University of Texas Rio Grande Valley
ScholarWorks @ UTRGV

Theses and Dissertations

7-29-2024

Personalized Driving Using Inverse Reinforcement Learning

Rodrigo J. Gonzalez Salinas
The University of Texas Rio Grande Valley

Follow this and additional works at: https://scholarworks.utrgv.edu/etd

Part of the Artificial Intelligence and Robotics Commons, and the Engineering Commons

Recommended Citation

Gonzalez Salinas, Rodrigo J., "Personalized Driving Using Inverse Reinforcement Learning" (2024). *Theses and Dissertations*. 1506.
https://scholarworks.utrgv.edu/etd/1506

This Thesis is brought to you for free and open access by ScholarWorks @ UTRGV. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of ScholarWorks @ UTRGV. For more information, please contact justin.white@utrgv.edu, william.flores01@utrgv.edu.

PERSONALIZED DRIVING USING INVERSE REINFORCEMENT LEARNING

A Thesis
by
RODRIGO J. GONZALEZ SALINAS

Submitted in partial fulfillment of the
Requirements for the degree of
MASTER OF SCIENCE IN ENGINEERING

Major Subject: Mechanical Engineering

The University of Texas Rio Grande Valley
May 2024

PERSONALIZED DRIVING USING INVERSE REINFORCEMENT LEARNING

A Thesis
by
RODRIGO J. GONZALEZ SALINAS

COMMITTEE MEMBERS

Dr. Constantine Tarawneh
Co-Chair of Committee

Dr. Tohid Sardarmehni
Co-Chair of Committee

Dr. Horacio Vasquez
Committee Member

Dr. Qi Lu
Committee Member

May 2024

Copyright 2024 Rodrigo J. Gonzalez Salinas

All Rights Reserved

ABSTRACT

Gonzalez Salinas, Rodrigo J., Personalized Driving Using Inverse Reinforcement Learning. Master of Science in Engineering (MSE), May 2024, 63 pp., 5 tables, 29 figures, 35 titles.

This thesis introduces an autonomous driving controller designed to replicate individual driving behaviors based on a provided demonstration. The controller employs Inverse Reinforcement Learning (IRL) to formulate the reward function associated with the provided demonstration. IRL is implemented through a dual-feedback loop system. The inner loop utilizes Q-learning, a model-free reinforcement learning technique, to optimize the Hamilton-Jacobi-Bellman (HJB) equation and derive an appropriate control solution. The outer loop leverages this derived control solution to generate parameters for the reward function, which are subsequently integrated into the HJB equation. The ultimate control policy is deduced from the final reward function obtained through IRL. To facilitate the recording of expert demonstrations and the evaluation of the final control policy, the CARLA autonomous driving simulator is utilized.

DEDICATION

This thesis is dedicated to those who have made an everlasting impact on me as a person: to my loving family, who have encouraged me throughout my education and supported me throughout my life; to my father, Jesus Gonzalez, and my mother, Adriana Gonzalez, whose unconditional love and teachings have shaped me into the person I am today; to my siblings, Rebeca and Rodolfo, who have always been there for me; to my friends, who have patiently listened to my ramblings and have not hesitated to call me out when I am not acting the part.

ACKNOWLEDGEMENTS

This work was made possible through the support of the NSF CREST Center for Multidisciplinary Research Excellence in Cyber-Physical Infrastructure Systems (MECIS) through NSF Award No. 2112650, the Dwight David Eisenhower Transportation Fellowship Program (DDETFP) under US DOT Award No. 693JJ32245176, and the University of Texas Rio Grande Valley’s Presidential Research Fellowship.

I would like to express my gratitude to my advisor, Dr. Constantine Tarawneh, for his support, availability, and understanding. I appreciate his unwavering commitment to addressing students' concerns promptly and for guiding me back onto the right path when I strayed from it. The positive impact he has had on my academic, personal, and professional development is greatly acknowledged.

I am also grateful to my advisor, Dr. Tohid Sardarmehni, for introducing me to this research field and for providing guidance and support throughout my graduate journey. His boundless patience and knowledge, shared with me along this journey, played a pivotal role in my success. I extend my thanks for his mentorship and assistance in bringing this project to its conclusion.

TABLE OF CONTENTS

Page

ABSTRACT iii

DEDICATION iv

ACKNOWLEDGEMENTS v

TABLE OF CONTENTS vi

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER I BACKGROUND AND INTRODUCTION 1

- 1.1 Optimal Control 1
- 1.2 Reinforcement Learning 2
  - 1.2.1 Reinforcement Learning and Supervised Learning 4
  - 1.2.2 Offline and Online Training 5
- 1.3 Dynamic Programming 6
  - 1.3.1 Twin Curses 8
- 1.4 Approximate Dynamic Programming 10
  - 1.4.1 Q-Learning 12
- 1.5 Inverse Reinforcement Learning 12
  - 1.5.1 Methods of IRL 14
- 1.6 Applications 15
- 1.7 Contribution 16

CHAPTER II APPROXIMATE DYNAMIC PROGRAMMING ALGORITHMS 18

- 2.1 Heuristic Dynamic Programming 18
  - 2.1.1 The Bellman Equation 19
  - 2.1.2 HDP: Defining an Optimal Policy 20

2.2 Action Dependent Heuristic Dynamic Programming 21
2.2.1 ADHDP: Defining the Optimal Policy 22
2.3 The Value Iteration Algorithm 23
CHAPTER III INVERSE REINFORCEMENT LEARNING ALGORITHM 26
3.1 Inverse Reinforcement Learning 26
3.1.1 Gradient Based Learning 28
3.1.2 Algorithm Optimizers 29
3.2 Gradient Based Inverse Reinforcement Learning 32
CHAPTER IV SIMULATIONS AND RESULTS 35
4.1 Simulation Overview 35
4.2 Van der Pol Overview 35
4.2.1 Van der Pol Reinforcement Learning Overview 36
4.2.2 Van der Pol Reinforcement Learning Results 39
4.2.3 Van der Pol Inverse Reinforcement Learning Results 42
4.3 CARLA Overview 44
4.3.1 CARLA Setup 45
4.3.2 CARLA Results 50
CHAPTER V CONCLUSION 55
4.1 Future Works 56
REFERENCES 58
APPENDIX 61
BIOGRAPHICAL SKETCH 63

LIST OF TABLES

Page

Table 1. Performance metrics formulation. 2

Table 2. Dynamic programming table (Kirk, 2004). 7

Table 3. Value iteration table of variables. 25

Table 4. Q-Learning parameters. 38

Table 5. CARLA IRL algorithm parameters. 49

LIST OF FIGURES

Page

Figure 1: MDP loop of agent and environment interaction (Sutton, 2018). 4

Figure 2. Curse of dimensionality. 9

Figure 3. Iterative loop of IRL. 14

Figure 4. Basic flowchart of inverse reinforcement learning. 26

Figure 5. GBIRL Flowchart. 33

Figure 6. Phase diagram of undisturbed oscillator. 37

Figure 7. Undisturbed states as functions of time. 37

Figure 8. VDP regularized phase diagram. 39

Figure 9. VDP regularized states. 40

Figure 10. VDP control. 40

Figure 11. Converged weights of RL process. 41

Figure 12. State replication. 42

Figure 13. Control replication. 43

Figure 14. History of IRL predictions. 44

Figure 15. CARLA application window. 45

Figure 16. CARLA Town04 (Dosovitskiy, 2017). 46

Figure 17. Physical data gathering setup. 47

Figure 18. Velocity validation data for neural network. 48

Figure 19. Acceleration validation data for neural network. 48

Figure 20. Weight history of RL for the last iteration of IRL. 50

Figure 21. IRL loss history over training. 51

Figure 22. Parameter history of CARLA IRL. 51

Figure 23. States prediction comparison. 52

Figure 24. Brake control prediction comparison. 53

Figure 25. Throttle control prediction comparison. 53

Figure 26. VDP states with differing initial conditions. 61

Figure 27. VDP control with differing initial conditions. 61

Figure 28.VDP states with differing initial conditions outside domain of training. 62

Figure 29.VDP states with differing initial conditions outside domain of training. 62

CHAPTER I

BACKGROUND AND INTRODUCTION

1.1 Optimal Control

Optimal control is a specialized field of control engineering that focuses on analyzing systems and determining optimal control signals to maximize or minimize aspects of a dynamical system. The process of deriving such control systems is known as optimal control. The objective of optimal control theory is to determine a set of control signals that will cause a system to behave optimally within physical constraints. The control signals that allow such behavior are obtained by optimizing a function delineated by performance criteria (Kirk, 2004). Those control signals are referred to as an optimal policy and are obtained with the use of Bellman's principle of optimality. The principle states that an optimal policy must act optimally on the states leading to the desired final state. The optimal policy needs to act optimally without regard to the initial decision made (Bellman, 1957).

Optimal control begins with the modeling of a dynamic system. The goal of defining a model for any system is to have the simplest mathematical representation that can predict a sensical response to inputs. A common method to achieve this is through a system of differential equations in state-space form (Kirk, 2004). Constraints are enacted on the inputs of the model and may impact both the state space and the control space. In terms of the state space, constraints include the designation of starting and final values. This deliberate boundary assignment details a trajectory's beginning and ending. For the control space, constraints typically relate to the physical system's operational limits. For instance, a control action such as the rotation of the steering wheel is limited by the displacement of the tie rod, which pivots the wheels. Consequently, the control constraints would be defined by the range of angles that maximize the displacement of the tie rod in either direction.

The aspects of dynamical systems that are optimized in control systems are commonly referred to as performance metrics. According to Kirk (2004), performance metrics can describe anything depending on the problem type that needs optimization. For instance, minimum-time problems are a type of problem where the metric to be optimized is the time required to get from the beginning state to a defined final state. Terminal control problems aim to minimize the deviation between the final state of the controlled system and the desired final state. Minimum-control-effort problems are problems where the control effort, such as fuel expenditure, is to be minimized. The formulations of the performance metrics can be seen in Table 1.

Table 1. Performance metrics formulation.

Performance Metrics	Formulation
Minimum time	$J = t_f - t_0$ $J = \int_{t_o}^{t_f} dt$
Terminal control	$J = \left\Vert x(t_f) - r(t_f) \right\Vert^2$
Minimum control energy	$J = \int_{t_0}^{t_f} [u^T R u]\, dt$
Tracking	$J = \int_{t_0}^{t_f} \left[\left(x(t) - r(t)\right)^T Q \left(x(t) - r(t)\right) \right] dt$

1.2 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning method that empowers an agent to formulate the optimal behavioral pattern to achieve a designated goal through dynamic interactions with its environment. The essence of RL lies in constructing a policy that adeptly maps the agent's actions to the state of the environment, with the aim of maximizing rewards or minimizing costs. At the foundational level, RL presents the agent with a limited selection of actionable choices at any given moment, enabling it to learn the most rewarding actions in response to various environmental states. In scenarios of increased complexity, the choices made by the agent not only influence immediate rewards but also have ramifications on future rewards, culminating in the successful completion of the task. This detail transforms the learning process into a deliberate exploration for a sequence of actions that optimizes cumulative rewards. As described by Richard Sutton and Andrew Barto (2018), the defining characteristics of reinforcement learning are the strategic navigation through trial and error and the management of delayed rewards.

In RL, the interaction between the agent and the environment is the focal point in mastering a desired task. RL encompasses four fundamental elements: a policy, a reward signal, a value function, and, optionally, a model of the environment (Sutton, 2018). The policy functions as a mapping from observed states to potential actions the agent might take, essentially transcribing the agent's behavior either as a function or a trajectory. The reward signal defines the objectives of the RL problem by issuing numerical rewards to the agent following each action, thus playing a crucial role in shaping and refining the agent's policy by indicating the desirability of actions. While the reward signal focuses on immediate rewards, the value function predicts the total achievable reward from a given state, considering both the immediate and potential future states. This approach enables an assessment of long-term benefits of states and actions taken. Lastly, the environment model, when used, predicts the consequences of the agent's actions on the environment. Methods to solve RL problems can be differentiated within RL: model-based methods, which utilize this predictive model, and model-free methods, which operate without explicit knowledge of the environmental dynamics, relying solely on trial and error. The choice between model-based and model-free approaches depends on the specific requirements and constraints of the learning task.

An approach to properly formulate an RL problem is through the application of a Markov Decision Process (MDP). MDP offers a mathematical framework that models the decision-making process of an agent in a stochastic environment, presented as a sequence of discrete time steps. This can be observed in Figure 1, which depicts a flowchart of agent-environment interactions in an MDP. At each discrete timestep, t, the agent senses the current state, S_t, of the environment, and executes an action, A_t, to advance to the next time step. In the following time step, $t + 1$, the agent receives a numerical value as a reward, R_{t+1}, influenced by the preceding action, then the agent transitions to a new state, S_{t+1}.

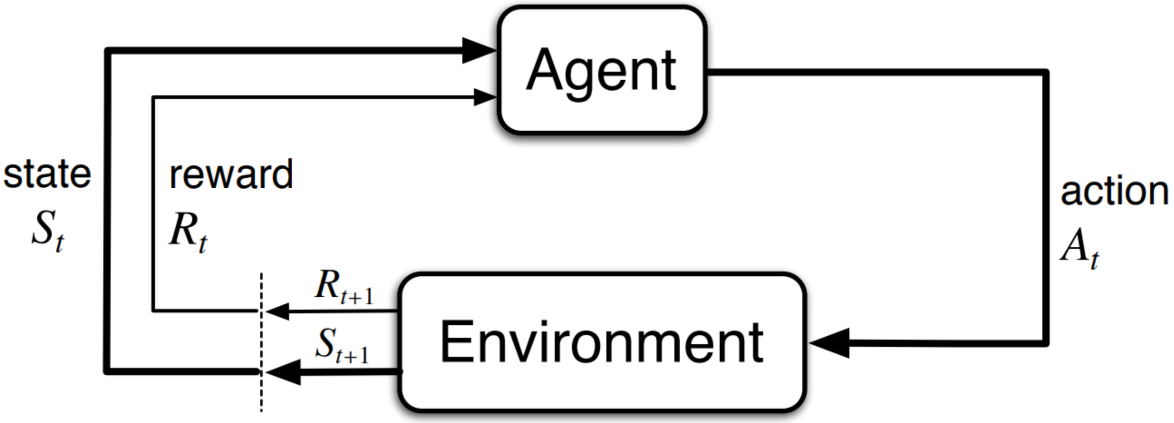


Figure 1: MDP loop of agent and environment interaction (Sutton, 2018).

1.2.1 Reinforcement Learning and Supervised Learning

RL is often mistakenly equated with a different but just as recognized machine learning model that is Supervised Learning. Supervised Learning revolves around the development of algorithms capable of generating functions that map inputs to their corresponding outputs (Nasteski, 2017). This mapping is refined through iterative comparisons between the algorithm's predicted outputs and the actual desired outputs. Supervised Learning algorithms are favored for tasks like multi-class object classification, where they employ the use of labeled data for training. In this context, each data label acts as a target value compared to the predicted label for a given input. Within a Markov Decision Process (MDP), a Supervised Learning model would learn by acquiring a function that maps a set of inputs to a policy, essentially being instructed on the optimal next action for any given state. The primary goal in Supervised Learning is to minimize the discrepancy between the predicted and target values.

However, the fundamental distinction between Supervised Learning and RL lies in their approach to training data and decision-making. Unlike Supervised Learning, RL does not operate on predefined input-output pairs for training. Instead, the agent independently determines the optimal course of action without explicit guidance on long-term benefits. Instead, upon taking an action, the agent receives a reward and progresses to the state that follows. This process is essential for the agent to accumulate experiences across states, actions, and rewards to formulate an optimal policy (Kaelbling, 1996). Due to the ample knowledge collected from experiences, the optimal policy increases in robustness, becoming more applicable to an agent navigating unforeseen scenarios.

1.2.2 Offline and Online Training

The training methodologies for algorithms fall into two primary categories: online and offline. Online training involves the actor engaging directly with the environment, learning, and adapting in real-time. The computational intensity of this method varies with the complexity of the simulation and tends to be slower than offline training due to processing individual observations sequentially. Online training shines in scenarios characterized by environmental unpredictability or when the simulation cannot be neatly encapsulated by a set of equations.

Conversely, offline training eliminates real-time interaction between the actor and the environment. Rather than accumulating data through live interaction, this approach relies on pre-gathered datasets comprising states, actions, and rewards. For offline training to effectively develop robust optimal behaviors, the data must be expansive and accurately reflect the target environment. A lack of representative data might lead the actor to develop sub-optimal behaviors, particularly in response to unexpected circumstances.

1.3 Dynamic Programming

Reinforcement Learning (RL) incorporates principles akin to those of Dynamic Programming (DP) (Bertsekas, 2019). DP is a recursive approach for solving multistage decision-making processes based on the principle of optimality (Bellman, 1966). It operates in a backward-in-time fashion, calculating the optimal policy starting from the trajectory's end and proceeding to the initial point. This method is particularly suited to problems utilizing discrete action and state spaces due to the tabular nature of the solution. For continuous spaces, the system must be discretized in time, and both the spaces for actions and states must be quantized.

