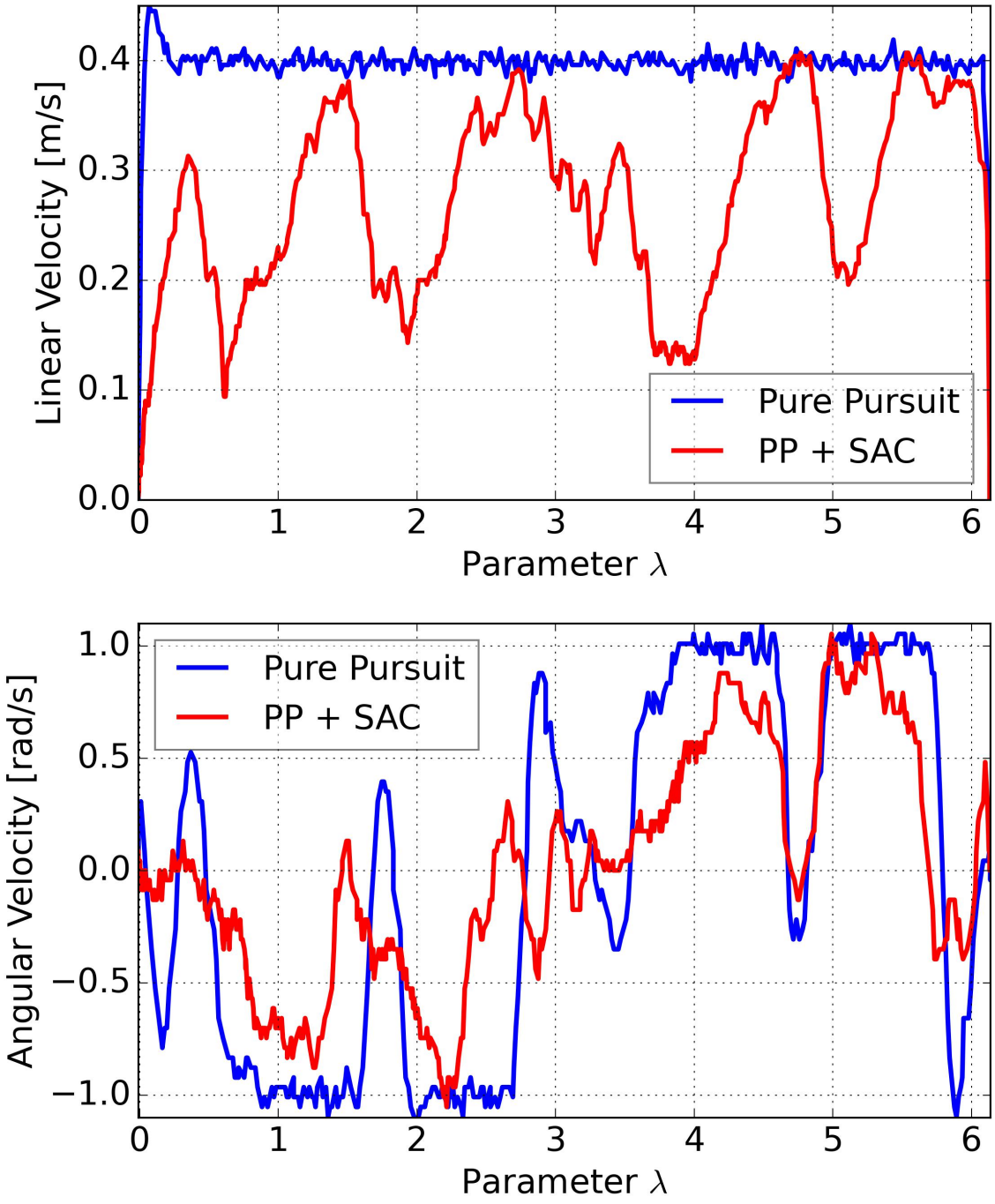

Figure 5.23: Velocity comparison for the figure-eight path in experiment.

