Minimalistic Adaptive Dynamic-Programming Agents for Memory-Driven Exploration

Mahrad Pisheh Var

A thesis submitted for the degree of **PhD in Computer Science**

School/Department: **Computer Science**

University of Essex

Date of submission for examination: November 2023

Keywords

- Adaptive Dynamic Programming
- Backpropagation Through Time
- Continuous Spaces
- Discrete Spaces
- Environment Exploration
- Food-Seeking Behavior
- Gated Recurrent Unit
- Long Short-Term Memory
- Maze Navigation
- Memory-Augmented Agents
- Optimal Policies
- Partially Observable Environments
- Reinforcement Learning
- Simulated Organism
- Spatial Intelligence

Abstract

Adaptive Dynamic Programming (ADP) and Reinforcement Learning (RL) are pivotal frameworks in machine learning, each presenting unique benefits and hurdles. This thesis examines the performance and adaptability of ADP agents using Backpropagation Through Time (BPTT) in continuous spaces. A Memory-Based Backpropagation Through Time (MBPTT) is reviewed, enhancing the conventional BPTT approach by integrating memory mechanisms to refine decision-making in partially observable environments.

Drawing upon foundational and recent developments in RL and ADP, this study explores the capability of BPTT agents across various environmental settings. It critically assesses different algorithms and memory models, including Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU), in simulating a "functionally sentient" organism seeking food.

The research makes two main contributions. Firstly, it empirically shows that even the simplest forms of memory-augmented agents can effectively navigate through a maze, performing better than existing techniques. This highlights the practical use of memory-based algorithms in spatial tasks. Secondly, the study investigates the performance of Backpropagation Through Time (BPTT) in bicycle navigation. It introduces a simulated organism that successfully combines BPTT with memory functions, demonstrating efficiency in environmental mapping and food search tasks. This work provides a solid foundation for future research in integrated learning systems.

In conclusion, this thesis reconciles the theoretical distinctions between memory and adaptive dynamic programming. Combining theoretical understanding with practical applications contributes to the ongoing effort to create more resilient, efficient, and adaptive agents in the rapidly advancing field of machine learning.

Contents

List of Tables

List of Figures

Statement of Original Authorship

The work in this thesis has not been previously submitted to meet the requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Acknowledgements

I want to express my deepest gratitude to my supervisors, Michael Fairbank and Spyridon Samothrakis, whose guidance, support, and expertise have been invaluable throughout this research. Your dedication to your students and your fields is genuinely inspiring.

Special thanks go to my uncle, Hakan Basagaoglu, for his valuable time and insightful feedback. His suggestions have played a significant role in shaping this work.

My appreciation extends to my colleagues and friends in the Computer Science Department (CSEE) for their camaraderie, constructive discussions, and all the shared moments that have made this journey enjoyable and educational.

Last but not least, I owe a debt of gratitude to my family. Thank you for your unwavering support, encouragement, and love, without which this journey would have been much more difficult.

Chapter 1

Introduction

Reinforcement Learning (RL) and Adaptive Dynamic Programming (ADP) have garnered significant attention for their potential in solving complex decision-making problems. These paradigms excel in environments where agents must make decisions under uncertainty, learning optimal policies through interaction with the environment. This thesis aims to explore how incorporating memory [1] mechanisms into these paradigms can enhance an agent's ability to learn and adapt, especially in dynamic environments.

1.1 Motivation and Research Questions

The dynamic and uncertain nature of environments in reinforcement learning (RL) poses significant challenges for agents to learn optimal policies through interaction [Shah, 2020]. Traditional RL and Adaptive Dynamic Programming (ADP) methodologies often fall short in complex scenarios where agents must adapt to changes and recall past experiences to make informed decisions [Chapman et al., 2023]. Recognising these limitations, this thesis reintroduces Backpropagation Through Time (BPTT), a powerful mechanism traditionally used in training Recurrent Neural Networks (RNNs) for supervised learning tasks into reinforcement learning. The motivation behind this strategic reintroduction is twofold:

1. To leverage BPTT's ability to capture temporal dependencies and apply it to the challenges of RL, thereby enhancing an agent's capacity to learn from sequences of events and actions

[1] *Memory* in the context of this thesis refers to the intrinsic memory mechanism of Recurrent Neural Networks (RNNs), which enables an agent to store and recall past experiences through its hidden state dynamics. This capability is crucial for enhancing the adaptability and efficiency of agents in dynamic or partially observable environments by allowing them to maintain a temporal sequence of events within their processing framework.

over time.

2. To further augment agents' performance by incorporating customised memory mechanisms into the BPTT framework, where in this thesis, this augmentation is referred to as Memory-Based Back Propagation Through Time (MBPTT), enabling agents to store and use past experiences more effectively. This integration addresses the inherent limitations of traditional RL approaches in dealing with dynamic environments.

In this thesis, the MBPTT algorithm builds on existing BPTT frameworks, incorporating memory functions similar to those found in Jordan and Elman's original RNNs [Jordan, 1997]. MBPTT refines these concepts to address the unique demands of control problem scenarios. By tweaking the memory management mechanisms within RNNs, MBPTT enhances the handling of sequential decision-making processes crucial in control tasks, where timing and accuracy of feedback are essential. This advancement enables more precise and efficient management of temporal dependencies and state transitions, addressing the challenges posed by dynamic systems.

This thesis is driven by the hypothesis that integrating advanced memory mechanisms with BPTT in reinforcement learning can significantly improve agents' adaptability and decision-making capabilities. It seeks to explore the following research questions:

1. How does the reintroduction of BPTT into RL, coupled with the incorporation of memory mechanisms, affect agents' learning efficiency and adaptability in complex environments?

2. Can the proposed memory-augmented BPTT framework outperform traditional RL algorithms in tasks that require strategic planning and decision-making over extended periods?

1.2 Objectives

This research is predicated on the innovative reintroduction of Backpropagation Through Time (BPTT) into reinforcement learning (RL), aiming to explore and demonstrate the use of memory mechanisms within this context. The objectives are structured to highlight the development and evaluation of a Memory-Based Backpropagation Through Time (MBPTT) algorithm and to critically analyse the implementation and integration of BPTT in RL environments. Specifically, the research seeks to:

1. Reintroduce and implement the BPTT algorithm within the RL framework, providing a methodological foundation for integrating memory mechanisms. This includes a detailed demonstration of how BPTT can be adapted and applied to enhance RL agents' learning process and decision-making capabilities.

2. Develop and critically evaluate the Memory-Based Backpropagation Through Time (MBPTT) algorithm. The focus will be on assessing the algorithm's effectiveness in leveraging past experiences to improve agent adaptability and performance in dynamic environmental settings.

3. Conduct a comparative analysis of agents enhanced with the MBPTT algorithm against traditional RL agents. This analysis will quantitatively measure improvements in adaptability, efficiency, and strategic planning capabilities across diverse environments, thereby showcasing the benefits of memory integration.