Within a given system, DP computes the immediate cost-to-go for each action at a specific state to transition to the next state (Kirk, 2004). Important details—current state, action taken, subsequent state, cost-to-go, and total cost—are stored in a table. This table effectively maps out the route with the least total cost, denoting the optimal policy. The optimal policy, or optimal control law, minimizes the total cost, which is the sum of the immediate costs-to-go from one state to the next, considering a specific control action, plus the total minimal cost from all subsequent steps to the end state. DP iteratively resolves the decision-making process to achieve minimum cost, utilizing table values and, if necessary, interpolating for intermediate steps. The resultant policy is a robust, closed-loop control strategy for the system. An illustration of DP's tabular approach is provided in Table 2. It is important to note that the table reflects the final step of a policy, excluding future costs.

Table 2. Dynamic programming table (Kirk, 2004).

Current State	Control	Next state	Cost-to-go	Minimum cost	Optimal Control
x_1	u	$x_2 = x_1 + u$	$J = x_2^2 + 2u^2$	$J^*(x_1)$	$u^*(x_1, 1)$
1.5	0.0 -0.5 -1.0	1.5 1.0 0.5	2.25 1.50 2.25	1.5	-0.5
1.0	0.5 0.0 -0.5 -1.0	1.5 1.0 0.5 0.0	2.75 1.00 0.75 2.00	0.75	-0.5
0.5	1.0 0.5 0.0 -0.5	1.5 1.0 0.5 0.0	4.25 1.50 0.25 0.50	0.25	0.0
0.0	1.0 0.5 0.0	1.0 0.5 0.0	3.00 0.75 0.00	0.00	0.0

While previously mentioned that Table 2 reflects only the final step of a policy, it can be used to show a simplified example of DP. To derive an optimal policy starting from an initial state of 1.0 and aiming for a final state of 0.0, DP evaluates the total cost for various strategies. One potential policy, $u_1 = [-1.0]$, carries a cost of 2.0. Another policy, $u_2 = [0.5, -1.0, -0.5]$, incurs a cost of 5.5, calculated by summing the costs-to-go between the sequential states. A third policy, $u_3 = [-0.5, -0.5]$, has a cost of 1.25, determined by the cumulative cost-to-go from state 1.0 to 0.0 through intermediate steps. Among these, the optimal policy is $u^* = u_3$, chosen for its lowest total cost. This example highlights the tabular methodology of DP, which, while offering a straightforward recursive solution, may face challenges with scalability in terms of state and action space size, as well as the incorporation of additional states and controls.

1.3.1 Twin Curses

Dynamic Programming (DP) excels at producing optimal solutions through its tabular approach, making it a robust method for solving control problems of low complexity. However, as the complexity of a problem escalates—caused by an increase in variables such as the number of controls, states, or the expansion of control and action spaces—the method faces significant problems. Specifically, the computational demands and memory requirements increase exponentially, a phenomenon known as the curse of dimensionality. This issue complicates, and in some cases, may even render many Markov Decision Processes (MDPs) unsolvable (Powell, 2011; Bethke, 2010). Gosavi, A. (2015) provides an example, analyzing an MDP with 1,000 states and two actions. The resulting transition probability matrix for each state would encompass one million elements, posing formidable computational hurdles due to the limitations of current computing resources. Figure 2 offers a graphical representation of the curse of dimensionality, illustrating the exponential growth of the state-action space with the addition of more states and actions.

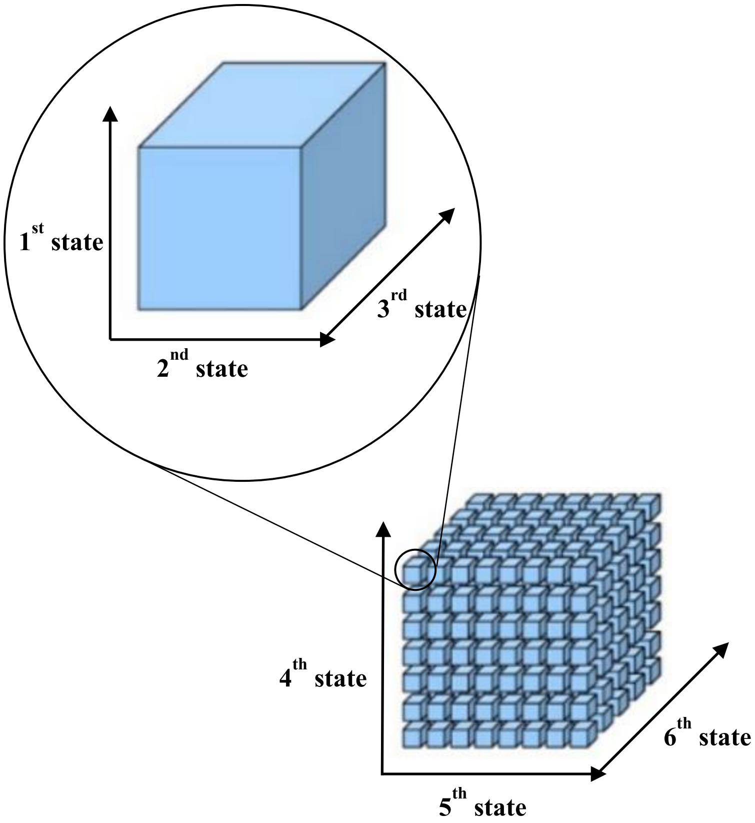


Figure 2. Curse of dimensionality.

The construction of a theoretical model for a system often involves defining elements such as time, rewards, and either transition probabilities or system dynamics. When focusing on transition probabilities, the task can become particularly difficult; evaluating these probabilities may require the computation of multiple integrals that involve both probability mass functions and distribution functions of numerous random variables. The presence of a significant number of random variables substantially complicates the creation of accurate transition probabilities for the system (Gosavi, 2015). Alternatively, when the model is based on system dynamics, it necessitates deriving a set of governing equations through the resolution of differential equations. This approach faces its own set of challenges, especially with highly complex or partially understood systems, where only fragments of the dynamics are known. Such scenarios make deriving comprehensive governing equations a challenging task. This difficulty in accurately modeling a system's theoretical framework is termed the curse of modeling (Gosavi, 2015). It represents a major obstacle in dynamic programming, alongside the well-documented curse of dimensionality. To navigate these twin challenges, practitioners often resort to Approximate Dynamic Programming (ADP), a method that offers a viable pathway for addressing the complexities associated with detailed system modeling.

1.4 Approximate Dynamic Programming

Approximate Dynamic Programming (ADP) emerged as an innovative solution to the curse of dimensionality, by introducing a different approach to solving complex decision-making problems. Unlike DP which relies on storing detailed tables to map out the action-space, ADP is able to ignore this requirement through the use of function approximators. By employing approximation tools such as neural networks, ADP can formulate policies that, while not perfectly optimal, are sufficiently close to optimal, near optimal. This trade-off in accuracy significantly decreases computational demands, making it feasible to tackle problems that were previously unsolvable to DP. ADP encompasses two primary algorithmic strategies: value iteration ADP and policy iteration ADP. These methods embody principles of RL, drawing on Bellman’s optimality condition to guide the search for near-optimal solutions (Li, 2023).

ADP frequently employs an architecture known as the adaptive-critic or actor-critic (AC) to facilitate learning. This architecture synergizes two neural networks, the actor and the critic, to iteratively learn an optimal policy (Konda, 1999). The process hinges on optimizing the Hamilton-Jacobi-Bellman (HJB) equation, a non-linear partial differential equation that encapsulates the value function. Given the complexity of directly solving the HJB equation, due to its inability to be analytically solved, ADP resorts to numerical solutions that approximate Bellman’s equation, a discrete version of the HJB equation (Kirk, 2004). Within the AC architecture, the actor neural network is tasked with approximating the optimal policy, while the critic network focuses on estimating the value function, effectively embodying the optimal value function. This setup is similar to the interaction in Markov Decision Processes (MDPs), where the actor represents the agent making decisions based on control actions, and the critic parallels the environment, evaluating those actions to estimate the value of the states and controls utilized. Through this iterative process, both networks concurrently refine their approximations until they converge, signaling the derivation of a near-optimal solution to the HJB equation.

Werbos (1992) introduces four methods of ADP methods: heuristic dynamic programming (HDP), dual heuristic programming (DHP), action-dependent heuristic dynamic programming (ADHDP), and action-dependent dual heuristic programming (ADDHP). Each method serves a unique role in the ADP landscape. For HDP it is necessary to have a comprehensive understanding of the system's model, employing a function approximator to encapsulate the value function, essentially Bellman’s equation. Similarly, DHP also demands full model knowledge but changes its focus towards learning the derivative of the Bellman equation, known as the costate, through approximation.

Diverging from the requirement of complete system dynamics knowledge, ADHDP can operate with no or partial knowledge of the system, applying an approximator to resolve the quality function variant of Bellman’s equation. Similarly, ADDHP, does not require detailed model insight but aims instead to approximate the action-dependent costate, the derivative of the quality function version of Bellman’s equation. Notably, ADHDP is synonymous with Q-learning, a widely recognized approach within the domain of reinforcement learning for deriving optimal policies. It is this method, Q-learning, that this thesis employs to approximate an optimal policy.

1.4.1 Q-Learning

In HDP, also known as value function learning, a detailed understanding of the dynamic model that represents the system is essential to approximate a solution to the HJB equation. This requirement stems from the algorithm's minimization process, which relies on elements from the dynamic model. However, to overcome the necessity for such detailed system knowledge, ADHDP, widely recognized as Q-learning, is employed (Lewis, 2009). Q-learning, a valuable model-free RL technique, empowers an agent to learn optimally without the prerequisite of mapping the relationship between controls and states (Watkins, 1992). Unlike traditional methods where the critic aims to approximate the optimal value function, in Q-learning, the focus becomes the approximation of a quality function. This function, a variation of the Bellman equation, evaluates the value of actions within specific states, incorporating both state and control action, unlike HDP where the focus is solely on states. The naming convention behind ADHDP is derived from the critic neural network's dependency on actions, emphasizing the critical role actions play in determining the quality of state transitions.

1.5 Inverse Reinforcement Learning

In traditional RL, the goal is for an actor to develop a policy that maximizes cumulative rewards across a given trajectory. This is facilitated by a pre-defined reward signal or reward function to promote specific behaviors. In the context of ADP, the reward function is often modeled after the Bellman equation, with the reward component tailored to direct the actor towards an intended objective. While defining a reward function might seem straightforward, the complexity of certain problems can make this challenging and severely time-consuming. Specifically, accurately encapsulating desired behaviors through a parameterized equation becomes difficult as complexities increase, and incorrect assumptions about these behaviors can introduce biases, potentially leading to the formulation of an inappropriate reward function (Russel, 1998).

Inverse Reinforcement Learning (IRL) emerges as a solution to this predicament. IRL seeks to deduce the behavior underlying an external actor's “expert” optimal policy, thereby circumventing the need for manually crafting a reward function (Ng, 2000). This approach not only addresses the challenge of defining precise reward functions in complex scenarios but also opens avenues for learning from observed behaviors without explicit reward notation.

One of the most compelling attributes of IRL is its ability to use recorded task data for training autonomous agents, thereby enabling them to achieve specific objectives without a decline in performance. As the complexity of a system increases, manually specifying a reward function becomes an obstacle within RL frameworks, limiting their applicability to a broader range of problems. IRL addresses this limitation by integrating with an RL algorithm that operates on a parameterized but initially undefined reward function (Arora, 2021).

In practice, an IRL neural network is given expert demonstrations as input and generates parameter values corresponding to those in the RL's reward function. These parameters are then utilized by the RL algorithm to simulate and refine an optimal control policy. The effectiveness of this policy is assessed by comparing it with the expert's policy, which was originally the input to the IRL model. Discrepancies between these policies guide the iterative updates to the IRL model, progressively aligning the RL-predicted policy with the expert demonstration. This iterative process continues until the RL-derived policy closely resembles, if not exactly matches, the expert's policy. A graphical representation of this iterative mechanism is illustrated in Figure 3.

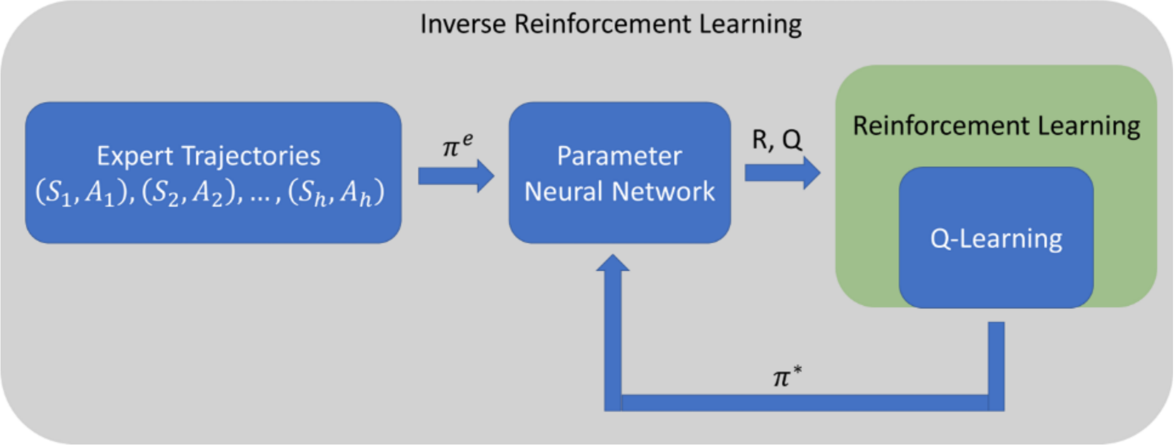


Figure 3. Iterative loop of IRL.

IRL not only streamlines the development of reward functions but also enhances their versatility. A key advantage of IRL is that it produces reward functions, which reward functions serve as concise and robust descriptors of behavior, independent of any specific agent (Ng, 2000; Abbeel, 2004; Arora, 2021). This versatility allows the same reward function to be effectively applied to diverse actors, facilitating the replication of desired behaviors or tasks across actors of different forms. Even when transitioning a reward function to a new actor encounters challenges in reaching optimality, it still offers a better starting point than a reward function initialized with arbitrary parameters. The inherent flexibility and robustness of reward functions significantly contribute to the appeal of IRL.

1.5.1 Methods of IRL

IRL can be implemented through several methods, each with its unique approach to deducing or optimizing reward functions. Margin Optimization IRL focuses on generating reward function parameters that yield an optimal policy that optimizes a specific margin (Abbeel, 2004; Arora, 2021; Imani, 2022). A common margin that is maximized is the difference between the total value of the optimal policy and the total value of the second-best policy. Another margin that is used, this time it is minimized, is the difference in feature expectations between expert demonstrations and predictions. This optimization aims to align the behavior of the agent closely with that of the expert by minimizing this marginal loss.

Maximum Entropy IRL applies the principle of maximum entropy to predict distributions or trajectories, ensuring the chosen distribution accounts for all possibilities equally, ignoring any bias aside from the constraints imposed (Arora, 2021; Wulfmeier, 2015; Ziebart, 2008; Guiasu, 1985). This method seeks to replicate the expert policy accurately without overly relying on potentially misleading assumptions.

Bayesian IRL introduces prior knowledge into the learning process, using it alongside a likelihood distribution of possible rewards to generate a posterior distribution of reward functions (Arora, 2021; Imani, 2022). This approach blends historical knowledge with new data to craft a reward function to interpret the intended task. Popular distributions used include Gibbs, Gaussian, and Laplacian.

Gradient Based IRL, meanwhile, optimizes a loss function that evaluates the difference between the expert and predicted policies. It employs an approximator, commonly a neural network, to turn the expert policy into reward function parameters, subsequently the parameterized reward function is used through RL to approximate an optimal policy. The loss function's gradient with respect to the approximator’s trainable parameters, typically some sort of weights, is then optimized using algorithms like Adam, gradient descent, or RMSprop. It is an iterative process that updates the reward function to better describe the behavior displayed by the expert policy.

1.6 Applications

Adams (2022) categorizes the applications of IRL into three areas, each highlighting its versatile potential in artificial intelligence, machine learning, and robotics. The first category focuses on replicating the behavior of expert demonstrators. This is useful for replicating complex tasks that are challenging to describe directly through reward functions. Early implementations of IRL, as demonstrated by Abbeel and Ng (2004), successfully mimicked diverse driving styles using a 2-D car simulation and controlled aerobatic helicopter to replicate demonstrated maneuvers (Abbeel, 2007). Similarly, Kuderer et al. (2015) applied IRL to replicate driving preferences termed as “relaxed” and “accelerated,” while Mombaur et al. (2011) replicated human path planning.

The second application area involves enhancing interactions between systems through learned reward functions. For instance, Kolter et al. (2008) extracted reward functions for a quadruped robot for navigating rough terrain through an initial simplified path, Chung et al. (2010) enabled a robot to predict human pathing to avoid collisions, and Shkurti et al. (2018) applied IRL for tracking and predicting the location of a target through an aerial view. The final category, as outlined by Adams (2022), employs IRL to gain deeper insights into agent behavior. This analytical application of IRL was used by Asoh et al. (2018), who analyzed medical treatment decisions for diabetes, to better understand the “mental reward function” a doctor uses to prescribe treatments. Kin et al. (2022) utilized IRL to understand and combat interference in drone swarms.