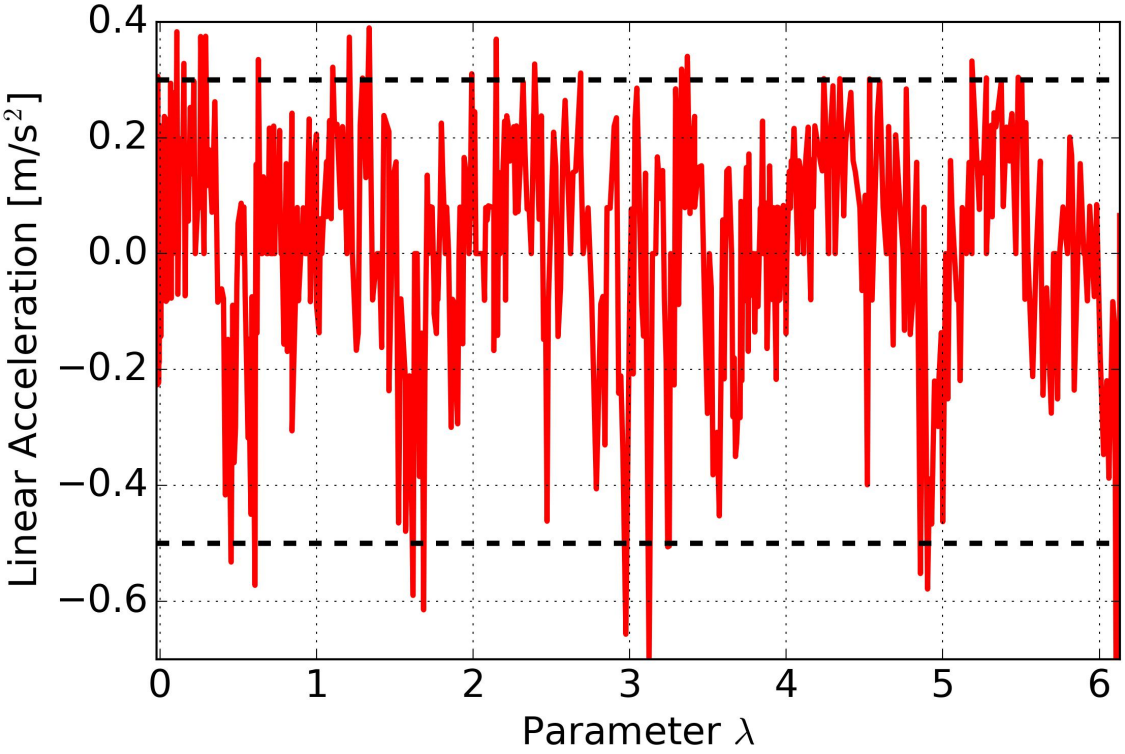

Figure 5.24: The accelerations corresponding to Figure 5.23.