4. Analyse the architectural and computational implications of incorporating BPTT and memory mechanisms into RL algorithms. This objective aims to identify and discuss the challenges, limitations and impacts on performance and complexity, offering insights into the practical applicability of the MBPTT algorithm.

Through these objectives, the research aims to advance the understanding of BPTT's role and potential in RL, illustrating how its integration with memory mechanisms can significantly enhance the capabilities of BPTT agents. The findings are expected to contribute to the broader field of artificial intelligence by providing a robust framework for future agent learning and adaptability innovations.

1.3 Novel Contributions

This thesis makes several novel contributions to the field of reinforcement learning and adaptive dynamic programming:

1. Introduction to implementing the Memory-Based Backpropagation Through Time (MBPTT) algorithm, a new approach that leverages memory for enhanced decision-making in RL agents.

2. Comprehensive evaluation of memory-augmented agents in a dynamic environment, providing insights into their adaptability and performance relative to traditional approaches.

3. Theoretical and practical implications for the design of more efficient and adaptive learning agents, bridging the gap between memory mechanisms and adaptive dynamic programming.

1.4 Thesis Structure

The structure of this thesis is organised as follows to provide a clear and comprehensive exploration of the research objectives:

1. Chapter 2 (Literature Review): Reviews seminal works and contemporary advancements in RL and ADP, setting the stage for the thesis.

2. Chapter 3 (Memory-Augmented Q-Learning): Introduces a novel algorithm for maze navigation, demonstrating the benefits of memory in discrete spaces.

3. Chapter 4 (Exploring Continuous Spaces with ADP): Explores the application of MBPTT in continuous environments, highlighting the challenges and solutions for integrating memory.

4. Chapter 5 (Simulating Adaptive Behaviours): Presents the application of memory-augmented learning in simulating food-seeking behaviour in agents, showcasing the practical benefits of the approach.

5. Chapter 6 (Conclusion and Future Work): Summarises the findings and contributions of the thesis and outlines directions for future research.

1.5 Contributed Papers

This thesis has led to the development and publication of two significant papers:

- *Finding Eulerian Tours in Mazes Using a Memory-Augmented Fixed Policy Function*: Addresses memory application in solving complex mazes.

- *A Minimal "Functionally Sentient" Organism Trained with Backpropagation Through Time*: Explores the effectiveness of MBPTT in simulating adaptive behaviours in agents.

Chapter 2

Background & Literature Review

The domain of Reinforcement Learning (RL) [Kaelbling et al., 1996] manifests as a compelling synthesis of decision-making, optimisation, and learning. Adaptive Dynamic Programming (ADP) further accentuates this landscape, originating as a confluence of dynamic programming and supervised learning and eventually evolving to play an instrumental role in modern RL frameworks. RL and ADP share the foundational goal of optimising decision-making in uncertain environments, but each brings unique methodologies and perspectives. This chapter delves deep into the principles, advancements, and intricacies that define RL, illuminating ADP's intertwined relationship and contributions.

The discussion commences by explaining the foundational principles of RL. It covers essential constructs such as agents and environments and culminates in the quintessential objective of all RL paradigms: the maximisation of long-term rewards. This foundation paves the way for insights into adaptive dynamic programming, detailing its origins and intricate relationship with RL.

As the narrative unfolds, it underscores the essential considerations inherent to RL research. Discussions delve into the contrasts between model-free and model-based approaches and illuminate the nuances of partially observable environments. Furthermore, the importance of standardisation and differentiability in modern RL paradigms is explained in detail.

Delving deeper, the chapter represents the realm of Deep Reinforcement Learning (DRL). It outlines the myriad challenges associated with DRL, traces its evolutionary journey, and emphasises its intersections with high-dimensional sensory environments. Notable attention is given to issues of convergence, the subtleties of multi-task learning, and the domain of multi-agent systems.

Subsequent sections carefully explain the algorithms and techniques that underpin RL. From the classic Q-learning to the pioneering Proximal Policy Optimisation, the chapter aspires to provide readers with an encompassing view of RL's algorithmic landscape, interspersed with discussions on backpropagation and actor-critic methodologies.

Turning to the delicate balance of memory and adaptability in RL, the narrative explores the mechanisms by which modern agents employ memory architectures, from LSTM to GRUs. The paramount role of meta-learning in enhancing adaptability is highlighted.

To conclude this chapter, it introduces the platforms and engines essential for RL research and applications. It discusses the details of DeepMind Lab, explores the concept of continuity in physical tasks, and presents modern simulations, such as Brax, which uses the features of JAX for enhanced performance.

2.1 Introduction to Reinforcement Learning

Reinforcement Learning (RL) can be envisioned as a bridge between classical decision processes and the computational means to solve them. Instead of relying on explicit instruction, an RL agent learns by interacting with its environment, constantly adjusting its strategies based on feedback, typically in rewards or penalties. This feedback-driven loop makes RL particularly well-suited for many applications, from game-playing agents to robotic controls.

The concept of memory is central to enhancing the adaptability and efficiency of RL agents, especially in dynamic environments. This thesis emphasises the memory mechanisms inherent in Recurrent Neural Networks (RNNs), exploring how these can be leveraged within RL frameworks to improve decision-making processes. By integrating RNN-like memory capabilities, RL agents can maintain a temporal sequence of events that is pivotal for tasks requiring recalling past experiences to inform future actions.

The following subsections will explore and introduce the key reinforcement learning algorithms considered in this thesis. Each algorithm has been selected for its unique contribution to the field and its relevance to the comparison. In this exploration, particular attention will be paid to how these algorithms can incorporate or benefit from memory mechanisms, aligning with the thesis's focus on enhancing RL through the principles of RNN memory. This section delves into each algorithm's foundational principles, practical applications, and distinct roles in comparative analysis, with a keen eye on their interfacing with memory to solve complex decision-making problems. This exploration aims to provide a comprehensive understanding of the current state

of reinforcement learning, its potential for solving complex decision-making problems, and the pivotal role of memory in advancing these capabilities.

2.1.1 Basics of Reinforcement Learning

At the heart of RL lies a trial-and-error mechanism, where an agent aims to discover an optimal policy that maximises its cumulative reward over time. Unlike supervised learning, where clear input-output pairs dictate the learning process, RL often requires agents to explore uncertain territories, balancing the need to exploit known strategies with the drive to explore new ones. The agent's environment often dictates this delicate balance, current knowledge state, and the broader objectives it seeks to achieve.

Definitions: Agents, Environments, Actions a_t, and Rewards r_t

It is essential to start with its foundational elements to understand the intricacies of Reinforcement Learning (RL). At the heart of any RL process are agents that make decisions, environments wherein these decisions are made, actions a_t which represent the choices made by agents at each time step t, and rewards r_t which are feedback mechanisms indicating the success or quality of those decisions.