1.7 Contribution

This research aims to develop an inverse control strategy to enhance the autonomous driving experience tailored to individual drivers and their preferences. At the core of this thesis is the application of a specialized form of Inverse Reinforcement Learning (IRL), specifically Gradient-Based Inverse Reinforcement Learning. This approach integrates Q-learning, a model-free Reinforcement Learning (RL) technique, employing value iteration to approximate solutions to the Hamilton-Jacobi-Bellman equation to derive near-optimal policies. Initial algorithmic tests were conducted using the Van der Pol Oscillator, yielding positive outcomes. Subsequently, data from a ground vehicle simulator, Carla, was incorporated. The aim was to use this data within the Q-learning algorithm and, by extension, the IRL algorithm, to create a personalized reward function that resonates with the preferences of any user of the simulator. This project represents a step towards creating highly personalized autonomous driving experiences by refining reward functions to match individual driving styles and preferences.

CHAPTER II

APPROXIMATE DYNAMIC PROGRAMMING ALGORITHMS

2.1 Heuristic Dynamic Programming

The primary RL methodology employed in this project is ADHDP, commonly referred to as Q-learning. However, HDP was initially explored to gain a deeper understanding of ADP and to make apparent the advantages of Q-learning. HDP is an ADP method that focuses on the optimization of a value function, representing the worth of being in a particular state. This approach requires full knowledge of the system dynamics, usually described within a continuous space with continuous inputs. For ease of integration into the learning process, these dynamics required conversion into a discrete-time framework in an affine state space form (Lewis, 2009). This conversion was accomplished through Euler integration with a small timestep. The resultant system description is presented in Equation 1,

$$x_{k+1} = F(x_k) + G(x_k) * u_k \tag{1}$$

where the state at the current time step, $x_k \in \mathbb{R}^n$, and the control input, $u_k \in \mathbb{R}^m$, are utilized to as inputs to the dynamics leading to the subsequent state, x_{k+1}. Here, n represents the number of distinct states defining the system, while m indicates the number of control inputs that can be applied to influence the behavior of the system.

As outlined in Chapter I, a policy is a mapping of the state space of the system to the control space utilized by the system to achieve a desired task. In the context of DP, a policy is depicted as a vector of control inputs, showing directions across a table. However, in ADP, policies are expressed as a function of the states, as illustrated in Equation 2.

$$u_k = h(x_k) \tag{2}$$

In this representation, $h(.), \mathbb{R}^n \rightarrow \mathbb{R}^m$, captures the functional relationship from the state space to the control space, transitioning from the number of states, n, as inputs, to the number of control actions, m, as outputs.

2.1.1 The Bellman Equation

A reward function serves as a robust and succinct descriptor of a task, guiding RL algorithms in training the actor towards desired outcomes (Abbeel, 2004). The objective is to refine a policy via the reward function, which maximizes the cumulative reward received. In this instance, the function deployed to optimize the policy is referred to as a cost function, being similar to a reward function in that it specifies desired behaviors. However, it is objectively different since rather than maximizing cumulative rewards, the focus shifts to minimizing the cumulative cost.

The cost function is described using a performance metric, which the learning algorithm aims to minimize. To regularize a system, the performance metric employed is represented in Equation 3,

$$J = \frac{1}{2}\sum_{k=1}^{\infty}(x_k^T Q x_k + u_k^T R u_k) \tag{3}$$

where $Q \in \mathbb{R}^{n\times n}$ is defined as a real symmetric positive semi-definite matrix that penalizes states, while $R \in \mathbb{R}^{m\times m}$ is a real symmetric positive definite matrix that penalizes controls (Kirk, 2004). The cost is influenced by the magnitude of both states and controls, with Q and R dictating the focus of the algorithm by penalizing the inputs to various extents based on the desired task. These matrices ensure that both states and controls are behaving within expectations and serve as parameters to the cost function.

Equation 3 can be adapted to express the cost-to-go, which is the cost to transition between time steps. This formulation, Equation 4, illustrates that the cost described by the cost function encompasses the immediate cost of being in a state and a discounted cost of advancing to the next state.

$$J(x_k) = \frac{1}{2}(x_k^T Q x_k + u_k^T R u_k) + \gamma J(x_{k+1}) \tag{4}$$

$$J^*(x_k) = \min_{u_k}\left(\frac{1}{2}(x_k^T Q x_k + u_k^T R u_k) + \gamma J^*(x_{k+1})\right) \tag{5}$$

The discount factor, γ, has a role in determining the weight of future costs on the value of being in the current state. By applying the principle of optimality, we can transform Equation 4 into Equation 5. This formulation is referred to as the Bellman equation of optimality, representing a discrete-time adaptation of the HJB equation. Utilizing Equation 5, the optimal policy is obtained by selecting the policy that minimizes the Bellman equation, shown in Equation 6.

$$u^* = \operatorname*{argmin}_{u_k}\left(\frac{1}{2}(x_k^T Q x_k + u_k^T R u_k) + \gamma J^*(x_{k+1})\right) \tag{6}$$

2.1.2 HDP: Defining an Optimal Policy

The optimal policy is one to minimize the total cumulative cost incurred throughout the execution of a desired task. This optimization process aims to minimize the cost function defined by the Bellman equation of optimality. To identify a policy that meets the desired criteria, one approach involves taking the derivative of Bellman's equation with respect to the control variable and then solving for this control variable where the derivative equals zero. This establishes the condition for optimality, the partial derivative of the cost function with respect to the control variable must equal zero, as in Equations 7 – 8.

$$0 = \frac{\partial}{\partial u_k}\left(\frac{1}{2}(x_k^T Q x_k + u_k^T R u_k) + \gamma J^*(x_{k+1})\right) \tag{7}$$

$$0 = R u_k + \left(\frac{\partial x_{k+1}}{\partial u_k}\right)^T \nabla\phi^T(x_{k+1})\gamma W^* \tag{8}$$

In the process of computing the derivative, a cost function approximator substitutes the traditional cost function, where $\mathrm{J}(x_k) = W^T\phi(x_k)$, as highlighted in Equation 8. This equation utilizes $\nabla\phi^T(x_{k+1})$, representing the gradient of the basis functions of the approximator, $\phi^T(x_{k+1})$. The weights, W^*, are employed by the approximator along with the basis functions to generate an approximation of the cost.

Further in the derivation, the relationship $\frac{\partial x_{k+1}}{\partial u_k} = G(x_k)$, derived from Equation 1, shows how the optimal policy is dependent on the dynamics of the model. This dependency is used in formulating the optimal policy, shown in Equation 9.

$$u^* = -R^{-1}G(x_k)\nabla\phi^T(x_{k+1})\gamma W^* \tag{9}$$

2.2 Action Dependent Heuristic Dynamic Programming

Action-dependency marks a significant differentiation between HDP and ADHDP, as implied by their names. In HDP, while the control action directly impacts the cost function, as shown by Equation 3, the approximator for the cost is exclusively a function of the states, $J(x_{k+1})$. This is caused by the basis functions, $\phi(x_k)$, which are formulated as monomials of unique state combinations of growing degree.

In contrast, Q-learning introduces action-dependency within these basis functions, $\phi(x_k) \rightarrow \phi(x_k, u_k)$. This inclusion allows control variables, alongside state variables, to create unique monomials of differing degrees. Consequently, the value function in HDP, depicted in

Equation 4, evolves into a more descriptive 'quality' function in ADHDP, transitioning from $J(x_{k+1}) \rightarrow J(x_k, u_k)$, as illustrated in Equation 10.

$$J(x_k, u_k) = \frac{1}{2}(x_k^T Q x_k + u_k^T R u_k) + \gamma J(x_{k+1}, u_{k+1}) \tag{10}$$

Subsequently, the Bellman equation of optimality is adapted to incorporate the quality term, morphing into Equation 11.

$$J^*(x_k, u_k) = \min_{u_k}\left(\frac{1}{2}(x_k^T Q x_k + u_k^T R u_k) + \gamma J^*(x_{k+1}, u_{k+1})\right) \tag{11}$$

Equation 11 shows the objective of Q-learning, serving as the cornerstone of the algorithm's reward function. Employing a linear-in-parameter neural network as a function approximator, Q-learning strives to find a policy that minimizes the cost function. The introduction of action-dependency allows Q-learning to encapsulate more information, thereby eliminating the need to fully know the system dynamics.

2.2.1 ADHDP: Defining the Optimal Policy

The process of identifying an optimal policy in Q-learning initiates with a method similar to the one shown for HDP. However, to incorporate action-dependency, Equation 6 is modified. This adjustment incorporates an action-dependent cost approximator, shown in Equation 12.

$$u^* = \operatorname*{argmin}_{u_k}\left(\frac{1}{2}(x_k^T Q x_k + u_k^T R u_k) + \gamma J^*(x_{k+1}, u_{k+1})\right) \tag{12}$$

In ADHDP, the setup for the derivative process mirrors that outlined in Equation 7, now incorporating the action-dependent approximator. However, proceeding with the derivation presents a challenge. As illustrated in Equation 8, the derivative contains a term intrinsically linked to the system dynamics. This connection requires a modification to Equation 12, turning it into a simpler version depicted in Equation 13.

$$u^* = \underset{u_k}{\operatorname{argmin}}(J^*(x_k, u_k)) \tag{13}$$

Equation 13 shifts from utilizing the derivative of the Bellman equation to derive the optimal policy. Instead, it utilizes a function approximator. As previously mentioned, this approximator is a linear-in-parameter neural network, $J(x_k, u_k) = W^T\phi(x_k, u_k)$. The process of defining an optimal policy proceeds through substitution.

$$0 = \frac{\partial}{\partial u_k}(J^*(x_k, u_k)) \tag{14}$$

$$0 = \frac{\partial}{\partial u_k}\left(W^{*T}\phi(x_k, u_k)\right) = W^{*T}\frac{\partial}{\partial u_k}\phi(x_k, u_k) \tag{15}$$

To obtain an explicit equation for the policy, like in Equation 9, one must solve for the control variables from Equation 15. This method is restricted by the resolution of the basis functions. Some errors may be encountered when the basis functions include the control variables of high degrees, due to the possible introduction of complex or imaginary numbers. Another source of error is observed when there is more than one unique control variable in a basis function term. When finding the derivative and solving for the individual control policies, having control variables together in basis functions will cause control interdependence.

2.3 The Value Iteration Algorithm

Within ADP, a common algorithm used is known as “value iteration”. The algorithm is an iterative training process, value iteration operates as a closed-loop algorithm through two stages, the q-value update step and the policy improvement step (Lewis, 2009). The steps of the algorithm are delineated in Equations 16 – 17, respectively, establishing a structured approach that refines policies towards optimality.

$$W_{j+1}^T\phi(x_k, u_k) = \frac{1}{2}(x_k^TQx_k + u_k^TRu_k) + \gamma W_j^T\phi(x_{k+1}, u_{k+1}) \tag{16}$$

$$h_{j+1}(x_k) = \underset{u_k}{\operatorname{argmin}}\left(W_{j+1}^T\phi(x_k, u_k)\right) \tag{17}$$

The variable j functions as an iterative counter throughout the training process, indicating the current values of the iteration. These values transition to $j + 1$ upon the completion of the current iteration's update steps. Training commences with the initialization of the approximator weights, typically utilizing random values. Subsequently, policy functions, as derived from Equation 17 using these randomized weights, are employed to formulate a preliminary policy. This initial policy is then applied in Equation 16 to compute a q-value. The calculated q-value is important in adjusting the weights. Due to the weights being employed in q-value prediction, Equation 16, and in evaluating the policy, Equation 17, changes to the weights are linked to changes in both the policy and the Q-function.

The weights are updated using the least-squares method. This update mechanism is formalized in Equation 18, which shows the relationship between the basis functions, the q-value, and the adjustment of weights.

$$W_{j+1} = \left(\phi(x_k, u_k)^T\phi(x_k, u_k)\right)^{-1}\left(\left(\frac{1}{2}(x_k^TQx_k + u_k^TRu_k) + \gamma W_j^T\phi(x_{k+1}, u_{k+1})\right)^T\phi(x_k, u_k)\right) \tag{18}$$

In Q-learning, updating the weights serves to refine the Q-Value function approximator directly. Given that the policy is linked to the function approximator, any modification to the weights concurrently updates the policy as well. This iterative training cycle proceeds until either of two conditions is met: the process reaches the maximum allotted number of iterations, or the discrepancy between the weights across successive iterations falls beneath a predetermined threshold.

The concise Q-value iteration algorithm is:

1. Randomly generate $x_k \in \Omega$, and $u_k \in \Psi$.
2. Generate the initial weights, W_0.
3. Obtain x_{k+1} from simulation or neural network trained using gathered data.
4. Use x_{k+1} into the policy functions derived from Equation 17 to get u_{k+1}.
5. Calculate Q-value of x_k and u_k, using x_{k+1} and u_{k+1} in Equation 16.
6. Update $W_j \rightarrow W_{j+1}$ using Equation 18.
7. Repeat steps 4-6 until the difference in magnitude of W_j and W_{j-1} is smaller than E.

Table 3. Value iteration table of variables.

Variable	Description
k	Time step index
j	Loop iteration index
x_k	Random states
u_k	Random controls
Ω	State domain of training
Ψ	Control domain of training
x_{k+1}	Future states
u_{k+1}	Future controls
W_j	Weights for neural network
W_{j+1}	Update to the weights
W_{j-1}	Weights of the previous iteration
E	Error threshold

CHAPTER III

INVERSE REINFORCEMENT LEARNING ALGORITHM

3.1 Inverse Reinforcement Learning

Elaborating further on the Inverse Reinforcement Learning (IRL) description discussed in Chapter 1, IRL can be succinctly described as an algorithm within an algorithm. It encompasses a Reinforcement Learning (RL) algorithm that trains to find an optimal policy, nested within the IRL algorithm that modifies the reward function used by the inner RL. Figure 4 illustrates the feedback loop created by the IRL algorithm.

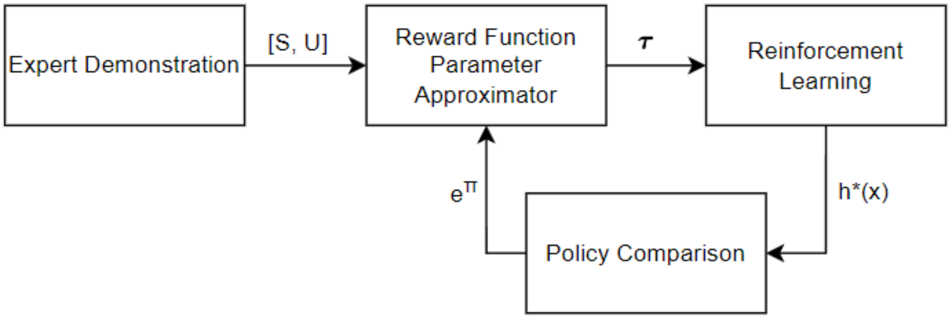


Figure 4. Basic flowchart of inverse reinforcement learning.

An expert demonstration serves as the input for the algorithm, represented as a sequence of state and control pairs: $(s_1, u_1), (s_2, u_2), \dots (s_p, u_p) \rightarrow [S, U]$. This demonstration comprises a list of states and controls recorded from an 'expert'—someone proficient in performing the desired task. The states in the expert demonstration are descriptive characteristics of an agent, captured by sensors. For example, in the context of a vehicle, states could include position, velocity, and acceleration, which directly describe the dynamics of the vehicle. Other states may encompass environmental data, such as that obtained from LIDAR and cameras. The controls used in the algorithm are straightforward: they are the recorded actions of devices used to influence the system. In the case of a vehicle, these controls would include the steering wheel, brake pedal, and throttle pedal, among others.

An approximator utilizes the expert demonstration to predict a set of parameters, denoted as τ. This approximator is a deep neural network, designed such that its input layer corresponds to the number of samples in the demonstration. The architecture of the hidden layers is determined experimentally to optimize learning. Furthermore, the output layer of the approximator is configured to match the number of parameters required for the reward function in the RL algorithm.

$$J(x,u) = \tau_1\phi_1(x,u) + \tau_2\phi_2(x,u) \dots \tau_b\phi_b(x,u) \tag{19}$$

$$e^{\pi} = u^{*} - u^{e} \tag{20}$$

The reward function for the RL algorithm is parameterized as outlined in Equation 19. This parameterization helps IRL identify suitable parameters that accurately reflect the behavior of the expert. In Equation 19, ϕ represents combinations of states and controls that form the basis of the reward function. The parameters employed in ADP would be Q and R, as detailed in Equation 13. Utilizing this adjusted reward function, the RL algorithm seeks to approximate an optimal policy in the form described in Equation 2. The final step in this loop involves comparing the policy predicted by RL, denoted as u^*, with the policy recorded from the expert, u^e. The discrepancy between these policies, e^{π} (referring to the error, e, between the policies, π, not to Euler's constant), is used to update the parameter neural network, enhancing its predictions for subsequent iterations. This iterative process continues until the error between the policies falls below a predetermined threshold, or the specified number of iterations is reached.

A basic IRL algorithm involves the following steps:

1. Randomly generate the trainable parameters for the IRL parameter neural network, $NN(x, u)$.
2. Input the expert demonstration into the neural network, $NN(S, U)$, to obtain parameters, τ.
3. Use τ to update the reward function in the RL algorithm.
4. Obtain u^* from the RL algorithm.
5. Calculate e^{π} by comparing u^* and u^e (the policy recorded from the expert).
6. Update $NN(x, u)$ based on the error.
7. Repeat steps 2—6 until the error e^{π} falls below a predetermined threshold.