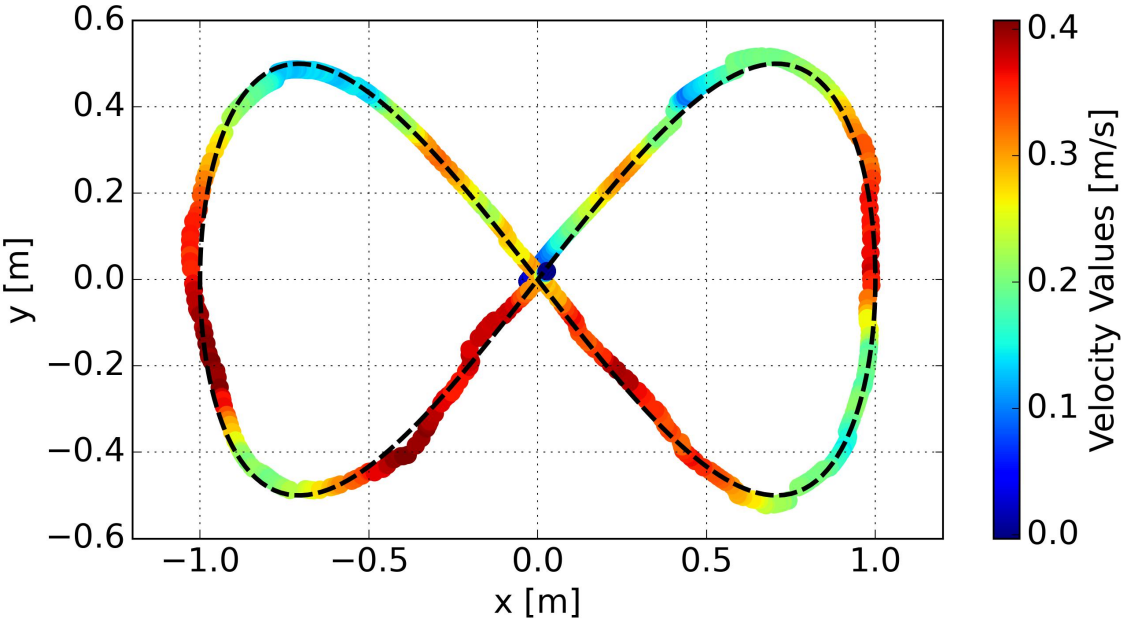

Figure 5.25: Linear velocity variation along the trajectory for the figure-eight path in experiment.

(b) Lane-Change Curve

The trajectories from both approaches in tracking the lane-change curve are depicted in Figure 5.26. This experiment confirms a strong alignment with the simulation, emphasizing the robustness of the proposed policies across diverse scenarios. Notably, the pure pursuit algorithm tends to overshoot during sharp turns, requiring a subsequent correction to realign with the path. In contrast, specifically policy 3 (as seen in the figure) adeptly maneuvers through sharp turns without deviating. Detailed comparative results are outlined in Table 5.8. Figure 5.27 illustrates the comparison of linear and angular velocity, with the corresponding acceleration variations depicted in Figure 5.28. A clearer representation of velocity variation is provided in Figure 5.29.

Table 5.8: Results of the lane-change path in experiment.

	$\bar{e_p}$ **[m]**	$\lvert e_p \rvert_{max}$ **[m]**	$\bar{v}$ **[m/s]**
pure pursuit	0.0559	0.1505	0.3931
policy 1	0.0131	0.0317	0.2488
policy 2	0.0273	0.0702	0.2305
policy 3	0.0166	0.0360	0.2494
policy 4	0.0138	0.0416	0.2296
policy 5	0.0181	0.0583	0.2176

In this trajectory, due to the presence of more straight segments compared to the figure-eight path, the average velocity does not exhibit a significant decrease compared to the simulation. Overall, policies trained based on an ideal model with an identity transform show approximately similar effects to the simulation when experimentally scaling the action size. Moreover, this scaling parameter is applicable to different paths, eliminating the need for re-experimentation for each path. The tracking performance in comparison to the figure-eight curve is consistent and significantly superior to pure pursuit control.

Both in simulation and experiments, the policies demonstrate the ability to appropriately decelerate before encountering a curve and accelerate as much as possible after navigating the curve, validating the feasibility and superiority of the proposed approach. The acceleration is only slightly exceeding the specified range, indicating a minor deviation that could be addressed through fine-tuning or adjustment of the scale parameter.