- r_t: This represents the immediate reward received after taking an action at time t. It reflects the immediate benefit or cost associated with the action taken in the current state. Mathematically, it can be expressed as:

  $$r_t = r(s_t, a_t)$$

  where s_t is the current state, a_t is the action taken, and s_{t+1} is the next state.

- R_t: This denotes the total accumulated reward from time t onwards, also referred to as the return. It encompasses not just the immediate reward but all subsequent rewards, potentially discounted by a factor γ (gamma) to represent the decreasing value of future rewards. The return can be formulated as:

  $$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

  where γ is the discount factor, with a value in the range $[0, 1]$.

These fundamental concepts form the backbone of RL, shaping its algorithms, methodologies, and outcomes.

The Objective: Maximising Long-term Total Rewards

While the immediate rewards received by an agent after each action play a crucial role in guiding its behaviour, the true essence of RL lies in its long-term vision. An agent's primary objective is not just to maximise the immediate reward but to ensure maximising the cumulative sum of rewards over many steps in the future. This foresight often requires the agent to make sacrifices in the short term to reap more substantial benefits in the long run. For instance, in a chess game, sacrificing a pawn might seem like a loss initially, but if it opens a pathway to checkmate the opponent's king in subsequent moves, the sacrifice is justified. This principle of delaying immediate gratification for long-term success is at the heart of many RL algorithms and strategies.

2.1.2 Adaptive Dynamic Programming

Adaptive Dynamic Programming (ADP) and Reinforcement Learning (RL) share a common goal: to learn a policy function that maximises a long-term reward function. ADP, however, differentiates itself by not treating the environment as a black box, thereby enabling the use of efficient gradient calculations [Murray et al., 2002, Wang et al., 2009, Prokhorov and Wunsch, 1997a]. This distinction is crucial in environments where the physics model is known and differentiable, making methods like Backpropagation Through Time (BPTT) and Dual Heuristic Programming (DHP) significantly more efficient in learning than model-free RL models, sometimes by several orders of magnitude [Fairbank and Alonso, 2012].

The Implications and Relevance to This Thesis: The use of BPTT in ADP, particularly when enhanced with optimisation algorithms like the Levenberg-Marquardt (LM) method, showcases the potential for significant improvements in the training of Recurrent Neural Networks (RNNs) [Fu et al., 2014]. This thesis emphasises the critical role of memory mechanisms inherent in RNNs and their synergistic potential when combined with ADP approaches. The ability of BPTT to adapt to changing environment physics models underscores flexible adaptability, essential for dynamic environments discussed in later chapters [Fairbank et al., 2014a].

Furthermore, the origins of ADP and RL, tracing back to the pioneering work of Minsky et al. [1963], Samuel [1959], and the introduction of function approximators by Adami [2023], lay the groundwork for contemporary RL systems. The evolution from heuristic approaches to the development of HDP and DHP algorithms [Werbos, 1997, 2021, Prokhorov and Wunsch, 1997a] illuminates the trajectory of RL towards leveraging complex models and algorithms for

decision-making. *Linking Back to This Thesis:* The foundational principles of ADP, particularly the emphasis on understanding and leveraging the environment's dynamics, align closely with the objectives of this thesis. By exploring the intersection of ADP methodologies like BPTT with memory-driven RNN mechanisms, this research seeks to push the boundaries of what is achievable in RL, particularly in applications requiring nuanced decision-making and adaptability. The subsequent chapters will delve deeper into how these ADP methodologies are integrated within the proposed RL frameworks, their implications for learning efficiency, and their role in enhancing the adaptability of agents in complex environments.

Origins of Adaptive Dynamic Programming and Reinforcement Learning

The historical context provided by the works of Minsky et al. [1963], Samuel [1959], and Adami [2023] not only highlights the evolution of RL but also sets the stage for the advanced methodologies discussed in this thesis. The transition from early checker-playing algorithms to sophisticated ADP algorithms like HDP and DHP reflects a broader shift towards more adaptive, efficient, and complex RL systems. This evolution mirrors the trajectory of this thesis, from understanding the foundational aspects of RL to applying advanced ADP techniques in novel ways to enhance agent performance. *Connecting to Later Chapters:* As this thesis progresses, the focus will shift towards applying the insights gleaned from the study of ADP and its algorithms to address specific challenges within the RL domain. This includes exploring the practical applications of these algorithms in real-world scenarios, their integration with RNNs for improved memory and decision-making capabilities, and the broader implications of these approaches for the field of artificial intelligence. By grounding our exploration in the rich history and proven methodologies of ADP, it aims to contribute meaningful advancements to the field of RL, particularly in developing agents capable of navigating the complexities of dynamic and uncertain environments.

2.1.3 Backpropagation Through Time

BPTT is a foundational algorithm for training RNNs. This approach is pivotal in dealing with sequential data by bridging the chasm between current predictions and past information, thus allowing the effective propagation of gradients through time. The origins of BPTT can be attributed to the groundbreaking work of Werbos [1990], wherein adaptations were proposed to address challenges intrinsic to sequential datasets. Since its inception, the emphasis has been on

leveraging BPTT in control scenarios, with notable contributions from Prokhorov and Wunsch [1997a], Lillicrap and Santoro [2019] augmenting this concept.

BPTT finds common usage in supervised learning tasks like time-series forecasting or natural language processing. However, the limitations and strengths of BPTT to apply to RL tasks remain relatively unknown. This possibility arises when a known and differentiable model of the training environment exists. [1]

BPTT is used to compute the derivative of the total reward R with respect to the neural network weights, $\vec{w}$. To do this, it views the environment model as an extra layer of a recurrent neural network, combined with the agent's original neural network ("Agent Brain"), as shown in Fig. 2.1.

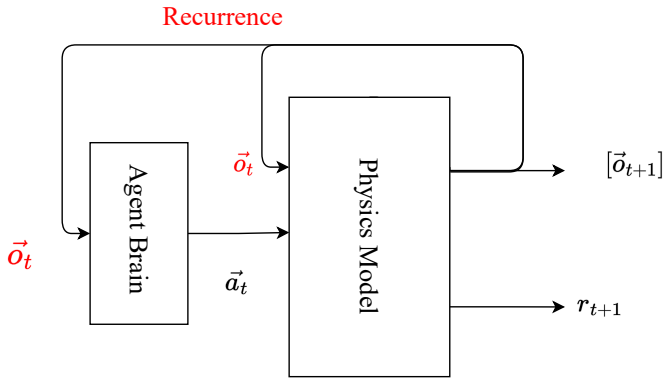


Figure 2.1: Recurrence between Agent Brain and Physics model allows the Agent's brain.