3.1.1 Gradient Based Learning

Gradient Based Inverse Reinforcement Learning (GBIRL) is an IRL method that derives a reward function by utilizing the gradients of a cost function. Unlike in RL or ADP, where a cost function explicitly describes a behavior or task, in GBIRL, the cost function calculates the discrepancy between a target value and a predicted value. Within the context of IRL, the target value corresponds to the expert demonstration, while the predicted value refers to the trajectory and policy generated using RL. Examples of cost functions used in gradient-based learning are detailed in Equations 21 and 22.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(D_i - P_i)^2 \tag{21}$$

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|D_i - P_i| \tag{22}$$

Equation 21 illustrates a loss function known as mean squared error, which calculates the average of the squared differences between the demonstration, D, and the prediction, P. This function is particularly useful when there is a need to heavily penalize larger errors. The squaring operation causes larger errors to have a disproportionately greater impact, biasing the algorithm towards minimizing errors of larger magnitude. Distinctly, Equation 22 introduces the mean absolute error equation. As the name suggests, this function calculates the mean of the absolute differences between D and P. This cost function is advantageous when all errors should be considered equally, avoiding any bias that might amplify the effect of outliers. This attribute is especially valuable in scenarios where outliers are present, and where the mean squared error might otherwise lean the learning process towards these outliers.

3.1.2 Algorithm Optimizers

Gradient-based learning updates the weights of a neural network by using the gradient of the selected cost function with respect to the weights. The process of optimizing and updating these weights is governed by gradient optimizers. These optimizers are algorithms designed to minimize the cost function by adjusting the weights of the neural network. There are several such algorithms, each with varying degrees of effectiveness depending on the context. The simplest among them is gradient descent (GD) , which is considered the foundational method of gradient-based optimization. GD straightforwardly employs the gradients of the loss function, as illustrated in Equation 23.

$$W_{j+1} = W_j - \alpha \frac{\partial L}{\partial W_j} \tag{23}$$

Here, W_j represents the set of weights used in the current training iteration, while W_{j+1} depicts the set of weights to be used in the next iteration. The derivative of the loss function indicates how the loss changes in response to adjustments in the weights of the neural network. These derivatives form a Jacobian vector, containing the partial derivatives of the loss function with respect to each individual weight. It is multiplied by the learning rate, a variable that dictates the extent to which the gradient influences the current weights. A higher learning rate accelerates the learning process but may compromise accuracy. On the other hand, a lower learning rate enhances accuracy at the cost of increased learning time. Variables like the learning rate that define specifics of the learning process are known as hyperparameters.

A variant of gradient descent, known as momentum or gradient descent with momentum, was developed to accelerate the optimization process. As suggested by its name, gradient descent with momentum enhances the standard gradient descent technique by incorporating a momentum term. This term involves using exponentially weighted averages of past gradients to update the weights, which helps in smoothing out the updates and potentially speeding up convergence. The equations governing this optimization technique can be found in Equations 24 and 25.

$$v_{dW_j} = \beta v_{dW_{j-1}} + (1-\beta)\frac{\partial J}{\partial W_j} \tag{24}$$

$$W_{j+1} = W_j - \alpha v_{dW_j} \tag{25}$$

In this context, v_{dW_j} represents the exponentially weighted average of the gradients, which is used for updates instead of the raw gradient typically employed in standard gradient descent. An additional hyperparameter, β, is introduced to control the extent of averaging. This parameter determines the number of past gradient values to average, effectively setting the scope of iterations considered. A β close to 1.0 increases the number of iterations factored into the average, thereby reducing the influence of the most recent iteration. This method generally accelerates the optimization process compared to traditional gradient descent.

Another widely used GD method is called Root Mean Square Propagation, or RMSProp. Like momentum, RMSProp is designed to expedite the optimization process of GD. This gradient-based method employs a similar formula to that found in Equation 24 to calculate the exponentially weighted average of the gradients. However, RMSProp distinguishes itself by utilizing the square of the gradient, as detailed in Equation 26.

$$s_{dW_j} = \beta s_{dW_{j-1}} + (1-\beta)\left(\frac{\partial J}{\partial W_j}\right)^2 \tag{26}$$

$$W_{j+1} = W_j - \alpha \frac{\partial L / \partial W_j}{\sqrt{s_{dW_j}} + \varepsilon} \tag{27}$$

RMSProp modifies the weight update mechanism distinct from that used in momentum. Instead of directly multiplying the gradient by the learning rate, as is done in standard GD, or combining the weighted averages with the learning rate as in momentum, RMSProp updates the weights by multiplying the learning rate with the gradient divided by the square root of the exponentially weighted average of the squared gradients. This is depicted in Equation 27. To ensure numerical stability and prevent division by zero, a small constant, ε, is added to the denominator.

The method selected for this research combines features of both gradient descent with momentum and RMSProp in an algorithm known as Adaptive Moment Estimation, or Adam. Adam integrates the principles from Equations 24, 26, and 27 but introduces a unique modification to the weight update process. This key difference includes a bias correction step, which adjusts the updates to account for the initial bias toward zero in the moment estimates when the moving averages are being initialized, ensuring more accurate calculations as training begins (John, 2021).

$$v_{dW_j} = \beta_1 v_{dW_{j-1}} + (1-\beta_1)\frac{\partial J}{\partial W_j} \tag{28}$$

$$s_{dW_j} = \beta_2 s_{dW_{j-1}} + (1-\beta_2)\left(\frac{\partial J}{\partial W_j}\right)^2 \tag{29}$$

$$v_{dW_j}^{corrected} = \frac{v_{dW_j}}{1-(\beta_1)^t} \tag{30}$$

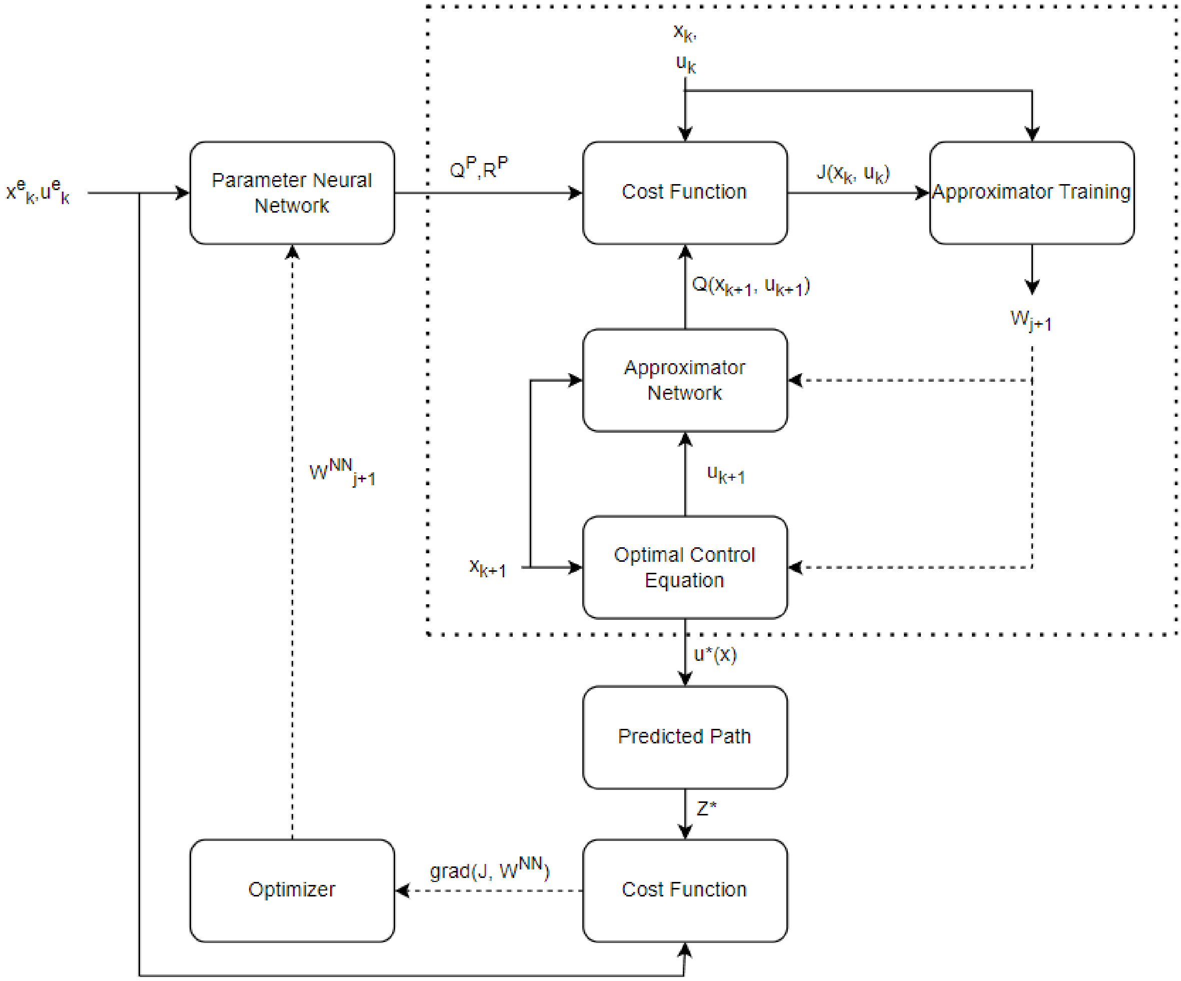
$$s_{dW_j}^{corrected} = \frac{s_{dW_j}}{1-(\beta_2)^t} \tag{31}$$

$$W_{j+1} = W_j - \alpha \frac{v_{dW_j}^{corrected}}{\sqrt{s_{dW_j}^{corrected}} + \varepsilon} \tag{32}$$

Equations 28 and 29 incorporate different β parameters, each representing a distinct hyperparameter of the Adam algorithm. Each β is specifically used for bias correction of the corresponding weighted averages. To update the weights, Equation 32 is employed, where the adjusted term involves the learning rate. This term is the corrected exponentially weighted averages divided by the square root of the exponentially weighted squared averages, with a small constant, ε, added for numerical stability. Equation 28 represents the first moment, and Equation 29 the second moment, which justifies the 'adaptive moment' aspect of the algorithm's name.

3.2 Gradient Based Inverse Reinforcement Learning

To fully grasp the gradient descent inverse reinforcement learning algorithm, it is essential to comprehend the role of each equation. The initial and pivotal step in this algorithm involves defining a parameterized reward function. For instance, in Equation 10, the parameters would be the values used in the Q and R matrices. The algorithm then utilizes two distinct sets of recorded values. The expert demonstration, represented as (x^e, u^e), serves as the primary input of the algorithm and illustrates the behavior that the reward function aims to encapsulate. The second set of data, (x, u), is employed within the reinforcement learning algorithm to understand how control actions influence the states of the system. This dataset is typically much larger to ensure it is comprehensive enough for the algorithm to learn the behavior of the system under various conditions.



x^e_k, u^e_k
Parameter Neural Network
Q^P, R^P
x_k, u_k
Cost Function
J(x_k, u_k)
Approximator Training
Q(x_k+1, u_k+1)
W_j+1
Approximator Network
W^NN_j+1
u_k+1
x_k+1
Optimal Control Equation
u*(x)
Predicted Path
Z*
Optimizer
grad(J, W^NN)
Cost Function

Figure 5. GBIRL Flowchart.

To initiate the process, the expert demonstration is input into a neural network to generate a set of parameters for the reward function, denoted as Q^p and R^p. These parameters are critical for the RL segment of the algorithm, which is highlighted within the dotted square in the diagram.

Within this RL segment, the weights of the approximator network are initially set to small random values. These weights play a crucial role in the optimal control equation. The training data employed here helps the agent learn by distinguishing between current and future states. The future states, represented as x_{k+1}, are utilized in the control equation to predict future controls, u_{k+1}, or the set of controls that would be applied if the system were in that future state. These future state-control pairs, along with the current state-control pairs, (x_k, u_k), are integrated into the cost function to calculate a Q-value.

The Q-values, combined with the basis functions of the approximator evaluated at the current state-control pairs, are employed in a least squares regression to update the weights of the RL function approximator. This forms an iterative process that continues until a predefined number of training iterations are completed. By the end of the training loop, the optimal control function utilizes the trained weights, concluding the RL portion of the algorithm.

At this time, the optimal control function is applied together with the expert states to derive an optimal control policy, u^*. Alternatively, a trajectory, Z^*, originating from an initial state and guided by the optimal policy function, illustrates how the agent would operate autonomously, diverging from a predetermined set of states. This trajectory is compared against the expert demonstration using a loss function that needs minimization. This minimization is achieved through an optimization algorithm like Adam, which adjusts the neural network's performance by leveraging the gradients of the loss function to reduce its output values. This closed-loop method continues until either a set number of iterations have been completed or a performance criterion, typically a loss function magnitude threshold, has been met. This entire process is depicted in Figure 5. Upon completion of this learning phase, the optimal control should effectively replicate the behavior demonstrated by the expert.

CHAPTER IV

SIMULATIONS AND RESULTS

4.1 Simulation Overview

The RL and IRL algorithms were evaluated using a benchmark known as the Van der Pol oscillator before being applied to a more complex simulated environment, CARLA (Car Learning to Act). The testing of these algorithms was conducted in stages. Initially, the RL algorithm was tested independently to verify that it performs according to certain expectations. One such expectation is a behavioral plot demonstrating the ability of the agent to learn from the consequences of its actions on itself and the environment. If the agent fails to display a clear understanding of the cause and effect of its actions, another significant indicator of performance is the convergence of the weights during the learning process. Although weight convergence signifies a successful learning process, it does not necessarily confirm that the behavior learned was as intended. This is particularly true since the RL algorithm was tested with randomly initialized parameters in the reward function, which may not always yield the desired behavior, especially in complex systems. Therefore, demonstrating weight convergence remains a crucial metric to assess the efficacy of the algorithm.

4.2 Van der Pol Overview

The Van der Pol oscillator is commonly used as a benchmark for control algorithms due to its inherent system characteristics. It is a harmonic oscillator that includes a nonlinear damping component, enabling it to exhibit self-sustained oscillations. The degree of nonlinearity in the system is determined by a parameter comparable to a coefficient of friction (Boccara, 2007). This oscillator is defined by a nonlinear, second-order differential equation, as shown in Equation 33.

$$\frac{d^2x}{dt^2} - \mu(1 - x^2)\frac{dx}{dt} + x = 0 \tag{33}$$

The nonlinear friction term in Equation 33 is represented as $\mu(1 - x^2)$, where μ is termed the 'coefficient of friction'. This second-order differential equation is converted into two first-order differential equations, as shown:

$$\frac{dx_1}{dt} = x_2 \tag{34}$$

$$\frac{dx_2}{dt} = \mu(1 - x_1^2)x_2 - x_1 \tag{35}$$

The Van der Pol oscillator is employed as a benchmark in control systems testing primarily due to its chaotic and nonlinear nature. Even minor variations in initial conditions can lead to markedly different system trajectories. Sample demonstrations that depict this behavior can be seen in the appendix. For an agent to effectively control such a system and steer it to a desired state, it must be capable of accurately predicting its behavior at any given point. The inherent chaos and the requisite ability to model nonlinear dynamics underline why the Van der Pol oscillator serves as an essential preliminary benchmark to evaluate the effectiveness of a control algorithm before advancing to more complex systems.

4.2.1 Van der Pol Reinforcement Learning Overview

The phase diagram of a Van der Pol oscillator is illustrated in Figure 6. It depicts the system's undisturbed oscillations, highlighting how the velocity is dependent on its position, with the motion forming a repetitive spiral over a predetermined duration. The degree of nonlinearity intensifies with the increase in the 'friction coefficient.' Additionally, the representation of the states as functions of time, rather than of each other, is shown in Figure 7.

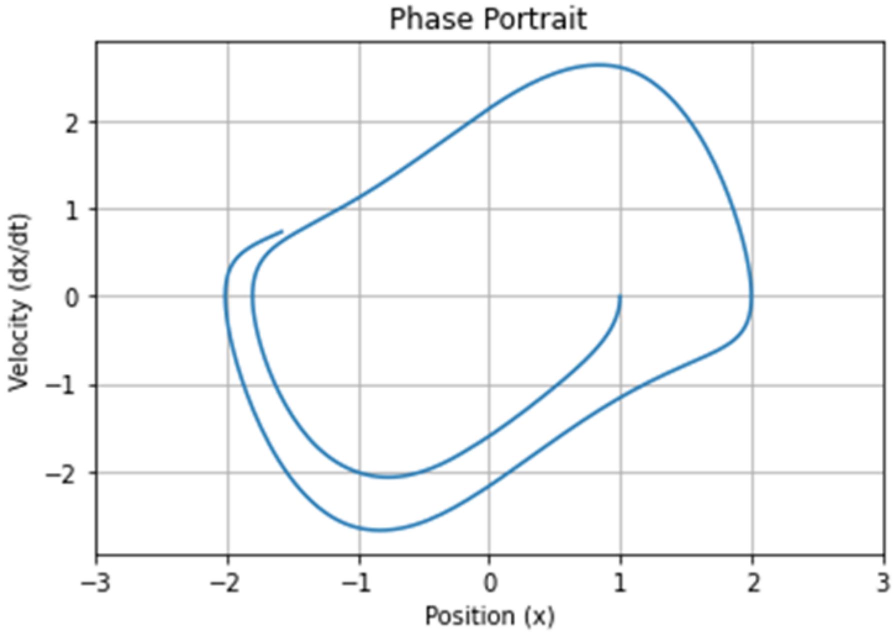


Figure 6. Phase diagram of undisturbed oscillator.