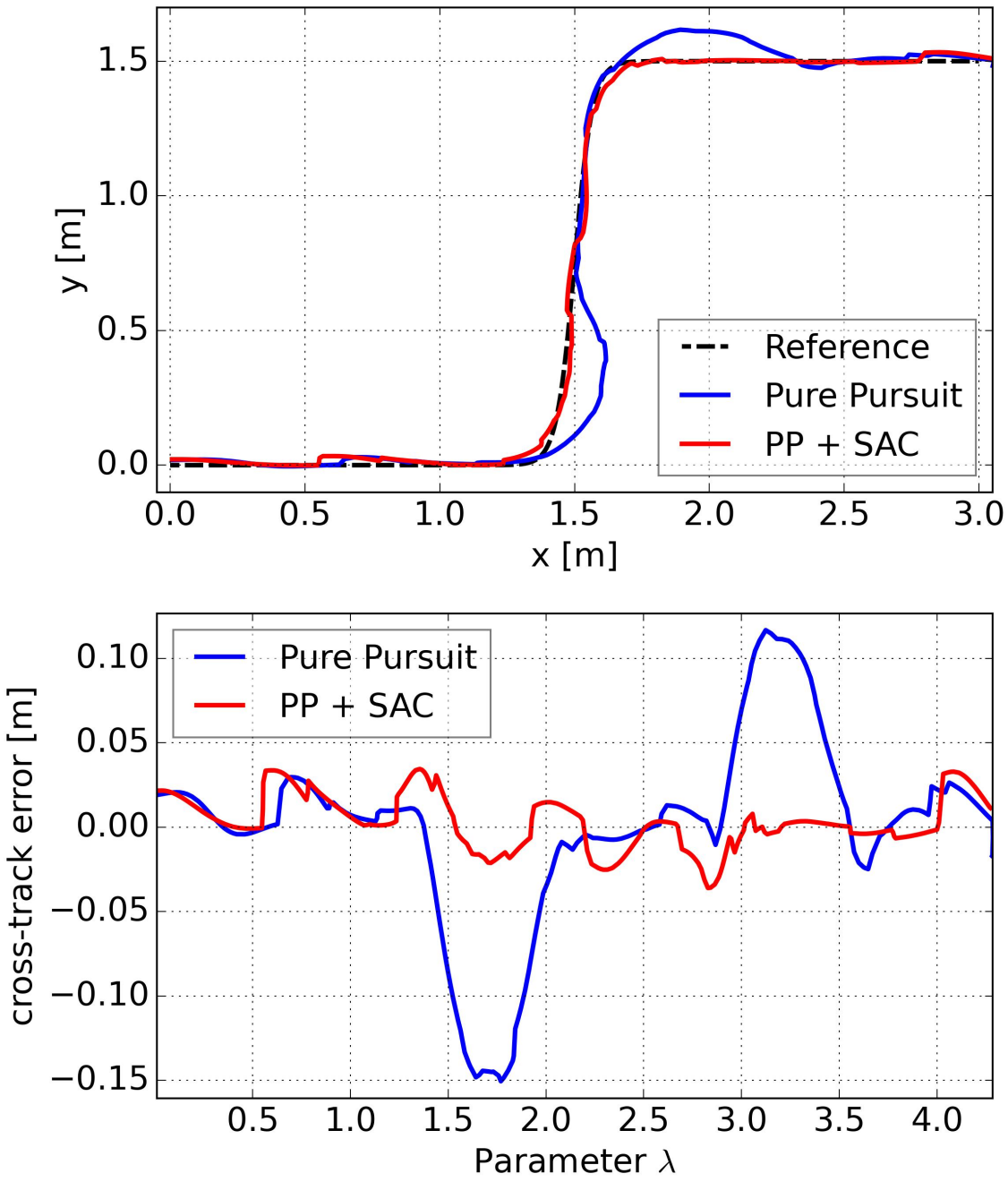

Figure 5.26: Trajectory and cross-track error comparison for the lane-change path in experiment.