BPTT uses automatic differentiation to compute the required derivative $\frac{\partial R}{\partial \vec{w}}$. Internally, this unrolls the combined network of Fig. 2.1 "through time" to obtain the unrolled network shown in Fig. 2.2. Automatic differentiation is then used to "backpropagate" the derivatives of R with respect to $\vec{w}$ right through the unrolled network.

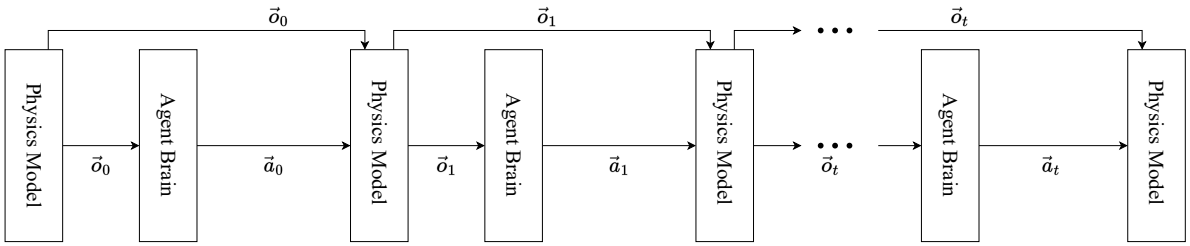


Figure 2.2: Unrolled combined network in BPTT.

[1] In the problems mentioned in the following chapters, it is stated that the requirement for BPTT to use a known environment model classifies it under ADP (Approximate Dynamic Programming) [Powell, 2007] rather than being categorised as "true" RL. The basis for this distinction is that pure RL is designed to handle environments without prior knowledge. In contrast, ADP depends on having access to an environment model, as explicitly defined in the Environment and Agent Definitions sections in Sections 4.2.3 and 5.1.

Once the quantity $\frac{\partial R}{\partial \vec{w}}$ is obtained, gradient ascent on R can then be performed to improve the performance of the neural network at maximising R:

$$\Delta \vec{w} = \alpha \frac{\partial R}{\partial \vec{w}}, \tag{2.1}$$

where $\alpha > 0$ is a small learning rate. This single simple process is model-based, i.e., it requires knowledge of the derivatives of the environment functions. If successful, this learning process will maximise R by (2.1).

RNNs, however, aren't without their challenges. They are notoriously known for the exploding gradient problem. This issue stems from an error gradient used to adjust network weights during training. Consequently, an exploding gradient can result in an unstable network, possibly converting weights into a "not a number" (NaN) value, thereby hindering further weight updates.

While LSTM networks are commonly proposed as a remedy, alternative solutions include network redesign, gradient clipping, or weight regularisation. Fairbank et al. [2014a] introduces a unique solution involving a tracking problem and stabilisation matrix. This methodology enabled the neural network to yield consistently reduced training errors, thus enhancing its adaptive capabilities.

The environment under scrutiny in Fairbank et al. [2014a]'s work, a power plant, demanded offline training due to its ever-changing conditions. This setting, being a renewable energy generator application, required an adept controller for interfacing between the DC and AC sides of electric power.

Notably, Fairbank et al. [2014a] highlights that Adaptive Critic Designs (ACDs) [Prokhorov and Wunsch, 1997b] have been primary approaches to such problems. Characterised by two neural networks, an action network and a critic network, the critic network guides the action network, permitting real-time online training. A critical challenge with critic learning is its convergence issues, as indicated by Sutton [1988]. Fairbank et al. [2014a] suggests an approach wherein only the action network is trained offline using BPTT. This innovation, they assert, allows for true gradient descent on the cost-to-go function, ensuring convergence.

Furthering this discussion, Fairbank et al. [2014a] posits that BPTT ensures a swift response and adaptation compared to ACDs. This assertion is underpinned by the observation that RNN weights don't necessitate updates for adaptation. Additionally, they integrated a stabilisation matrix, a predetermined neural weight matrix encapsulating basic control behaviour. Combined with BPTT, this approach adeptly focused on mastering advanced behavioural nuances.

The agent aims to minimise the cumulative cost function by delving into neural control, which embodies reinforcement learning objectives. Fairbank et al. [2014b] emphasises the importance of clipping, particularly for the agent's movement in the trajectory's final time step. Clipping, in this context, pertains to truncating this final step. This, Fairbank et al. [2014b] illustrates, enhances the performance of explicit derivatives of the environment's model function, calculating the learning gradient.

Comparative Analysis of Policy Gradients and BPTT in Reinforcement Learning

The landscape of Reinforcement Learning (RL) showcases a variety of approaches, among which policy gradient methods and Backpropagation Through Time (BPTT) are particularly noteworthy for their distinct yet complementary functionalities. The comparison between these two methodologies underscores fundamental differences in their operational mechanisms and application contexts within RL.

1. **Policy Gradients**:

   - *Direct Policy Optimisation:* Policy gradient approaches focus on directly optimising the policy by adjusting the policy parameters to maximise the expected return. This contrasts with value-based methods that aim to learn a value function.
   - *Gradient Estimation:* The core of policy gradient methods lies in estimating the gradient of the expected return with respect to policy parameters, formulated as:

     $$\nabla_\theta R(\theta) \approx \mathbb{E}[\nabla_\theta \log \pi_\theta(a_t|s_t) R_t], \tag{2.2}$$

     emphasising the stochastic nature of policy optimisation and the reliance on expected values over trajectories.
   - *Variance Challenges:* A notable characteristic of policy gradients is their inherent variance, stemming from the expectation over trajectories. This variance can introduce challenges in achieving stable and efficient learning.

2. **Backpropagation Through Time (BPTT)**:

   - *Training Recurrent Architectures:* Primarily used for training RNNs, BPTT is instrumental in RL scenarios involving recurrent architectures to handle partial observability, facilitating error backpropagation across sequences.

- *Analytic Gradient Computation:* BPTT distinguishes itself by providing an exact gradient of the loss with respect to the model's parameters for specific sequences, thus termed "analytic" for its precision in gradient determination.

The above comparison reveals that, despite their operational differences, policy gradients and BPTT share foundational similarities. Policy gradients allow for model-free learning, generating gradients based on expected outcomes without requiring an intricate model of the environment. In contrast, BPTT delivers precise gradients for sequence-based learning but necessitates unfolding sequences over time, which can be computationally intensive for long sequences.

The convergence of these methods is particularly evident in actor-critic architectures, where the actor's policy updates are guided by policy gradients, and the critic's value function, potentially employing an RNN, is trained using BPTT. This synergy illustrates the nuanced relationship between policy optimisation and sequence modelling in RL, highlighting the complementary roles of policy gradients and BPTT in advancing gradient-based learning approaches.

For further reading on the theoretical underpinnings and practical applications of these methodologies, referencing seminal works such as Sutton and Barto's "Reinforcement Learning: An Introduction" [Sutton and Barto, 2018] and the comprehensive analysis by Lillicrap et al. [Lillicrap et al., 2015] on continuous control with deep policy gradients can provide deeper insights into their comparative advantages and limitations.