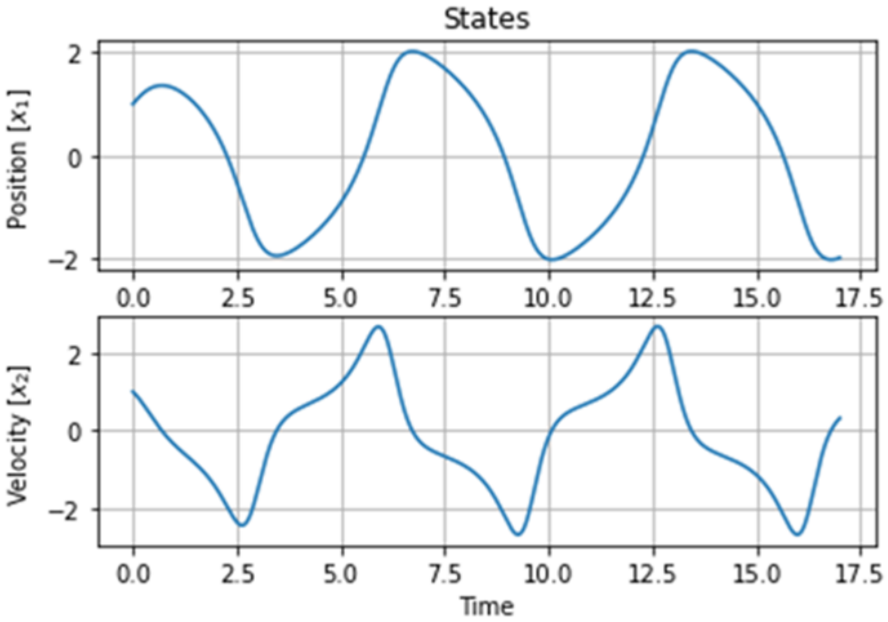


Figure 7. Undisturbed states as functions of time.

The controller is tasked with steering the states towards their desired values. In the case of the Van der Pol controller, these desired state values are set to zero, as reflected in the selected regularization cost function. Detailed specifications of the parameters and controller settings are provided in Table 4.

Table 4. Q-Learning parameters.

Q-Learning Controller	
Number of Patterns	1000
Quantity of Basis Functions/Neurons	17
Training Iterations	200
Convergence Threshold	1×10^{-4}
Time Step	0.01
State Penalizing Matrix	$\begin{bmatrix} 5000 & 0 \\ 0 & 2500 \end{bmatrix}$
Control Penalizing Matrix	$[100]$
Domain of Training	State Domain $(x, \dot{x})$: $[-2 \quad 2]$ $\left(m, \frac{m}{s}\right)$ Control Domain (u): $[-10 \quad 10]$

The parameters outlined in Table 4 detail the training process for the controller. The 'number of parameters' refers to the count of parameters utilized per iteration in the vectorized code, which facilitates faster learning by exposing the agent to a greater variety of states per iteration. However, this approach also increases computational demands as the number of patterns rises. Too high a number of patterns can counterproductively slow down the learning process.

The basis functions consist of unique combinations of the system's variables. In this controller, 17 basis functions are obtained from unique combinations of position, velocity, and control variables. Due to the significant number of patterns, the algorithm tends to converge relatively quickly, enabling the use of a smaller number of training iterations. If the convergence threshold is reached prematurely, it indicates that further learning would yield minimal changes in weights, suggesting that convergence has effectively occurred.

The penalizing matrices, square matrices whose dimensions equal the number of controls or states, enforce constraints on the system’s behavior. For the Van der Pol oscillator, which has two states, the state penalizing matrix is a 2×2 matrix. Similarly, the control penalizing matrix is a 1×1 matrix, reflecting the single control input in this setup. The training domain establishes lower and upper bounds for the randomly selected patterns, ensuring the algorithm encounters a broad range of states and actionable decisions within those states.

4.2.2 Van der Pol Reinforcement Learning Results

The training algorithm successfully derived an optimal control law for regulating the Van der Pol oscillator. This law enables the states to converge to zero with minimal control effort, ensuring efficiency in terms of control activity.

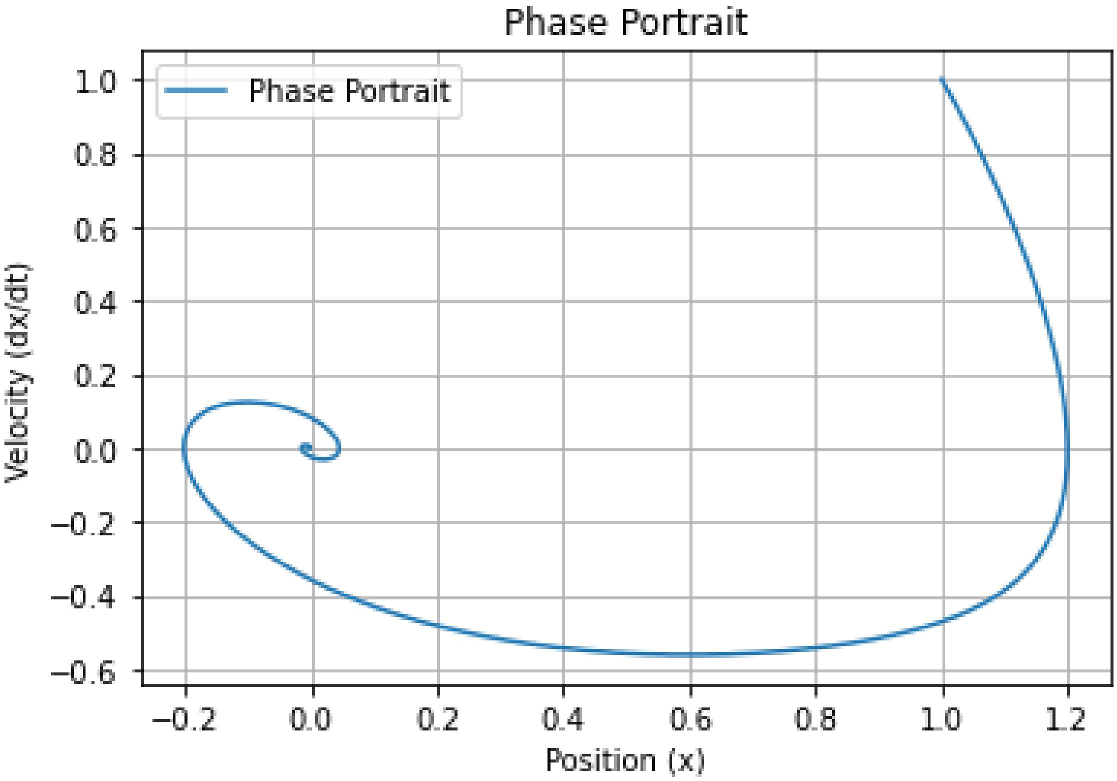


Figure 8. VDP regularized phase diagram.

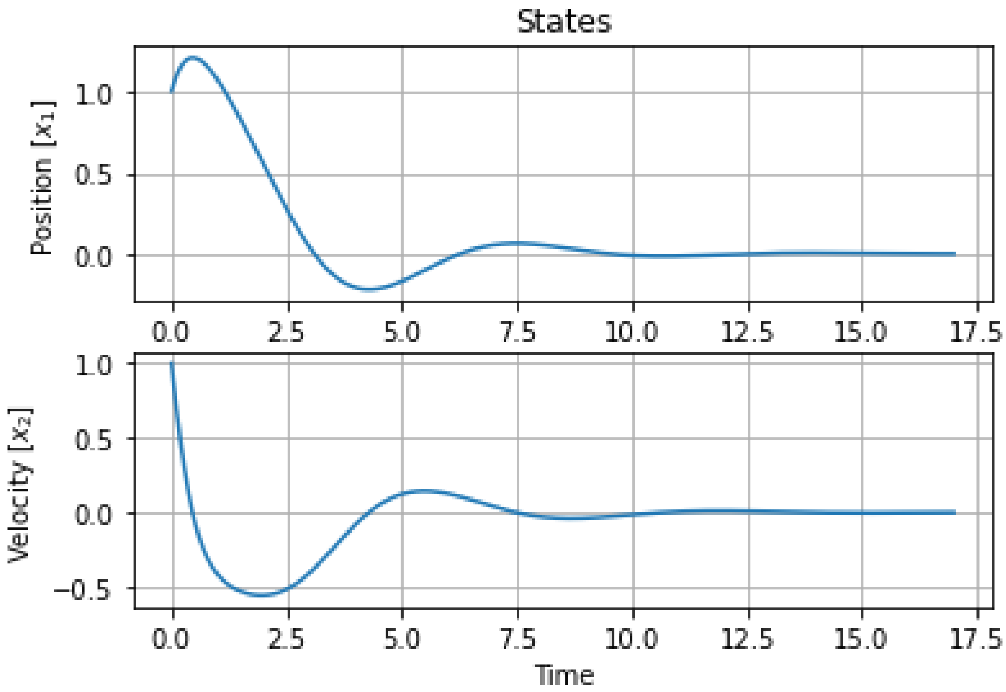


Figure 9. VDP regularized states.

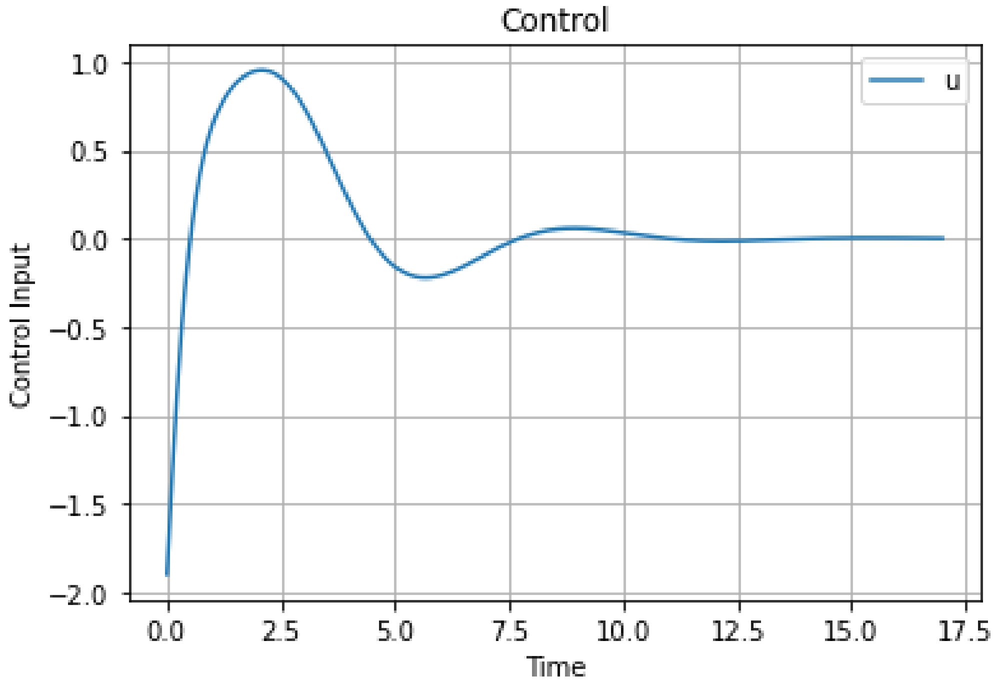


Figure 10. VDP control.

Figures 8—10 display plots that illustrate the regularization of the Van der Pol oscillator system. Figure 9 is particularly informative, showing both position and velocity starting at 1 meter and 1 meter per second, respectively. The phase diagram indicates the onset of oscillation, which rapidly converges towards the origin. The control is active only when the states deviate from their desired positions, but it too converges to zero magnitude once the control objectives are met and further activity becomes unnecessary.

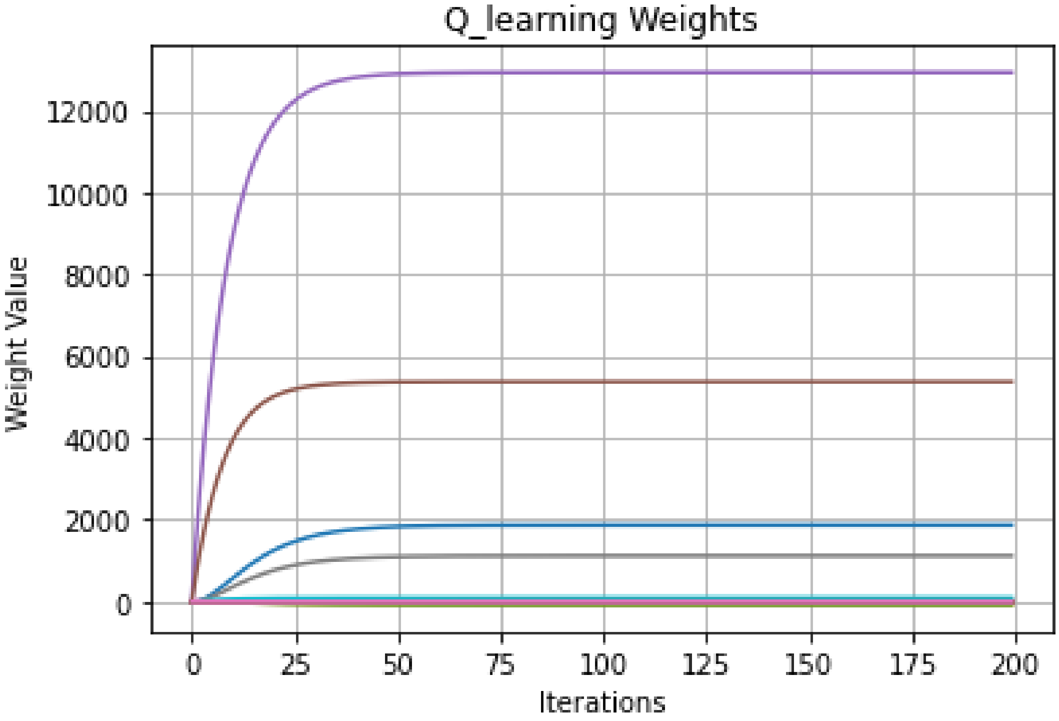


Figure 11. Converged weights of RL process.

Figure 11 illustrates the progression of the weights with each successive iteration, demonstrating how convergence is observed within the algorithm. This behavior is particularly crucial when the reward function employed in training incorporates random parameters to define behavior. Although the states and controls learned may not always reflect the desired behavior accurately, convergence plots like Figure 11 are key indicators that the agent has effectively learned.

4.2.3 Van der Pol Inverse Reinforcement Learning Results

Incorporating the RL algorithm as a function within the IRL framework requires modifications to the penalizing matrices outlined in Table 4. Given that a new neural network's predictions can include negative values, which are incompatible with the requirements of the penalizing matrices, the network was pretrained to ensure it outputs positive values only. Utilizing data from an experiment described in Section 4.2.2, the IRL algorithm successfully predicted state-action pairs that aligned closely with the expert demonstration.

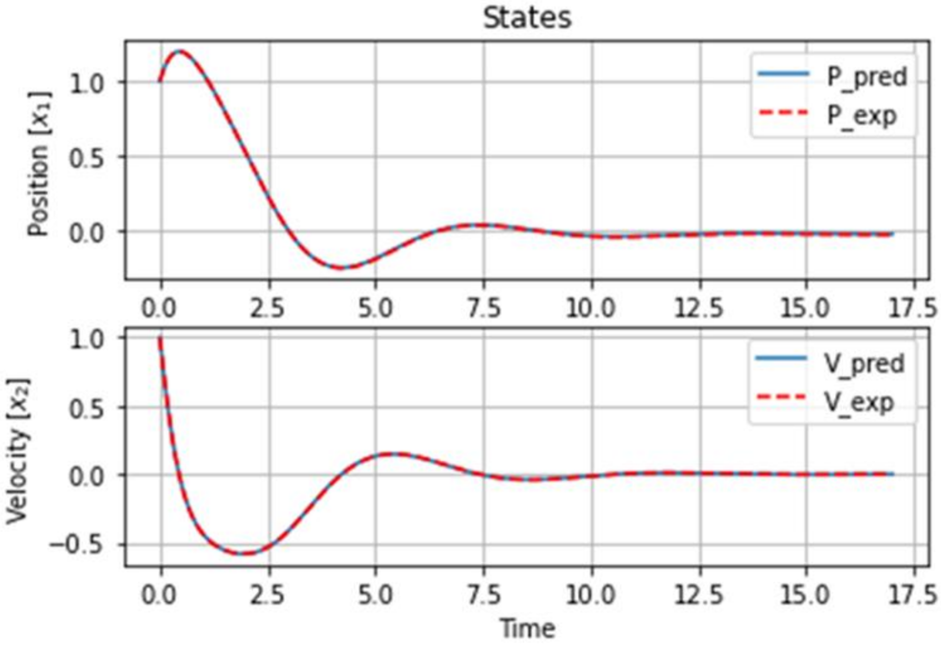


Figure 12. State replication.