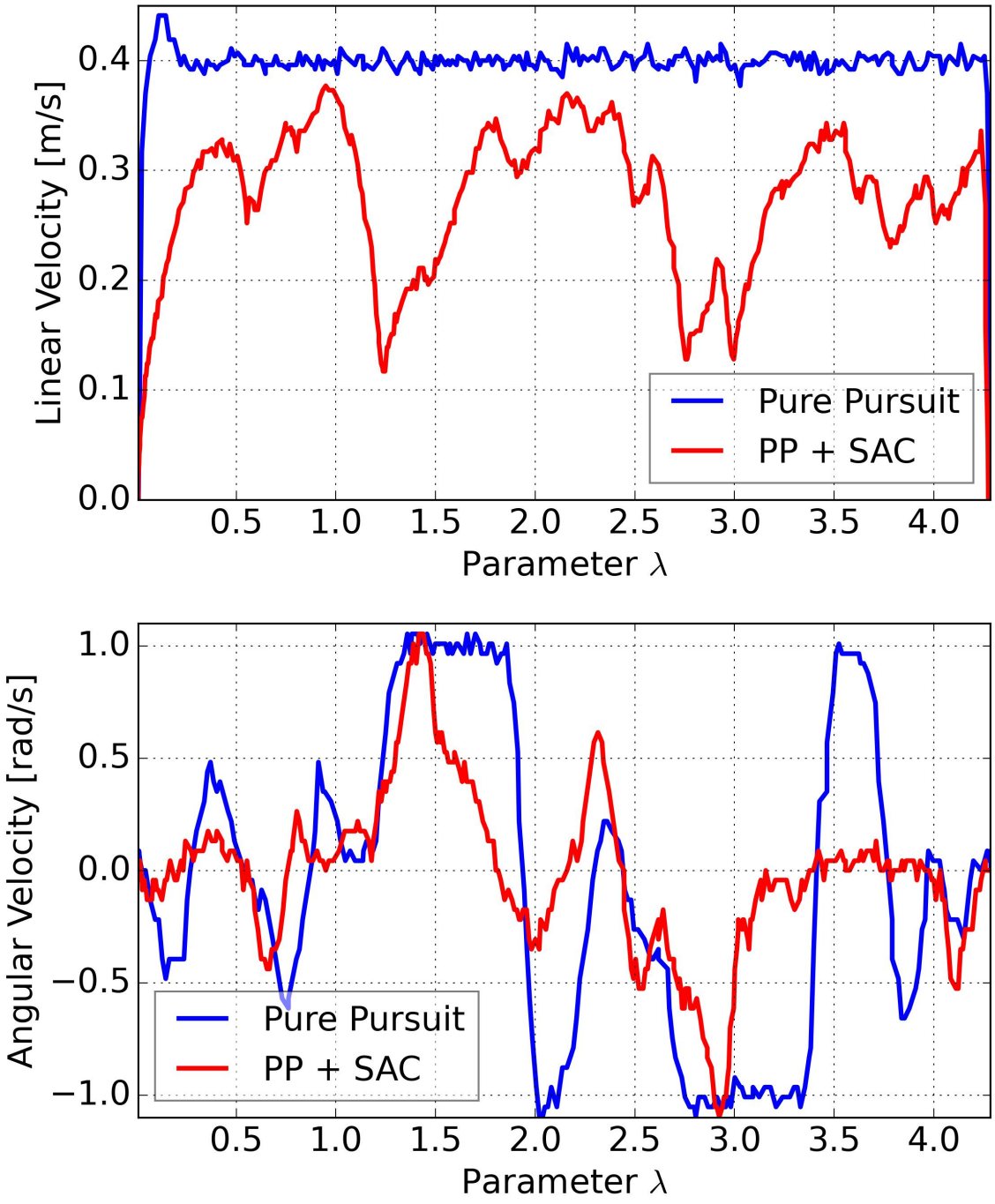

Figure 5.27: Velocity comparison for the lane-change path in experiment.