Harnessing Analytic Policy Gradients for Efficient Robotic Control

The quest for balancing computational efficiency with tracking accuracy in robotic control has led to significant advancements in both traditional and learning-based control strategies. Model Predictive Control (MPC) has been the gold standard for achieving high precision in trajectory tracking but at the cost of computational resources [Rawlings, 2000, Wiedemann et al., 2022]. Conversely, Reinforcement Learning (RL) offers a promising avenue for efficient control with reduced computational demands, though it historically lagged behind MPC regarding tracking precision.

Introduction to Analytic Policy Gradient: The recent work by Wiedemann et al. [2022] presents a groundbreaking approach to this challenge through the Analytic Policy Gradient (APG) method. By leveraging differentiable simulators, APG enables efficient offline training of controllers, directly optimising the tracking error via gradient descent. This method challenges the conventional limitations attributed to RL in precision tasks. It demonstrates superior tracking

performance across various robotic applications, including but not limited to CartPole, quadrotors, and fixed-wing drones. APG's ability to rival the precision of MPC while dramatically reducing the computational overhead marks a pivotal shift towards more efficient robotic control methodologies.

Implications and Relationship to This Thesis: The advent of APG underscores a critical theme of this thesis: the exploration of advanced control strategies that merge the computational efficiency of RL with the precision traditionally reserved for model-based approaches like MPC. APG's success in employing gradient-based optimisation directly on tracking errors aligns with our investigation into how RL can be adapted for high-precision tasks in dynamic environments, particularly when augmented with memory mechanisms or analytic gradients.

Future Directions and Link to Later Chapters: While APG showcases exceptional potential in fixed-horizon, short-duration tracking tasks with known dynamics, its exploration in extended-duration tasks and real-world scenarios remains an open frontier. This presents an exciting opportunity for future research, particularly in how curriculum learning schemes can be integrated to mitigate training instabilities and broaden the applicability of APG to more complex robotic control problems. Subsequent chapters of this thesis will delve into these aspects, examining how APG and similar strategies can be further optimised and applied within the broader context of RL for robotic control. The discussion will include potential limitations, areas for improvement, and the exploration of real-world applications, aiming to bridge the gap between RL efficiency and the precision of traditional control strategies. In essence, APG represents a significant stride towards the objectives outlined in this thesis, offering a compelling case study in the convergence of RL efficiency and MPC-like precision. Its integration with differentiable simulators and potential for application in real-world scenarios provides a robust foundation for future explorations to enhance RL's adaptability and effectiveness in complex, dynamic control tasks.

2.1.4 Q-Learning: A Brief Overview

Q-learning, a cornerstone of off-policy reinforcement learning algorithms, seeks to establish the optimal action-selection policy within a finite Markov decision process. Introduced by Watkins in 1989 [Watkins and Dayan, 1992], its use spans robotics, game theory, and natural language

processing, among other domains. Off-policy learning[2] permits agents to learn the value of the optimal policy independently from their actions, thus enabling exploration of the environment under a more exploratory or random policy while learning about a different, potentially optimal policy. *Implications for Memory-Augmented Learning:* The framework of Q-learning, especially its off-policy characteristic, lays a foundational role in exploring memory-augmented learning strategies within this thesis. Chapter 3, which introduces a novel memory-augmented Q-learning algorithm for maze navigation, builds upon the principles of Q-learning to demonstrate the benefits of incorporating memory mechanisms in discrete spaces. The adaptability and efficiency of agents in dynamic or partially observable environments, as highlighted in this thesis, can be significantly enhanced by integrating memory capabilities, allowing for a nuanced balance between exploration and exploitation—a key challenge in reinforcement learning.

Connection to Thesis Objectives: The exploration of Q-learning serves as a basis for understanding traditional reinforcement learning strategies and as a springboard for advancing the discussion on how memory mechanisms can be woven into these frameworks to address complex decision-making problems. The subsequent chapters will delve into the practical applications of these concepts, showcasing how memory-augmented learning algorithms can outperform traditional methods in environments that require strategic planning and decision-making over extended periods.

By examining Q-learning through the lens of memory augmentation, this thesis aims to contribute to the broader field of artificial intelligence by offering innovative solutions for enhancing agents' adaptability and learning efficiency in complex scenarios. Introducing memory mechanisms into the Q-learning algorithm represents a key innovation in this research, setting the stage for a detailed exploration of their implementation and impact in later chapters.

One of the main advantages of Q-learning is its ability to learn optimal policies without requiring a model of the environment, making it well-suited for problems where the environment is only partially observable or is computationally expensive to model. This characteristic makes it particularly relevant to the study done in Chapter 3, as the computational resources are often a limiting factor in maze-solving algorithms.

[2]Off-policy learning is a reinforcement learning strategy where the policy being learned (the optimal policy) differs from the policy employed to perform actions (the exploration or behaviour policy). This distinction allows agents to explore the environment under a more exploratory or random policy while learning about a potentially optimal policy. It offers the advantages of separating exploration from exploitation, learning efficiently from diverse sources of experience, and employing techniques like importance sampling to reconcile differences between the behaviour and target policies. Off-policy methods like Q-learning are invaluable in scenarios where acquiring new data is costly or impractical, enabling learning from pre-existing datasets or the experiences of other agents.

The algorithm uses an action-value function denoted as $Q(s,a)$, which estimates the expected return when taking action a from state s.

In a typical reinforcement learning setup, an agent interacts with an environment over discrete time steps. At each time step t, the agent receives an observation o_t and selects an action a_t from the available action set based on its policy $\pi(a|s)$. The environment then transitions to a new state s_{t+1} and returns a reward r_t.

The value function, $V^{\pi}(s)$, represents the expected return starting from state s and following a given policy π:

$$V^{\pi}(s) = \mathbb{E}[R_{t:\infty}|s_t = s, \pi].$$

In contrast to the value function, the action-value function $Q^{\pi}(s,a)$ estimates the expected return if the agent takes action a from state s and then follows the policy π:

$$Q^{\pi}(s,a) = \mathbb{E}[R_{t:\infty}|s_t = s, a_t = a, \pi].$$

Q-learning employs temporal difference (TD) learning to estimate Q values without requiring prior knowledge of the environment. A table $Q[S,A]$ is maintained to store these Q-values, and it is updated using the Bellman equation:

$$Q^{\pi}(s_t,a_t) = \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | s_t, a_t].$$

The Q-values are updated through the following equation:

$$\Delta Q(s_t,a_t) = \alpha \left(r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t) \right).$$

where the $\alpha > 0$ is the learning rate.