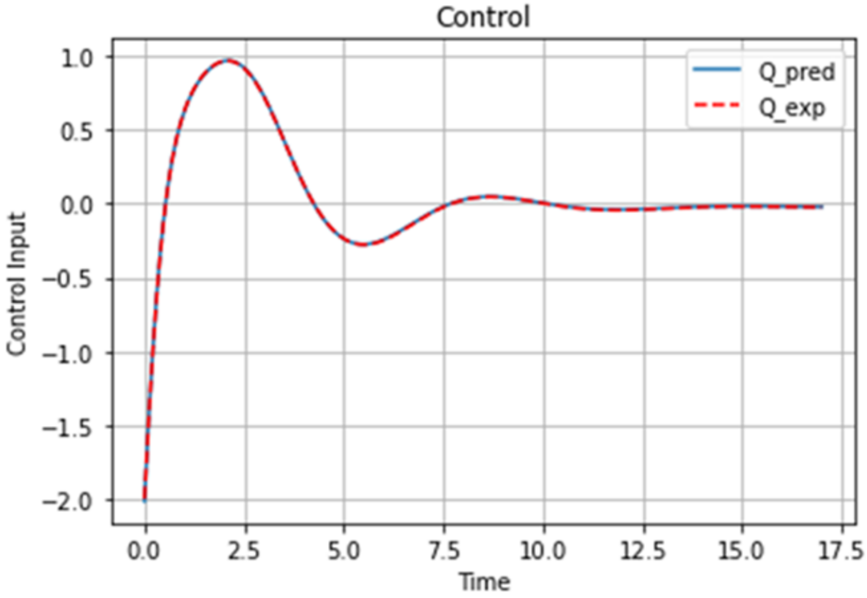


Figure 13. Control replication.

Figure 12 illustrates how the states—position and velocity—align with the recorded expert demonstration. This alignment is a key objective of IRL, as it demonstrates that the behavior of the agent is consistent with that of the expert. However, replicating the states addresses only one aspect of the challenge, as the trajectory also involves control inputs. Figure 13 confirms that the controller successfully replicated the expert's control inputs, effectively reproducing the entire expert demonstration. It is important to note that there is no unique set of parameters that can regulate the Van der Pol dynamics. As shown in Figure 14, the parameters for the reward function varied throughout the training process.

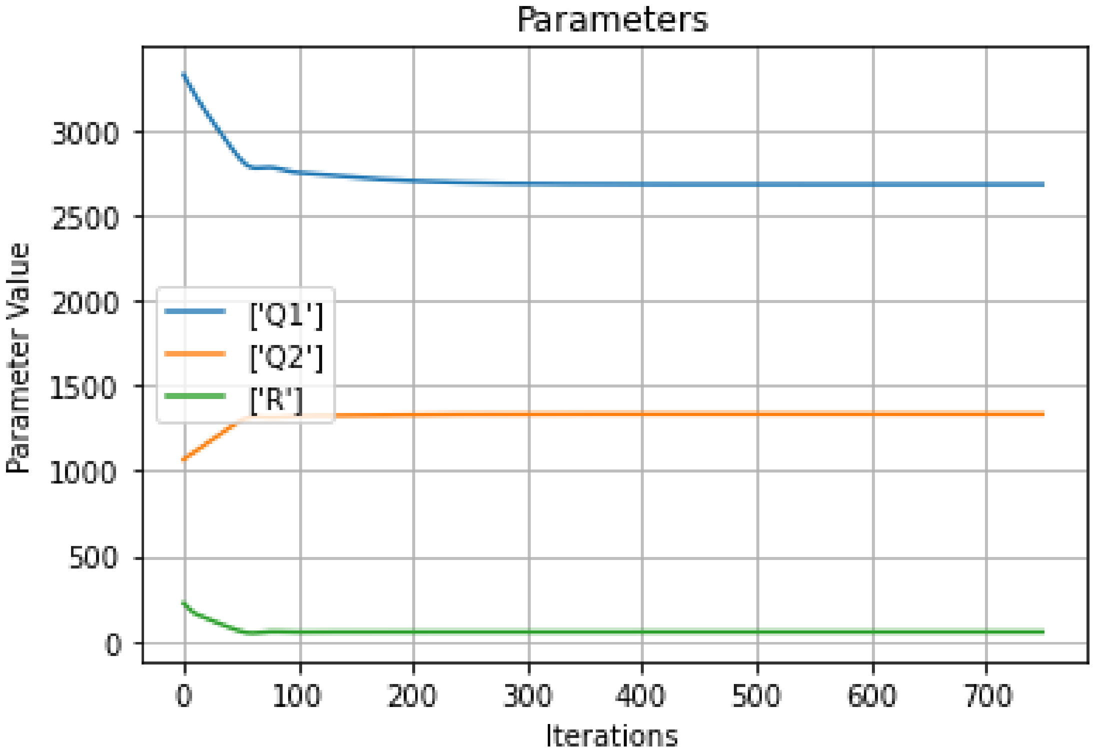
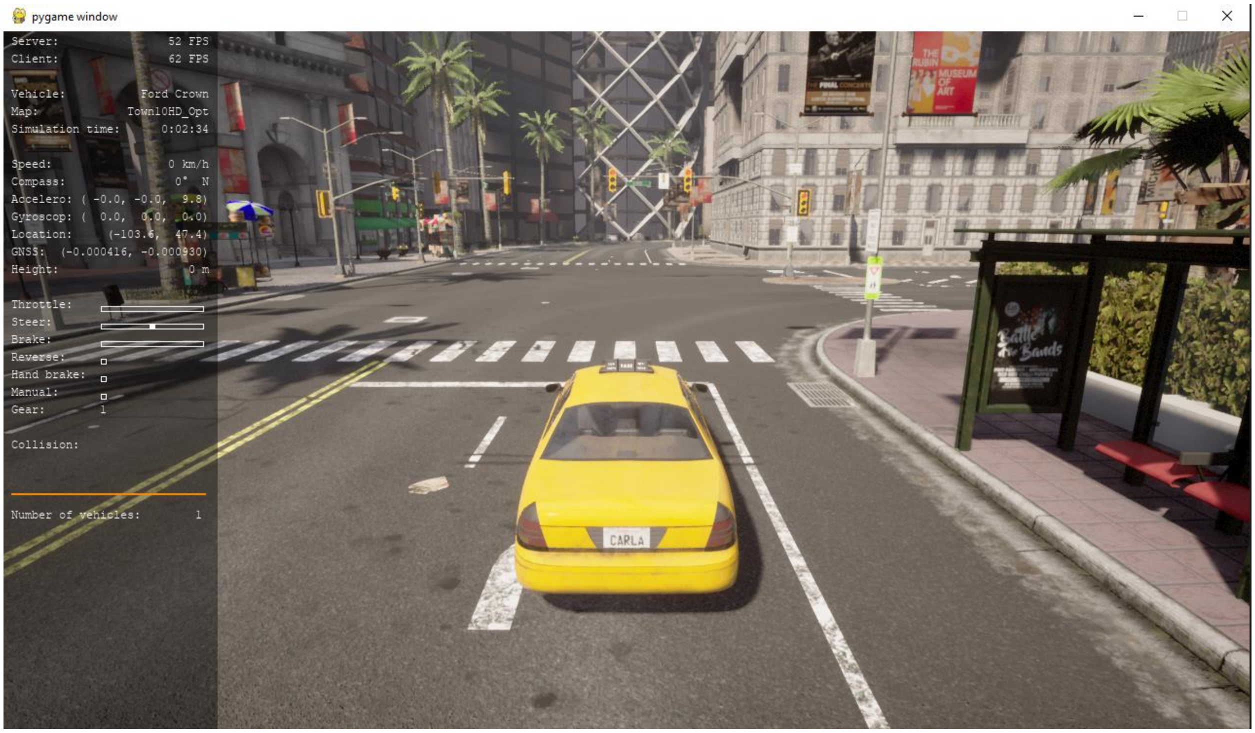


Figure 14. History of IRL predictions.

4.3 CARLA Overview

CARLA is an open-source simulator, available as a Python library, designed for urban-driving scenarios. Its primary purpose is to support the training and validation of autonomous vehicle controllers, although it also facilitates the training of vehicle perception models. To enhance the realism and customizability of urban environments, CARLA offers a diverse array of vehicle models, pedestrians, and road signs (Dosovitskiy, 2017). Moreover, it features options for manipulating weather conditions and time of day, adding variability and challenging visibility scenarios to enrich training sessions. The simulator integrates a wide range of sensors, providing critical data such as position, velocity, and acceleration, as well as specific event-related information like collisions and lane departures. An illustration of CARLA in operation is presented in Figure 15.



```
Server:                  52 FPS
Client:                  62 FPS

Vehicle:             Ford Crown
Map:               Town10HD_Opt
Simulation time:        0:02:34

Speed:                   0 km/h
Compass:                  0°  N
Accelero: ( -0.0, -0.0,   9.8)
Gyroscop: (  0.0,  0.0,   0.0)
Location:      (-103.6,  47.4)
GNSS:  (-0.000416, -0.000930)
Height:                     0 m

Throttle:
Steer:
Brake:
Reverse:      ☐
Hand brake:   ☐
Manual:       ☐
Gear:         1

Collision:

Number of vehicles:          1
```

Figure 15. CARLA application window.

4.3.1 CARLA Setup

To streamline the testing and validation of the control algorithm, the CARLA simulation was simplified to reduce variability and enable the agent to more effectively learn the dynamics of CARLA's vehicle models. The simulation employed a 2020 Dodge Charger vehicle model and utilized the Town04 urban-environment map, which features a road layout similar to an ampersand. Among the available maps, Town04 offers the longest stretch of straight road, as depicted in Figure 16. This characteristic is crucial because a straight road reduces the need for steering control inputs. For the simulation, the sensors employed were the Global Navigation Satellite System (GNSS) and the Inertial Measurement Unit (IMU). The GNSS was used to track the vehicle's longitudinal and latitudinal positions during the drive, while the IMU recorded the vehicle's velocity and acceleration. These data points constituted the states recorded for the expert demonstration.

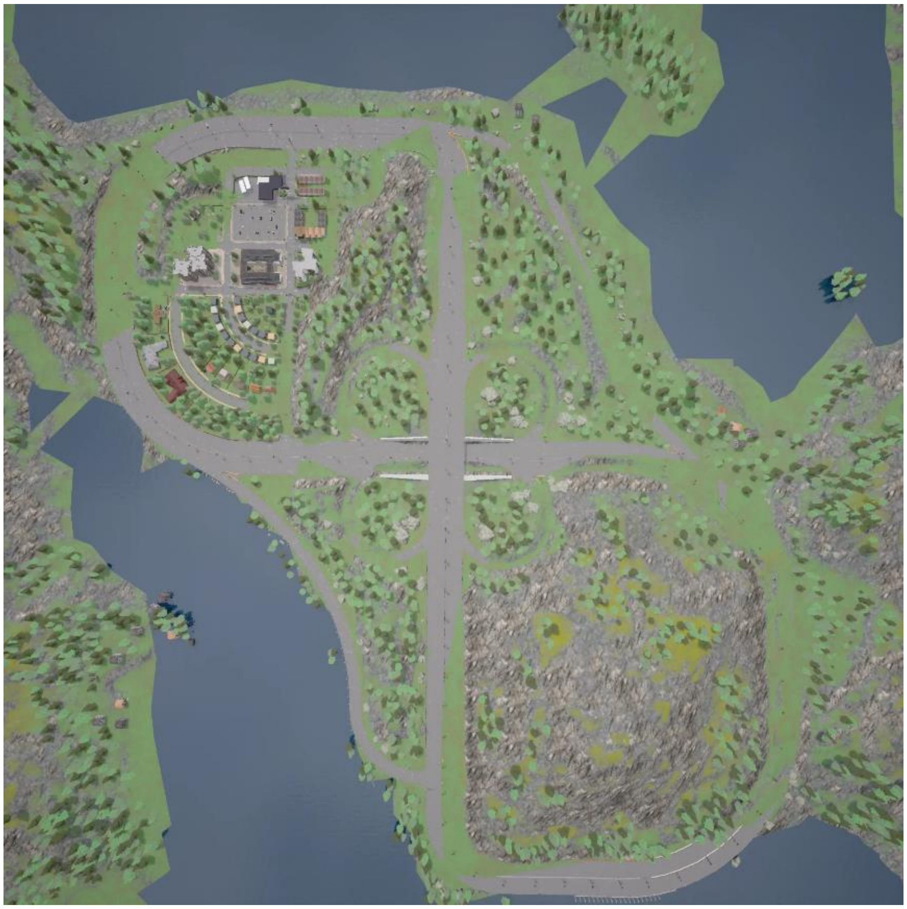


Figure 16. CARLA Town04 (Dosovitskiy, 2017).

The control data was captured using a Logitech G29 Driving Force steering wheel and pedals. CARLA facilitates the integration of these controls through the use of the Pygame Python library, which converts the inputs from the Logitech controller into data that CARLA can process. This setup enables precise recording of the degree to which the steering wheel is turned and the intensity of the pedal presses. The ability to capture both control inputs and vehicle dynamic states simplifies the process of recording expert demonstrations and gathering learning data. Numerous demonstrations were conducted, utilizing various combinations of braking and accelerating at differing rates. The physical setup used for data recording is depicted in Figure 17.

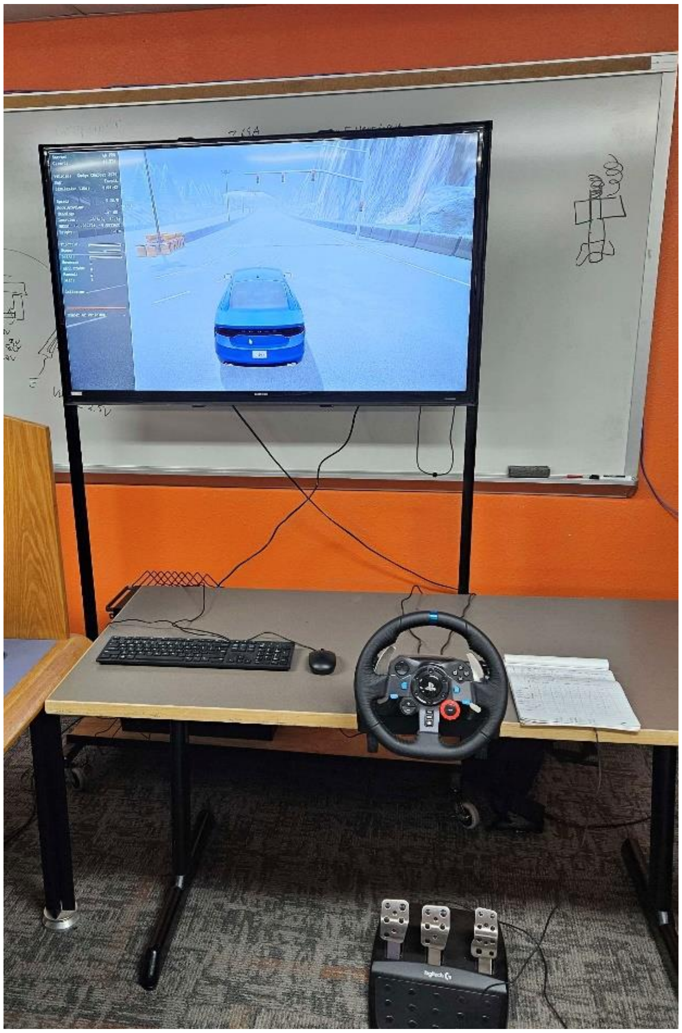


Figure 17. Physical data gathering setup.

The experiment generated several trajectories, accumulating over 100,000 time-steps, each lasting 0.001 seconds. These trajectories were utilized by a distinct neural network within a supervised learning framework. The objective of this neural network was to enhance the excitation in the IRL learning process. It was trained to accept random state-control pairs and predict the subsequent state. This approach could potentially reduce the volume of data required to adequately train the RL agent by leveraging random inputs for training. To validate the effectiveness of this neural network, approximately 10,000 state-action pairs were used, with the results depicted in Figures 18 and 19.

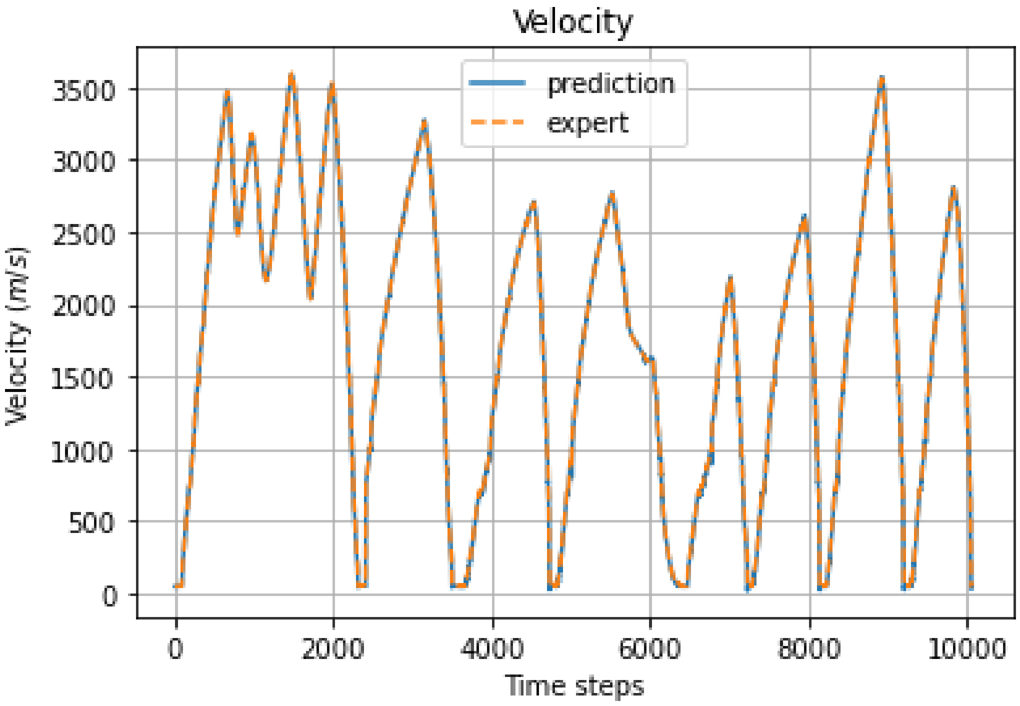


Figure 18. Velocity validation data for neural network.