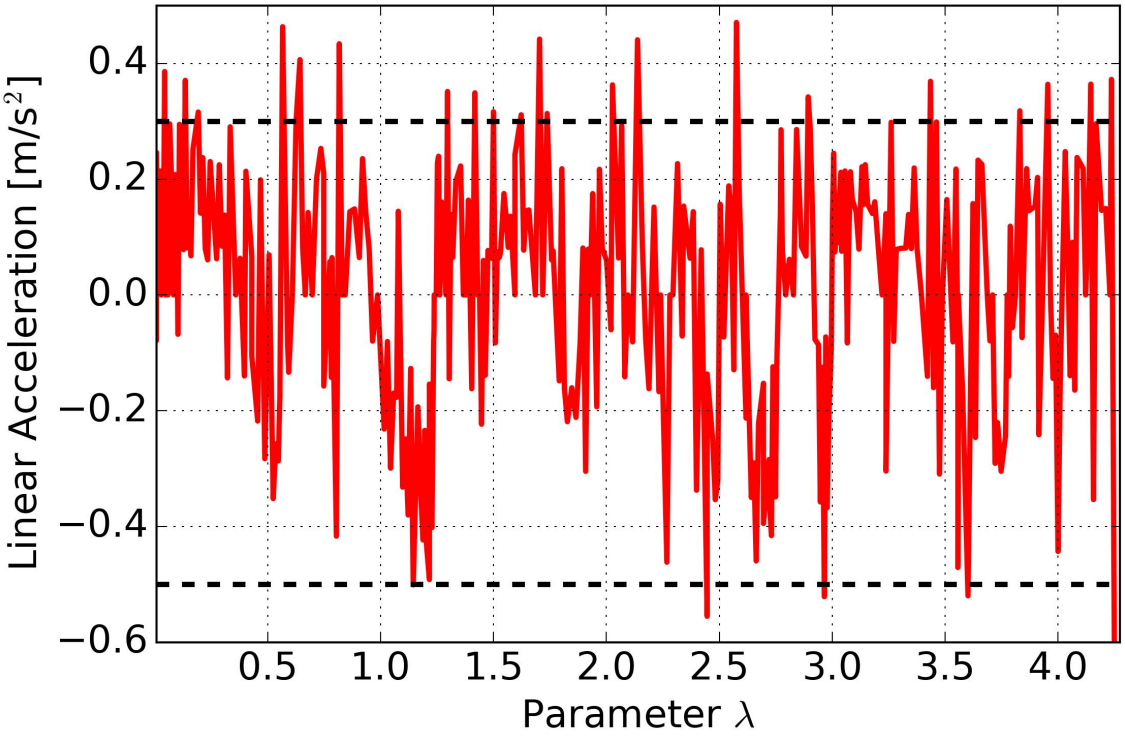

Figure 5.28: The accelerations corresponding to Figure 5.27.

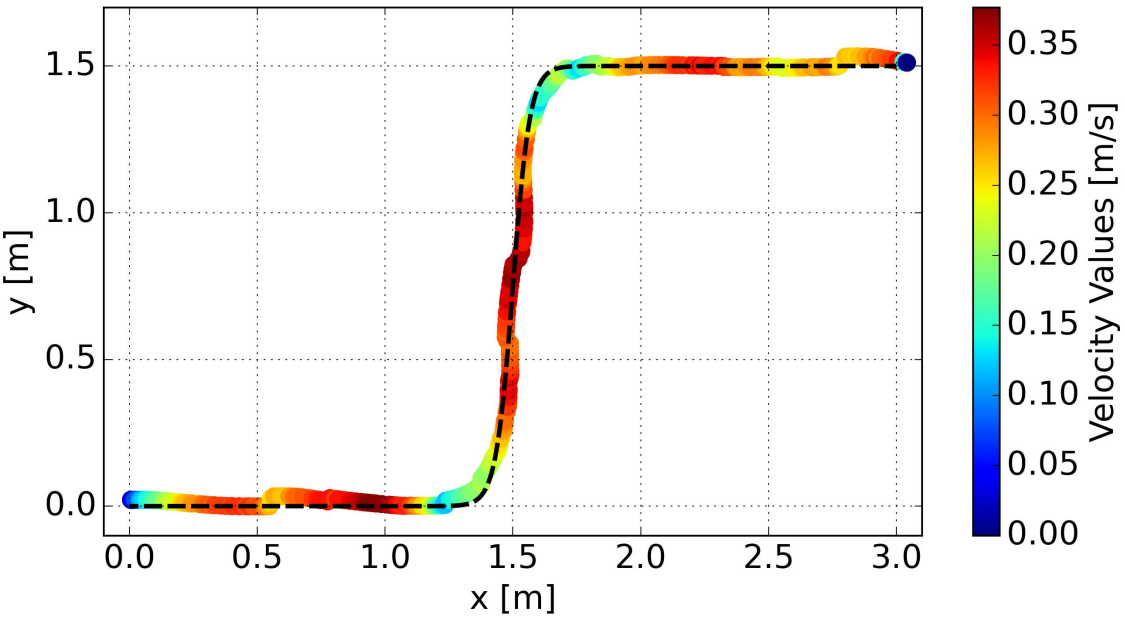

Figure 5.29: Linear velocity variation along the trajectory for the lane-change path in experiment.

References

[1] "Raspberry pi." https://www.raspberrypi.com. Accessed: 2023-11-18.

[2] "Ubuntu." https://ubuntu.com. Accessed: 2023-11-18.

[3] "Robot operating system." https://www.ros.org. Accessed: 2023-11-18.

[4] "Gmapping package." http://wiki.ros.org/gmapping. Accessed: 2023-11-18.