Actions are selected according to a policy $\pi : \mathbb{S} \to \mathbb{A}$, where the objective is to find a policy that maximises the expected return. The greedy policy is often defined as follows:

$$a = \arg\max_{a'} Q(s,a').$$

Thus, an episode comprises a sequence of state transitions, starting from an initial state and terminating at a final state.

Deep Q-Network (DQN)

The advent of the Deep Q-Network (DQN) by Mnih et al. [2013, 2015] marked a pivotal advancement in reinforcement learning, introducing a robust framework for integrating deep neural

networks with Q-learning. By implementing strategies such as experience replay, target networks, reward clipping, and frame skipping, DQN mitigated several key challenges associated with deep learning in RL, notably the instability of the learning process. The concept of experience replay, initially proposed by Lin [1992], effectively addresses the issue of overfitting by storing and using past experiences, thus enhancing the data efficiency and stability of learning. Introducing a separate target network helps stabilise the volatile target functions often encountered in deep neural network training. *Implications for BPTT Comparison:* The innovations introduced by DQN, particularly in stabilising the learning process and improving data efficiency, provide a critical backdrop for evaluating the BPTT algorithm within the thesis. While DQN offers a groundbreaking approach to handling discrete action spaces, this research explores BPTT. It focuses on its application in environments with continuous state and action spaces and how memory mechanisms can be effectively integrated. The comparison with BPTT underscores the adaptability and efficiency of memory-augmented learning strategies, particularly in dynamic environments where the ability to recall and leverage past experiences significantly enhances decision-making processes.

Challenges with Convergence in Q-learning

The integration of Q-learning with non-linear function approximators, while extending the applicability of RL to a wider range of complex problems, introduces challenges related to convergence. As highlighted by Tsitsiklis and Van Roy [1997], the marriage of Q-learning with non-linear function approximators, such as those used in DQN, can lead to divergence, a stark contrast to the guaranteed convergence when employing linear function approximators. This divergence risk underscores the importance of carefully designed algorithms and architectures to ensure stable and effective learning.

BPTT's Role in Addressing Convergence Challenges: Within the context of this thesis, the discussion of convergence challenges in Q-learning sets the stage for a deeper exploration of how BPTT, with its emphasis on sequential data processing and memory integration, offers alternative solutions. The application of BPTT in continuous spaces, as discussed in later chapters, reveals its potential to provide stable learning in complex environments and enhance the learning process's robustness and data efficiency. By comparing BPTT's performance against traditional and deep learning-based RL algorithms, this research aims to illuminate the pathways through which memory-augmented learning can overcome some of the enduring challenges in

the field, offering insights into the development of more sophisticated, efficient, and adaptable RL systems.

Recent Advancements in Value Function Estimation

Hinton [2007]'s method proved helpful in estimating the value function as shown by Sallans and Hinton [2004]. From Sallans and Hinton [2004]'s work, it can be understood that the gradient temporal difference methods have been partially addressed.

Achieving Convergence: Modern Methods and Limitations

Methods suggested by Maei et al. [2009, 2010] proved that the convergence is achievable when there is a fixed policy with a nonlinear function approximator or the agent is set to control the policy with linear function approximation with the usage of a restricted variant of Q-learning. However, Mnih et al. [2013] claims that this has not extended the research to non-linear control.

2.1.5 Optimisation Challenges in Deep Reinforcement Learning

The field of reinforcement learning (RL) has seen significant evolution with the advent of deep learning techniques, giving rise to various approaches to solving complex decision-making tasks. Deep Q-learning and policy gradient methods have emerged as foundational strategies, each with unique challenges and solutions.

Deep Q-Learning: Introduced by Mnih et al. [2015], deep Q-learning extends the classical Q-learning algorithm by using deep neural networks to approximate the Q-value function. This approach has demonstrated remarkable success in learning optimal policies directly from high-dimensional sensory inputs, as evidenced in playing Atari games from pixel input. However, deep Q-learning faces challenges in overestimating Q-values and sensitivity to hyperparameters.

Policy Gradient Methods: Policy gradient methods, as detailed by Mnih et al. [2016], aim to directly optimise the policy function by estimating the gradient of the expected return. Unlike Q-learning, policy gradient methods do not rely on value function approximation, which allows for the direct optimisation of stochastic policies and has been advantageous in continuous action spaces.

Trust Region Policy Optimisation (TRPO): To address the inefficiencies of vanilla policy gradient methods, Schulman et al. [2015] proposed TRPO. This method enhances the stability of policy updates by enforcing a constraint on the size of policy updates, ensuring that each update

is within a trust region. TRPO has improved learning stability and convergence, particularly in environments with high-dimensional action spaces.

Proximal Policy Optimisation (PPO): Building on the concepts of TRPO, Schulman et al. [2017] introduced PPO, a more practical approach that simplifies the optimisation process. PPO restricts the policy update step by employing a clipped objective function, which prevents large updates and maintains the advantages of TRPO while being computationally less demanding. Introducing an entropy bonus in PPO encourages exploration by adding a term to the objective function that rewards policy entropy.

Despite these advancements, RL grapples with scalability, robustness, and data efficiency challenges. PPO, with its balance of performance and practicality, represents a significant step forward. It also underscores the need for ongoing research to address the complexities of training deep RL agents.

In conclusion, the evolution from deep Q-learning through TRPO to PPO illustrates the continuous pursuit of more efficient and stable optimisation methods in deep reinforcement learning. Each approach brings us closer to developing RL agents capable of operating in increasingly complex and dynamic environments, highlighting the importance of adaptability, stability, and efficiency in RL algorithm design.

2.1.6 REINFORCE Algorithm

The REINFORCE algorithm, introduced by Williams [1992], represents a foundational approach within the realm of reinforcement learning, specifically under the Monte Carlo policy gradients category. This method updates policy parameters based on Monte Carlo estimations, relying on full episode rollouts to compute gradients [Bahdanau et al., 2016]. While REINFORCE simplifies the policy gradient calculation by not requiring a model of the environment, its notable drawback is the high variance in updates. This variance stems from the algorithm's dependence on the trajectory to estimate the gradient, leading to potentially unstable learning dynamics similar to those observed in early Deep Q-Network (DQN) implementations [Mnih et al., 2013].

Addressing Variance and Instability: The instability challenge associated with REINFORCE is akin to those encountered in other reinforcement learning strategies, including DQN. In response to these challenges, Wu et al. [2018] proposed using action-dependent factorised baselines to mitigate the variance inherent in log probabilities and cumulative reward estimations. The algorithm can achieve more stable and efficient learning by incorporating a baseline - a

technique that borrows from the Actor-Critic framework [Konda and Tsitsiklis, 1999]. This strategy, alongside advancements in Advantage Actor-Critic (A2C) and Soft Actor-Critic (SAC) methods [Haarnoja et al., 2018], illustrates the evolving landscape of reinforcement learning towards reducing variance and enhancing stability in policy updates.