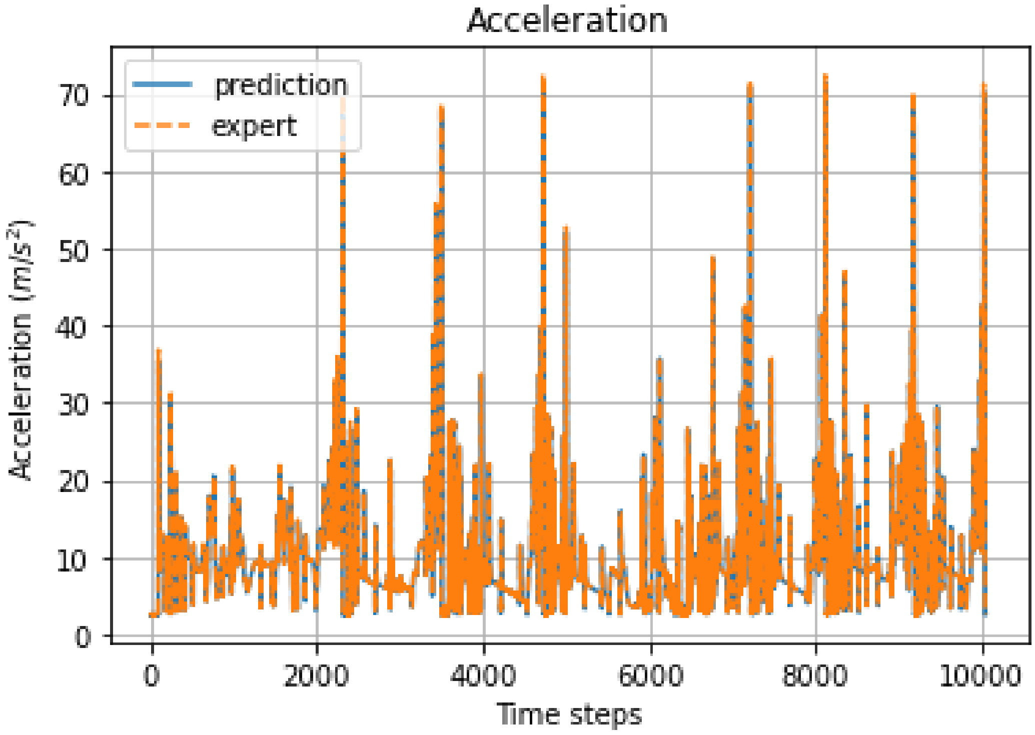


Figure 19. Acceleration validation data for neural network.

Table 5. CARLA IRL algorithm parameters.

CARLA Simulation IRL	
Number of Iterations	100
Number of Network Layers	3
Learning Rate	0.1
CARLA Simulation Q-Learning	
Number of Patterns	10,000
Quantity of Basis Functions/Neurons	88
Training Iterations	200
Convergence Threshold	1×10^{-4}
State Penalizing Matrix	$\begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix}$
Control Penalizing Matrix	$\begin{bmatrix} R_1 & 0 \\ 0 & R_2 \end{bmatrix}$
Domain of Training	Velocity Domain: $[\mu_v, \sigma_v]$ Acceleration Domain: $[\mu_a, \sigma_a]$

Table 5 provides a detailed description of the parameters used on the algorithm. The number of iterations for the IRL model was reduced based on the assumption that further learning was unnecessary for acquiring the desired parameters. However, extending the learning duration is a straightforward adjustment if needed. The neural network, tasked with approximating the parameters of the reward function, consists of three layers in addition to the input layer. This network is trained using a gradient-based optimization method with a selected learning rate of 0.1 to moderate the impact of the gradients.

In comparison with the setup used for the Van der Pol oscillator, the number of patterns for the RL algorithm was increased tenfold to 10,000 random patterns, enhancing the diversity of the learning experience and, consequently, the robustness of the agent. To accommodate the complexity of the system and improve adaptability, the number of basis functions was expanded to 88. This increase is intended to enable the agent to more accurately capture the vehicle dynamics, as reflected by the data obtained from CARLA. Unlike previous implementations, the penalizing matrices were not constants this time, and the control penalizing matrix was enlarged to accommodate two controls.

Additionally, the domain of training for the states and controls was distinctly configured. The recorded state data was analyzed to determine the mean and standard deviation of velocity and acceleration. These statistics were then used to standardize the data, enhancing training effectiveness. Unlike the state data, no normalization was applied to the control values, as they were already scaled between 0 and 1 by the simulator's pedal usage.

4.3.2 CARLA Results

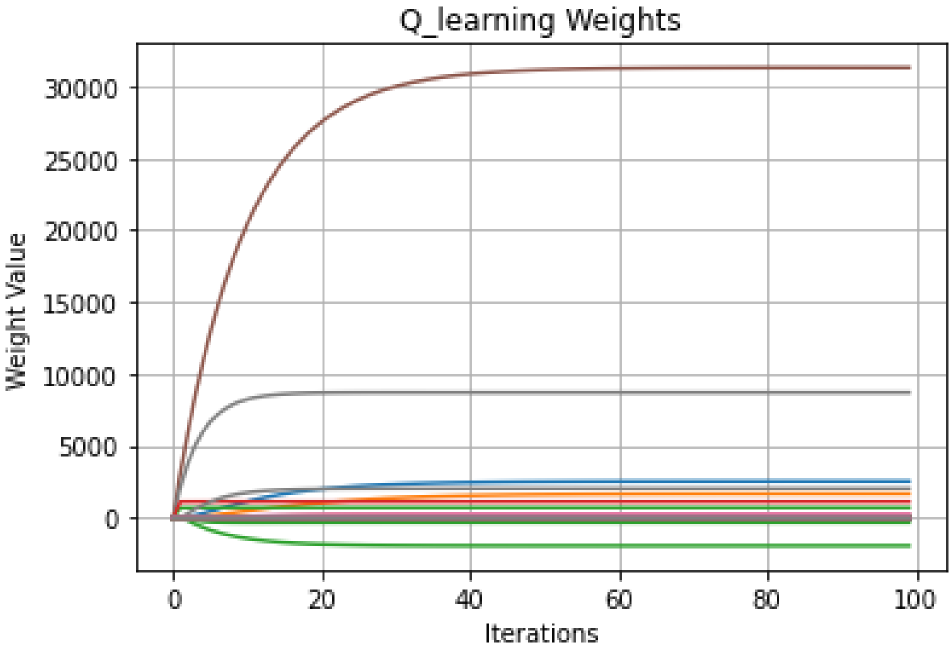


Figure 20. Weight history of RL for the last iteration of IRL.

Figure 20 illustrates the convergence of the RL training, indicating that the Q-learning algorithm has successfully learned the behavior dictated by the reward function. This snapshot captures the final iteration of the IRL algorithm, utilizing a reward function parameterized with the most current prediction data. Notably, some weights remain in the zero range, which is attributable to their specific roles in the model. In the control aspect, only the basis functions that include control terms significantly influence the outcome due to the derivative process involved. For the Q-value function, while all weights are important, the weights most crucial to the control aspects of the model tend to have a more pronounced impact.

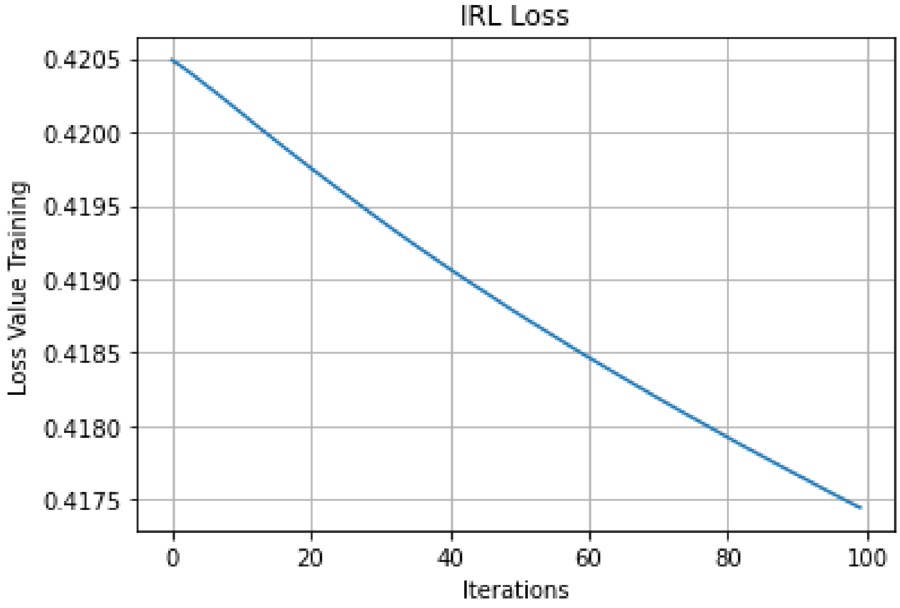


Figure 21. IRL loss history over training.

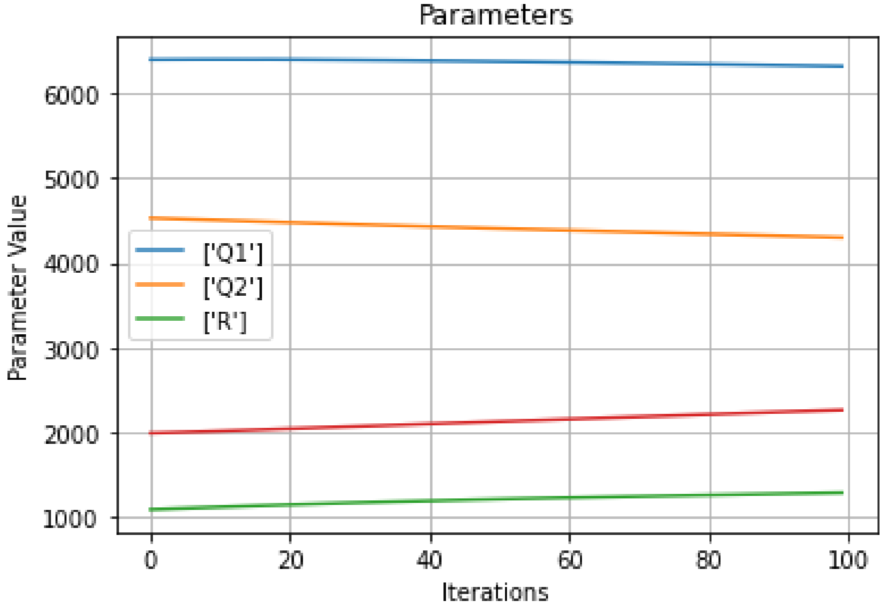


Figure 22. Parameter history of CARLA IRL.

Figure 21 displays the learning loss over the course of the IRL training. The downward trend in loss is a positive indicator of successful training. However, the magnitude of reduction per iteration is notably small, suggesting a slow learning process. This gradual change is further illustrated in Figure 22, which shows only minor adjustments in the parameter history with each iteration. Typically, such slow progress could be addressed by increasing the learning rate. However, as indicated in Table 5, the learning rate is already set at 0.1, which is considered quite high. This suggests a potential limitation within the IRL algorithm itself.

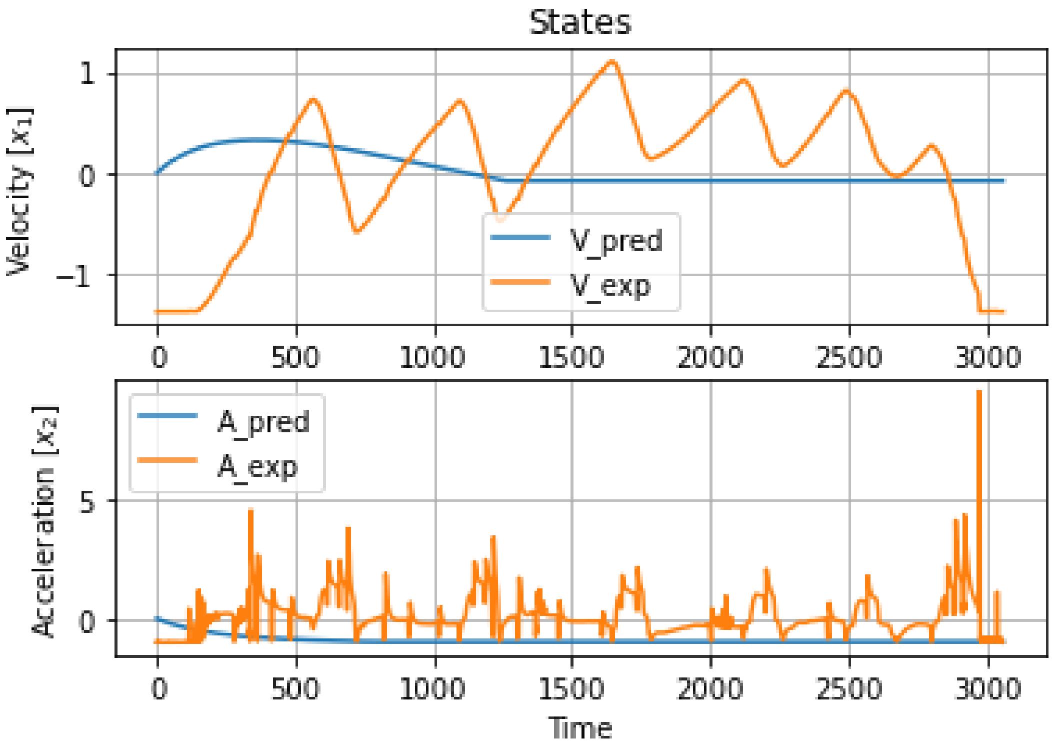


Figure 23. States prediction comparison.

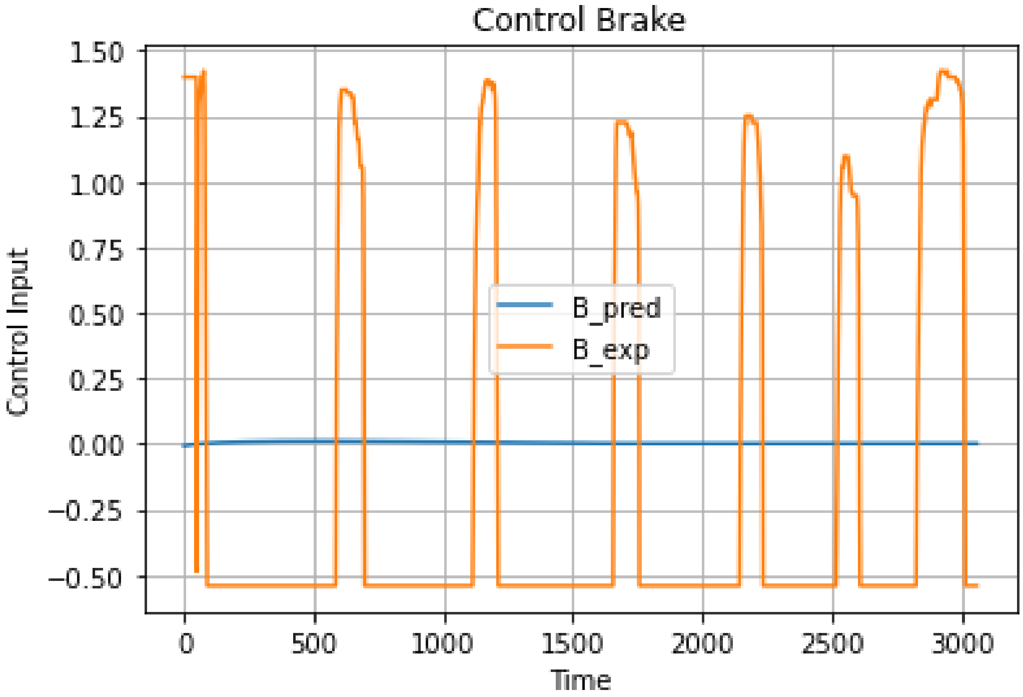


Figure 24. Brake control prediction comparison.