[5] "Adaptive monte carlo localization package." http://wiki.ros.org/amcl. Accessed: 2023-11-18.

[6] W. Zhao, J. P. Queralta, and T. Westerlund, "Sim-to-real transfer in deep reinforcement learning for robotics: a survey," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 737–744, 2020.

[7] Y. Bouteiller, S. Ramstedt, G. Beltrame, C. Pal, and J. Binas, "Reinforcement learning with random delays," in *International conference on learning representations*, 2020.

[8] B. Rubí, B. Morcego, and R. Pérez, "Deep reinforcement learning for quadrotor path following with adaptive velocity," *Autonomous Robots*, vol. 45, no. 1, pp. 119–134, 2021.

CHAPTER 6

Conclusion

In this dissertation, a novel approach to applying DRL to the problem of path following for a nonholonomic wheeled mobile robot is introduced. Unlike most existing methods, the proposed approach is a model-free optimal control method. This is achieved by allowing the DRL algorithm to explore the environment and create an internal representation in the form of an action-value function, used to optimize the control policy.

Notably, this approach places a significant emphasis on optimizing velocity control—a factor often overlooked in existing research, where the predominant focus is on steering control in path-following problems. Consequently, the proposed method can be generalized to train a velocity control policy based on a mature steering strategy. The specific steering strategy is not critical, as long as it ensures path convergence. Moreover, the proposed method is not limited to the two-wheel differential drive model used in this dissertation; it can be applied to any nonholonomic wheeled mobile vehicle, such as a car-like bicycle model, either by replacing it with a different steering control strategy like the Stanley controller or by maintaining the pure pursuit controller.

The policy is trained in a completely random environment, featuring diverse and challenging paths, enhancing the policy's robustness. The dissertation introduces two evaluation criteria: the failure rate and completion rate within a large sample of random paths. Compared to a fixed-speed control path following algorithm, the proposed method demonstrates significant improvements in both criteria, reducing the failure rate and increasing the completion rate. Additionally, the performance of the trained policy is validated through simulations and experiments on two specific paths. The policy exhibits specific behaviors, such as anticipatory deceleration before entering curved segments and maximizing speed in relatively flat segments.

To address training convergence and real-world data collection, the training is conducted in an ideal simulation environment. However, it does not consider the time constant and delay in speed response due to communication and physical limitations. This is mitigated by scaling the action size to ensure actions recover to reach the maximum, simplifying and ensuring the success of training without the need for system identification.

While the method has shown promising results, it does have a few drawbacks that must be considered. This dissertation did not explore simultaneous learning of steering and velocity control using RL. This decision is influenced by the fact that training both steering and velocity control simultaneously would lead to premature steering behaviors to maintain high speed, given that the observational inputs in the proposed design include future steering information. Moreover, introducing an additional action would exponentially increase the action space, significantly slowing down convergence. In this context, the proposed method, which prioritizes fast convergence and stability, is preferable. Additionally, the dissertation does not account for the gap between simulation and reality caused by action delay. Delayed actions or observations can violate the Markov property, where the conditional probability distribution of the current state only depends on the preceding state and not earlier ones. Recent studies suggest that addressing this issue by establishing an augmented Markov process, specifically a MDP with random delays, could provide a more accurate solution. Therefore, this represents a potential direction for future work, where the trained policy may be directly applied to experiments without adjusting action scaling.

In conclusion, the results obtained in this study provide compelling evidence for the feasibility of employing DRL in the context of path following for nonholonomic wheeled mobile robots. It contributes to the advancement of autonomous mobility by integrating conventional algorithm with state-of-the-art DRL techniques, thereby enhancing the robustness of path following. Furthermore, it has the potential for broader application to vehicles constrained by nonholonomic principles, extending beyond the specific model studied in this dissertation.

Publication List

Related journals

[1] Y. Cao, K. Ni, T. Kawaguchi, and S. Hashimoto, "Path Following for Autonomous Mobile Robots with Deep Reinforcement Learning," *Sensors*, vol. 24, no. 2, 561, pp. 1–22, 2024. DOI: 10.3390/s24020561.