Relation to BPTT and Memory-Augmented Learning: The exploration of variance reduction techniques in REINFORCE and its implications for reinforcement learning provides a pertinent backdrop for this thesis's investigation of Backpropagation Through Time (BPTT). Specifically, the comparison with BPTT aims to examine how incorporating memory mechanisms addresses the challenges of high variance and instability and leverages the strengths of policy gradient methods for complex decision-making tasks. Integrating BPTT with reinforcement learning, particularly in memory-augmented learning contexts, offers a promising avenue for enhancing the adaptability and efficiency of learning agents in dynamic environments.

This section underscores the continuous pursuit of more robust and effective learning strategies by situating the discussion of the REINFORCE algorithm within the broader examination of reinforcement learning optimisation challenges. The subsequent chapters will delve deeper into how BPTT and memory mechanisms can further advance the field, building on the foundational principles exemplified by REINFORCE and addressing its limitations through innovative approaches.

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta\left(a_t \mid s_t\right)\left(R_t - b\left(s_t\right)\right)\right]. \tag{2.3}$$

where $J(\theta)$ is the objective function or performance measure aimed to optimise with respect to the policy parameters θ, the function $b(s_t)$ represents the state value function $V(s_t)$ estimates the expected return (or total future rewards) from the state s_t.

2.1.7 Comparative Analysis Framework

Following the above review of state-of-the-art reinforcement learning algorithms, this thesis endeavours to juxtapose the Backpropagation Through Time (BPTT) algorithm against these notable methods. The exploration into Actor-Critic methods, including A2C, SAC, DDPG, TD3, and PPO, highlights their pivotal contributions to tackling challenges related to learning stability, exploration efficacy, and the exploration-exploitation balance in environments with both discrete and continuous action spaces.

Implications for Comparative Analysis with BPTT

The successes of these methods across various applications, from gaming to autonomous navigation, underline the advancement of reinforcement learning strategies towards more efficient, robust and highly adaptable solutions. Each method's unique approach to optimising policy and value functions sets a relevant context for assessing the BPTT algorithm's performance and utility.

- **Actor-Critic Methods:** The dual-network structure of these methods provides valuable insights into the potential for BPTT to integrate value-based and policy-based learning, hinting at hybrid applications.
- **Advantage and Soft Actor-Critic:** These methods' focus on the exploration-exploitation balance sets a benchmark for evaluating BPTT's effectiveness in managing continuous action spaces.
- **Deep Deterministic Policy Gradient and Variants (DDPG, TD3):** The deterministic approach to policy optimisation and strategies to mitigate overestimation bias offer a backdrop for comparing BPTT's capability for achieving stable and precise policy optimisation in continuous environments.
- **Proximal Policy Optimisation:** PPO's simplicity and efficacy in iterative policy improvement provide a metric for comparing BPTT's data efficiency and adaptability to policy constraints.

Structuring the Comparative Analysis

This analysis aims to extensively investigate how BPTT, with its intrinsic memory mechanism, aligns with or diverges from these algorithms, especially in scenarios requiring a deep understanding of temporal dynamics. The following chapters will methodically examine BPTT's theoretical foundations, practical implementations, and performance across various tasks, directly contrasting these with the accomplishments and shortcomings of the Actor-Critic methods.

Through this comparative perspective, the thesis will illuminate BPTT's distinct contributions to reinforcement learning, potential synergies with existing methods, and areas where it augments or complements the capabilities of contemporary strategies. This exploration is poised to highlight each approach's relative strengths and weaknesses and forge pathways for future

innovations in reinforcement learning, leveraging the adaptability and efficiency of BPTT alongside the robust foundations laid by Actor-Critic methodologies.

Actor-Critic Methods

In Actor-Critic algorithms, the "Critic" estimates the value function while the "Actor" modifies the policy distribution based on the Critic's feedback. Actor and Critic functions are typically parameterised using neural networks, as represented in Eq. (2.4).

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta \left(a_t \mid s_t \right) Q_w \left(s_t, a_t \right) \right]. \tag{2.4}$$

Advantage Actor-Critic

The Advantage Actor-Critic (A2C) and its asynchronous counterpart, the Asynchronous Advantage Actor Critic (A3C), are two primary variants. Introduced by Mnih et al. [2016], A3C employs parallel training, where agents update a global value function in parallel environments. This asynchronous approach facilitates broader state-space exploration. In contrast, A2C uses a single worker for state space exploration.

Soft Actor-Critic (SAC)

In reinforcement learning, environments can possess discrete or continuous state and action spaces. The Soft Actor-Critic (SAC) algorithm emerges as a potent solution for scenarios characterised by a continuous action space. SAC modifies the traditional RL objective function to maximise the entropy of the policy rather than focusing solely on maximising the cumulative reward.

A high entropy value signifies unpredictability when a variable is treated as a real number, and each potential value of that variable has an equal likelihood of being chosen. The principle behind emphasising entropy in the policy is to stimulate exploration. This is analogous to the strategy in Deep Q-learning, where equal probabilities are assigned to actions having identical or nearly equivalent Q-values, thus fostering exploration. The challenge lies in balancing exploration (searching for new strategies) and exploitation (leveraging known strategies). SAC addresses this by refraining from assigning overly high probabilities to actions.

SAC incorporates three distinct neural networks:

- State value function V, parameterised by ψ.

- Soft Q-function Q, parameterised by θ.
- Policy function π, parameterised by Θ.

The training objectives for these networks are as follows:

State Value Function Training: SAC aims to minimise the error in the state value function as depicted below:

$$J_V(\psi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}} \left[\frac{1}{2} \left(V_\psi(\mathbf{s}_t) - \mathbb{E}_{\mathbf{a}_t \sim \pi_\phi} \left[Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - \log \pi_\phi(\mathbf{a}_t \mid \mathbf{s}_t) \right] \right)^2 \right]. \tag{2.5}$$

where $\mathcal{D}$ is the replay buffer as a dataset that stores past experiences of the agent.

The gradient of the above equation guides the update to the parameters of the V function.

Q-network Training: The Q-network's training focuses on minimising the error as described in Eq. (2.6).

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \mathcal{D}} \left[\frac{1}{2} \left(Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - \hat{Q}(\mathbf{s}_t, \mathbf{a}_t) \right)^2 \right] \tag{2.6}$$

The Q-function parameters Q update rule is provided by Eq. (2.7).

$$\hat{\nabla}_\theta J_Q(\theta) = \nabla_\theta Q_\theta(\mathbf{a}_t, \mathbf{s}_t) \left(Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - r(\mathbf{s}_t, \mathbf{a}_t) - \gamma V_{\bar{\psi}}(\mathbf{s}_{t+1}) \right). \tag{2.7}$$

where $\bar{\psi}$ indicates the target network.

Policy Network Training: The training of the policy network π is focused on minimising the divergence as defined in Eq. (2.8).