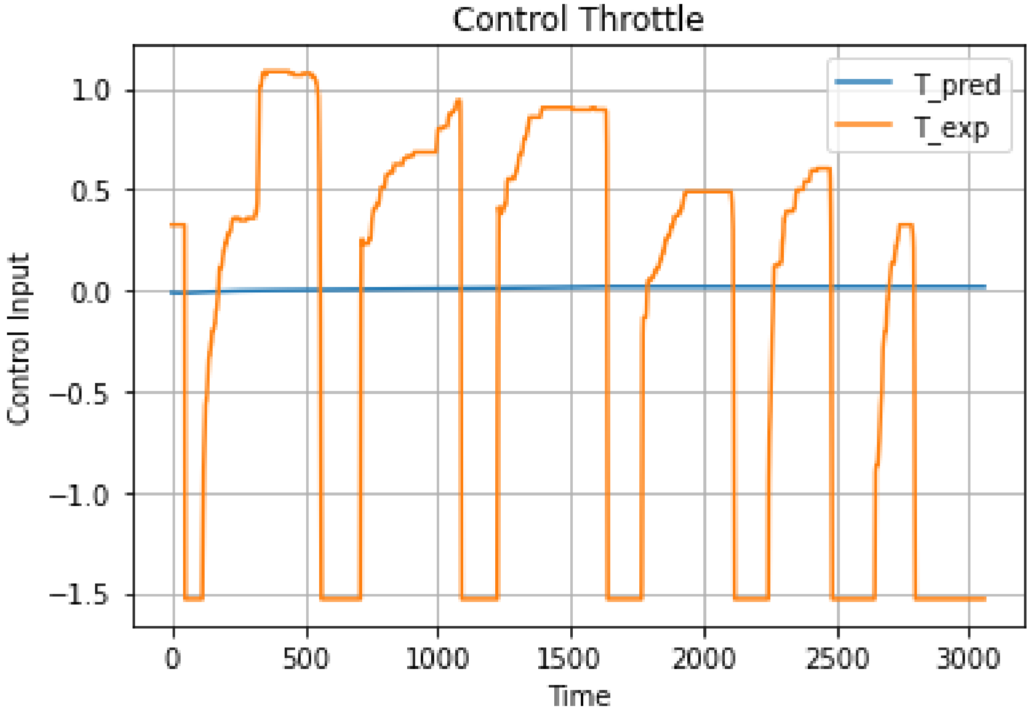


Figure 25. Throttle control prediction comparison.

Figures 23—25 depict the predicted trajectory of the agent. In Figure 23, the expert demonstration is shown in orange, and the predicted states are shown in blue. Unfortunately, the predictions do not align with the desired behavior, which may be indicative of issues with the format of the reward function. The assumption that a regularization function is appropriate might not hold true in this context. Alternative tracking methods were tested, but they yielded similar results. The performance metrics used in this attempt to learn the dynamics of CARLA did not accurately capture the desired behavior. There are indications of a regularization effort, as the velocity, throttle, and brake stabilize near zero values, seen in Figures 23, 24, and 25. However, the velocity not being driven to zero suggests that the parameters of the reward function require further tuning.

CHAPTER V

CONCLUSION

This thesis explores the use of Inverse Reinforcement Learning and Q-Learning to develop a personalized model for autonomous driving. The algorithms were initially validated offline using data from the simulated environment provided by CARLA and further tested on a benchmark system, where they successfully regulated its dynamics. Upon achieving control of the Van der Pol oscillator, the algorithms were applied to the recorded CARLA data. Unfortunately, in this instance, they failed to produce the desired control behavior.

By dividing the testing into the main components of IRL, the identification of potential weaknesses within the overall algorithm became clearer. The Van der Pol oscillator served as a benchmark to demonstrate the RL algorithm’s capability to manage nonlinear chaotic dynamics. Subsequently, the full algorithm was employed to estimate the parameters of a reward function aimed at generating an optimal control function, which would replicate a trajectory equivalent to the expert demonstration.

The limitations of the proposed IRL algorithm became evident when utilizing CARLA simulator data. The data was intended to train the RL algorithm and help the agent understand the effects of its actions within the environment. However, it appears that the linear in parameter (LIP) neural network was inadequate for capturing the complex behavior of the CARLA vehicle model. This issue suggests that the Approximate Dynamic Programming methodology employed may not be robust enough for such complex tasks, particularly because it forces the use of a LIP neural network for deriving the optimal control function. Attempts to use other types of neural networks were quickly aborted due to difficulties in computing their analytical derivatives relative to inputs.

Another potential factor for the unsuccessful control outcomes could be linked to the formulation of the cost or reward function. The performance metrics were designed for regularization, as partially evidenced by the stabilization of velocity, throttle, and brake at near-zero values in the predicted trajectory. Despite testing with other performance metrics, such as distinct tracking methods, no significant improvements were observed. Additionally, there may have been an underestimation of the data required to adequately train the model. Although about 110 thousand distinct state-control pairs were used, the complexity of the problem might require a larger dataset to achieve more accurate and robust learning outcomes.

4.1 Future Works

Once the algorithm performs as desired, the next step is to enhance the complexity of the personalized driving controller by incorporating steering as the third control and position as an additional state. Steering is an important control component in CARLA, necessary for navigating turns, making the inclusion of position data equally as important for recording precise maneuvers. Furthermore, integrating additional sensors such as lane departure and collision detection sensors will enhance the capability of the system to maintain lane integrity and prevent accidents.

The subsequent stage involves incorporating radar or LIDAR, which will enrich the simulation by introducing interactive traffic scenarios. These sensors will be essential in measuring proximity and could serve as new state variables. Such data will enhance the ability for the controller to be personalized, allowing for more detailed decisions regarding braking times and distances between vehicles. Incorporating these sensors not only introduces additional states to the RL problem but also adds complexity to the reward function, ultimately raising the level of personalization achievable by the controller.

REFERENCES

Abbeel, P., Coates, A., Quigley, M., & Ng, A. Y. (2007). An application of reinforcement learning to Aerobatic Helicopter Flight. *Advances in Neural Information Processing Systems 19,* 1–8. https://doi.org/10.7551/mitpress/7503.003.0006

Abbeel, P., & Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. *Twenty-First International Conference on Machine Learning - ICML '04*. https://doi.org/10.1145/1015330.1015430

Adams, S., Cody, T., & Beling, P. A. (2022). A survey of inverse reinforcement learning. *Artificial Intelligence Review*, *55(6)*, 4307–4346. https://doi.org/10.1007/s10462-021-10108-x

Arora, S., & Doshi, P. (2021). A survey of inverse reinforcement learning: Challenges, methods and progress. *Artificial Intelligence, 297*, 103500. https://doi.org/10.1016/j.artint.2021.103500

Asoh, H., Akaho, M. S. S., Kamishima, T., Hasida, K., Aramaki, E., & Kohro, T. (2013, September). An application of inverse reinforcement learning to medical records of diabetes treatment. In *ECMLPKDD2013 workshop on reinforcement learning with generalized feedback*.

Bellman, R. (1957). Dynamic Programming. *Science*, 153, 34 - 37.

Bellman, R. (1966). Dynamic Programming. *Science*, 153(3731), 34–37. http://www.jstor.org/stable/17196

Bethke, B., & How, J. P. (2010). Approximate dynamic programming using model-free bellman residual elimination. *Proceedings of the 2010 American Control Conference*. https://doi.org/10.1109/acc.2010.5530611

Boccara, N. (2007). *Essentials of mathematica: With applications to mathematics and physics*. Springer.

Chung, S.-Y., & Huang, H.-P. (2010). A mobile robot that understands pedestrian spatial behaviors. *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. https://doi.org/10.1109/iros.2010.5649718

Cook, J. D. (2022). Retrieved from https://www.johndcook.com/blog/2019/12/22/van-der-pol/

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., & Koltun, V. (2017, October). CARLA: An open urban driving simulator. In *Conference on robot learning* (pp. 1-16). PMLR.

Gosavi, A. (2015). *Simulation-based optimization*. Springer.

Guiasu, S., & Shenitzer, A. (1985). The principle of maximum entropy. *The mathematical intelligencer*, 7, 42-48.

Imani, M., & Ghoreishi, S. F. (2022). Scalable inverse reinforcement learning through multifidelity bayesian optimization. *IEEE Transactions on Neural Networks and Learning Systems, 33(8)*, 4125–4132. https://doi.org/10.1109/tnnls.2021.3051012

John, J. S. (2021). AdamD: Improved bias-correction in Adam. *arXiv preprint arXiv:2110.10828*.

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). *Reinforcement learning: A survey. Journal of artificial intelligence research, 4*, 237-285.

Kim, K. J., Santos Jr, E., Nguyen, H., & Pieper, S. (2022). An Application of Inverse Reinforcement Learning to Estimate Interference in Drone Swarms. *Entropy, 24(10)*, 1364.

Kirk, D. E. (2004b). *Optimal control theory an introduction*. Dover Publications.

Kolter, J. Z., Abbeel, P., & Ng, A. Y. (2008). Hierarchical apprenticeship learning with application to quadruped locomotion. In *Advances in Neural Information Processing Systems* (pp. 769-776).

Konda, V., & Tsitsiklis, J. (1999). Actor-critic algorithms. *Advances in neural information processing systems, 12*.

Lewis, F. L., & Vrabie, D. (2009). Reinforcement learning and adaptive dynamic programming for feedback control. *IEEE circuits and systems magazine, 9(3)*, 32-50.

Li, S. E. (2023). Approximate dynamic programming. *Reinforcement Learning for Sequential Decision and Optimal Control,* 231–295. https://doi.org/10.1007/978-981-19-7784-8_8

Mombaur, K., Laumond, J.-P., & Truong, A. (2011). An inverse optimal control approach to human motion modeling. *Springer Tracts in Advanced Robotics*, 451–468. https://doi.org/10.1007/978-3-642-19457-3_27

Nasteski, V. (2017). An overview of the supervised machine learning methods. *HORIZONS.B, 4*, 51–62. https://doi.org/10.20544/horizons.b.04.1.17.p05

Ng, A. Y., & Russell, S. (2000, June). Algorithms for inverse reinforcement learning. In *Icml* (Vol. 1, No. 2, p. 2).

Powell, W. B. (2011). *Approximate dynamic programming: Solving the curses of dimensionality*. John Wiley and Sons.

Russell, S. (1998). Learning agents for uncertain environments (extended abstract). *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*. https://doi.org/10.1145/279943.279964

Shah, S. I., & De Pietro, G. (2021). An overview of inverse reinforcement learning techniques. *Intelligent Environments 2021.* https://doi.org/10.3233/aise210097

Shkurti, F., Kakodkar, N., & Dudek, G. (2018). Model-based probabilistic pursuit via inverse reinforcement learning. *2018 IEEE International Conference on Robotics and Automation (ICRA)*. https://doi.org/10.1109/icra.2018.8463196

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (2nd ed.). MIT Press.

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine Learning, 8(3–4)*, 279–292. https://doi.org/10.1007/bf00992698

Werbos, P. (1992). Approximate dynamic programming for real-time control and neural modeling. *Handbook of intelligent control.*

Wulfmeier, M., Ondruska, P., & Posner, I. (2015). Maximum entropy deep inverse reinforcement learning. *arXiv preprint arXiv*:1507.04888.

Ziebart, B. D., Maas, A., Bagnell, J. A., & Dey, A. K. (2008). Maximum entropy inverse reinforcement learning. In *Proceedings of the 23rd National Conference on Artificial Intelligence, 3*, 1433-1438.

APPENDIX

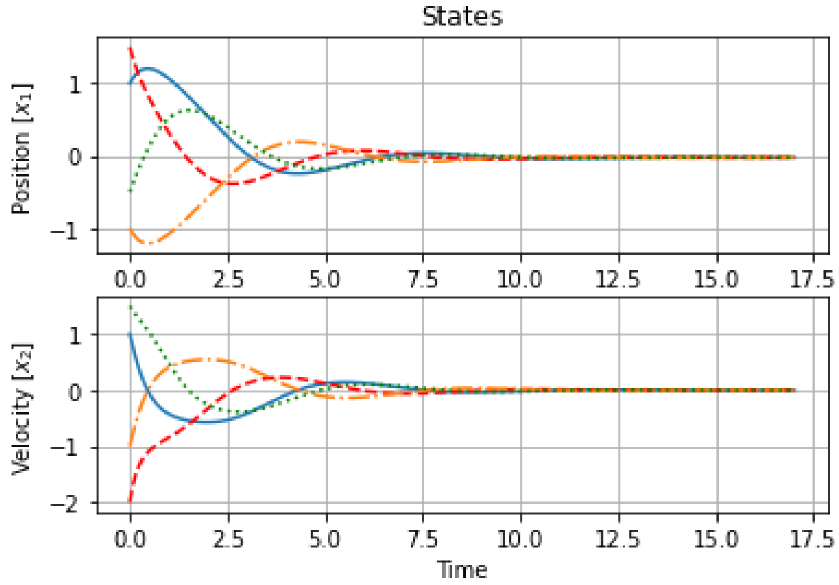


Figure 26. VDP states with differing initial conditions.

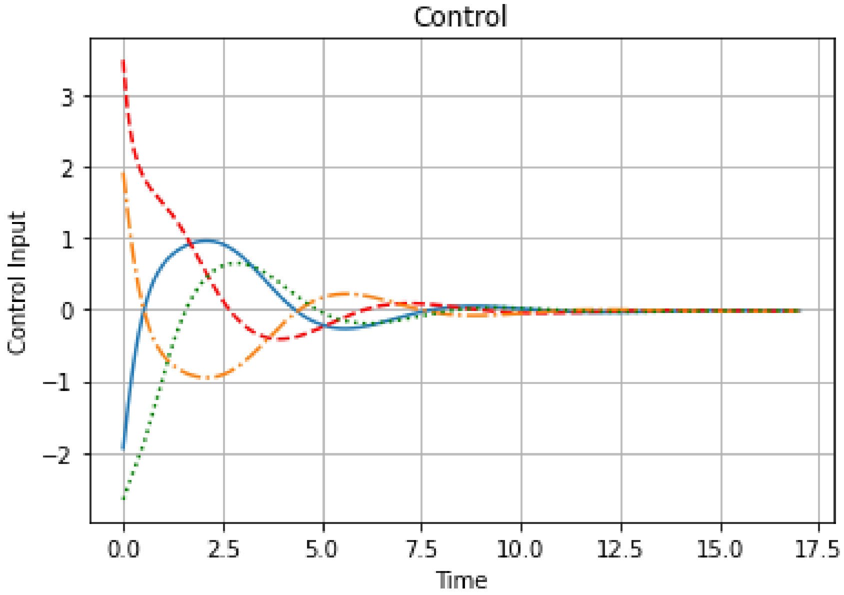


Figure 27. VDP control with differing initial conditions.

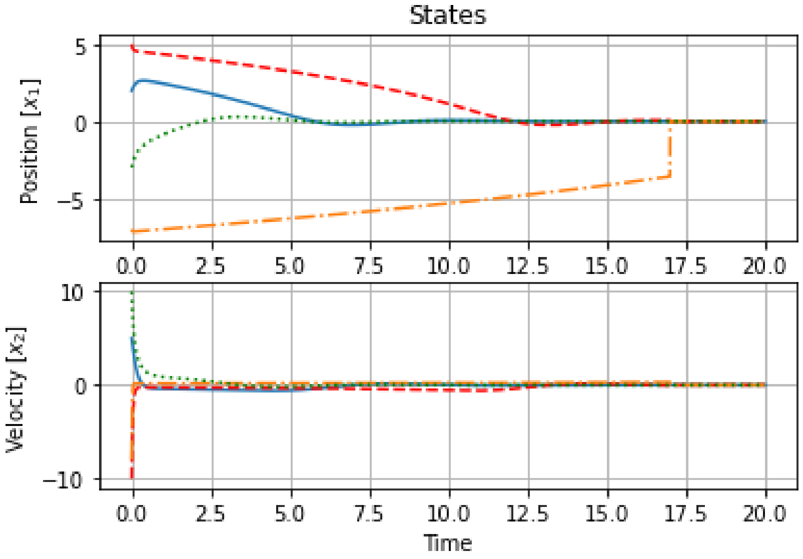


Figure 28.VDP states with differing initial conditions outside domain of training.

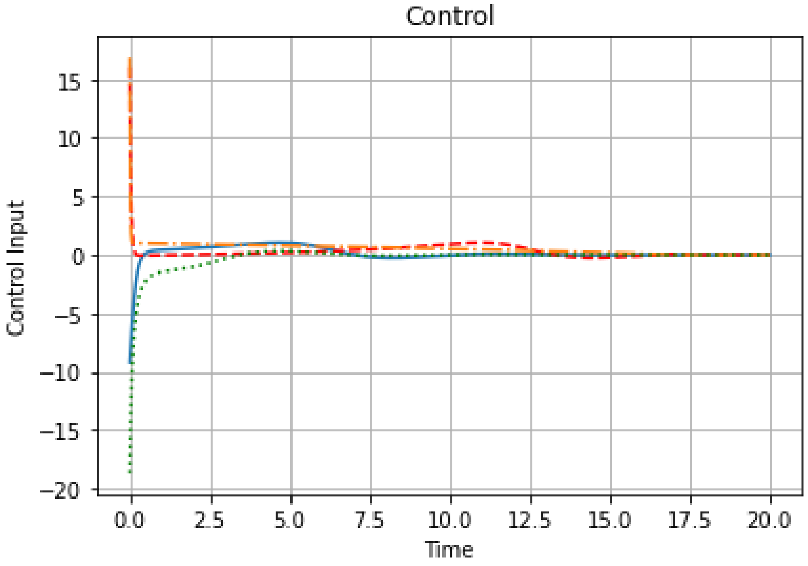


Figure 29.VDP states with differing initial conditions outside domain of training.

BIOGRAPHICAL SKETCH

Rodrigo Javier Gonzalez Salinas graduated from Sharyland Highschool in 2018. He pursued higher education as an undergraduate student at the University of Texas Rio Grande Valley, where he graduated Summa Cum Laude on May 2021 with a bachelor's degree in Mechanical Engineering. Rodrigo started his graduate education soon after and was awarded the Presidential Research Fellowship from UTRGV in the Fall of 2021, and the Dwight David Eisenhower Transportation Fellowship Program in the Spring of 2022. Rodrigo was the Vice-President and the Welding Lead for RGV Baja Racing at UTRGV team from 2020 to 2023. He completed his Master of Science in Mechanical Engineering degree in May 2024. Rodrigo Gonzalez can be reached at rodrigo-glz@live.com.