[2] Y. Cao, K. Ni, X. Jiang, T. Kuroiwa, H. Zhang, T. Kawaguchi, S. Hashimoto, and W. Jiang, "Path Following for Autonomous Ground Vehicle Using DDPG Algorithm: A Reinforcement Learning Approach," *Applied Sciences*, vol. 13, no. 11, 6847, pp. 1–20, 2023. DOI: 10.3390/app13116847.

Reference journals

[1] Y. Cao, Y. Ikenoya, T. Kawaguchi, S. Hashimoto, and T. Morino, "A Real-Time Application for the Analysis of Multi-Purpose Vending Machines with Machine Learning," *Sensors*, vol. 23, no. 4, 1935, pp. 1-17, 2023. DOI: 10.3390/s23041935.

[2] S. Xu, Y. Cao, S. Hashimoto, Y. Fuiji, A. Takita, and W. Jiang, "Control Design Applicable to a Helmet Type Full-face Mask," *Journal of Technology and Social Science*, vol. 4, no. 3, pp. 24–30, 2020.

International conference papers

[1] Y. Cao, S. Xu, S. Hashimoto, T. Kawaguchi, Y. Fuiji, A. Takita, and W. Jiang, "Control Design of a Lightweight Helmet Type Full-face Mask," in *Proceedings of the International*

Conference on Technology and Social Science 2020 (ICTSS2020), IPS10-01, pp. 1–7, Kiryu, Japan, December 2020.

[2] Y. Cao, S. Hashimoto, and K. Ni, "Nonlinear Observer-Based Sensor Fusion of IMU and GPS for Self-Localization," in *Proceedings of the International Conference on Mechanical, Electrical and Medical Intelligent System 2019 (ICMEMIS2019)*, IPS4-04, pp. 1–6, Kiryu, Japan, December 2019.

[3] Y. Ikenoya, Y. Cao, T. Kawaguchi, S. Hashimoto, and T. Morino, "Machine Leaning-Based Diagnosis for Multi-Purpose Vending Machines," in *Proceedings of the International Conference on Technology and Social Science 2021 (ICTSS2021)*, IPS6-03, pp. 1–6, Kiryu, Japan, December 2021.

Domestic conference papers

[1] Y. Cao, Y. Ikenoya, F. Kristianto, T. Kawaguchi, S. Hashimoto, and T. Morino, "A Deep Learning-based Model Applicable to Analysis of Vending Machines," 第64回自動制御連合講演会講演論文集, 2D2-4, pp. 968–970, November 2021.

[2] Y. Cao, K. Ni, Y. Kuwabara, T. Kawaguchi, and S. Hashimoto, "An Easy-to-Implement Self-Localization Algorithm using Nonlinear Observer-Based Fusion," 電気学会第11回東京支部栃木・群馬支所合同研究発表会資料, ETT21-14, pp. 1–3, March 2021. (優秀論文発表賞受賞)

[3] 三上 凌, 桑原 勇太, 曹 宇, 川口 貴弘, 橋本 誠司, 岩瀬 勉, 込山 菜摘, 杉山 大貴, 塚崎 裕一郎, "車両制御性能の線形二次レギュレータによる評価法とその感度解析," 自動車技術会2023年秋季大会学術講演会予稿集, 20236166, pp. 1–5, October 2023.

[4] 三上 凌，桑原 勇太，曹 宇，川口 貴弘，橋本 誠司, "線形二次レギュレータを用いた自動車制御系の評価法に関する一考察," 電気学会第12回東京支部栃木・群馬支所合同研究発表会資料, ETT22-31, pp. 1–3, March 2022.

[5] 桑原 勇太，曹 宇，三上 凌，川口 貴弘，橋本 誠司, "自動運転のための学習理論に基づく車両特性変化の検出," 電気学会第12回東京支部栃木・群馬支所合同研究発表会資料, ETT22–30, pp. 1–4, March 2022.

Awards

[1] 優秀論文発表賞

Y. Cao, K. Ni, Y. Kuwabara, T. Kawaguchi, and S. Hashimoto, “An Easy-to-Implement Self-Localization Algorithm using Nonlinear Observer-Based Fusion,” 電気学会第11回東京支部栃木・群馬支所合同研究発表会資料, ETT21-14, pp. 1–3, March 2021.