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}} \left[\mathrm{D}_{\mathrm{KL}} \left(\pi_\phi(\cdot \mid \mathbf{s}_t) \| \frac{\exp(Q_\theta(\mathbf{s}_t, \cdot))}{Z_\theta(\mathbf{s}_t)} \right) \right]. \tag{2.8}$$

where:

- $J_\pi(\phi)$ is the objective function for the policy, with phi representing the policy parameters.
- $\mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}}$ denotes the expected value over the states s_t, which are sampled from the dataset D.
- D_{KL} is the Kullback-Leibler divergence, which is a measure of how one probability distribution is different from a reference probability distribution.
- $\pi_\phi(\cdot \mid \mathbf{s}_t)$ represents the policy, parameterised by ϕ, conditioned on the state s_t.
- $\frac{\exp(Q_\theta(\mathbf{s}_t, \cdot))}{Z_\theta(\mathbf{s}_t)}$ is the softmax policy distribution. The action-value function Q, parameterised by θ, is exponentiated and normalised by the partition function Z_θ to form a valid probability distribution over actions.

To facilitate backpropagation and to update the policy network parameters, Haarnoja et al. [2018] employed the reparameterisation trick. This approach ensures that policy sampling remains a differentiable process. The parameterised policy is defined in Eq. (2.9).

$$\mathbf{a}_t = f_\phi\left(\varepsilon_t; \mathbf{s}_t\right). \tag{2.9}$$

Deep Deterministic Policy Gradient (DDPG)

Lillicrap et al. [2015] represents the deep deterministic policy gradients, also known as DDPG. Lillicrap et al. [2015] explains that DDPG uses four neural networks:

- A Q-network θ^Q.
- A Deterministic policy function θ^μ.
- A target Q-network $\theta^{Q'}$.
- A target policy network $\theta^{\mu'}$.

Both policy network and Q-network are similar to the A2C network; however, in DDPG, states are directly mapped to actions by the Actor, whereas in A2C, the probability distribution is used across a discrete action space [Lillicrap et al., 2015]. Like DQN, the original network (Q-network) gets copied by the target network with a time delay, improving learning stability. Methods without the target network are prone to divergence; the update functions are interdependent with the network's calculated values.

The DDPG process can be broken down into:

- Experience replay.
- Actor and Critic network updates.
- Target network updates.
- Exploration.

It starts with initialising all networks, including the actor, critic, and target networks. A replay buffer is used to sample the experience; the sampled experience is then used to update the network parameters. For each trajectory roll-out, the DDPG's target policy network and target

value network calculate the next state Q-value demonstrated in Eq. (2.10), which is different from the Bellman equation shown in Eq. (2.11).

$$\text{Loss} = \frac{1}{N}\sum_{i}\left(y_i - Q\left(s_i, a_i \mid \theta^{Q}\right)\right)^2, \tag{2.10}$$

$$y_i = r_i + \gamma Q'\left(s_{i+1}, \mu'\left(s_{i+1} \mid \theta^{\mu'}\right) \mid \theta^{Q'}\right). \tag{2.11}$$

The calculated next-state Q-values are then used to minimise the error; the error is the mean squared loss shown in Eq. (2.10). The policy function is set to maximise the expected return, the off-policy or on-policy type of the algorithm directly affects how the policy is updated, and the mean of the sum of gradients used for the off-policy algorithm is shown below:

$$\nabla_{\theta^{\mu}} J(\theta) \approx \frac{1}{N}\sum_{i}\left[\nabla_a Q\left(s, a \mid \theta^{Q}\right)\Big|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^{\mu}}\mu\left(s \mid \theta^{\mu}\right)\Big|_{s=s_i}\right]. \tag{2.12}$$

Lastly, the target network update uses a soft copy or update to track the learned networks shown in Eq. (2.13) [Lillicrap et al., 2015].

$$\begin{aligned} \theta^{Q'} &\leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}, \\ \theta^{\mu'} &\leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}, \\ &\text{where} \quad \tau \ll 1 \end{aligned} \tag{2.13}$$

Twin Delayed DDPG (TD3)

TD3 is a variation of DDPG that is off-policy and can only be used on continuous action space. DDPG's main problem is an overestimation, where the Q-function, after a number of iterations, starts to overestimate the Q-values. Therefore, TD3 is used to tackle this problem of policy breaking [Fujimoto et al., 2018]. TD3 itself has three distinctive features which are:

- Clipped Double-Q learning, which uses two Q-functions.
- "Delayed policy update" in which policy and the target network are updated with a delay.
- Target policy smoothing by adding noise to the target action, avoiding exploitation.

Proximal Policy Optimisation (PPO)

Proximal Policy Optimisation (PPO) [Schulman et al., 2017] is a policy gradient method with recent breakthroughs in using DNN for control. Commonly, it is known to be hard to get good

results using policy gradients; the space size and the step size are the sensitive variables that need to be considered before using the policy gradients.

Solutions such as Trust Region Policy Optimisation (TRPO) [Schulman et al., 2015] and Actor-Critic with Experience Replay (ACER) [Wang et al., 2016] are far more complicated than PPO, and the complication originates from their limited compatibility with auxiliary losses or algorithms that share parameters between the value and policy function. However, PPO is easy to implement, and sample complexity is possible. One of the main features of the PPO is that it ensures the deviation from the previous policy is small while it minimises the cost function [Schulman et al., 2017].

Schulman et al. [2017] explains the motivation behind PPO was to implement an algorithm to have data efficiency and reliable performance as TRPO [Schulman et al., 2015]. The TRPO uses the maximisation of the surrogate objective shown in Eq. (2.14):

$$L^{\mathrm{CPI}}(\theta) = \max_{\theta} \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right], \tag{2.14}$$

where:

- $L^{\mathrm{CPI}}(\theta)$: The Conservative Policy Iteration objective function as a function of policy parameters θ.
- $\hat{\mathbb{E}}_t$: An empirical expectation over the time steps, indicating an average over a finite batch of data.
- $\pi_\theta(a_t|s_t)$: The probability of taking action a_t in state s_t under the policy parameterised by θ.
- $\pi_{\theta_{\mathrm{old}}}(a_t|s_t)$: The probability of taking action a_t in state s_t under the old policy, before the update, parameterised by θ_{old}.
- $\hat{A}_t$: An estimate of the advantage function at time t, representing the expected improvement of taking action a_t in state s_t over the average action.
- $r_t(\theta)$: The probability ratio $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\mathrm{old}}}(a_t|s_t)}$, reflecting how the policy's probability of taking action a_t in state s_t has changed after the update.

In Eq. (2.14), $r_t(\theta)$ is the probability ratio. However, PPO aims to penalise policy changes that move away from the probability ratio [Schulman et al., 2017]. Therefore, the $L^{\mathrm{CPI}}(\theta)$ causes

large policy updates in conservative policy iteration where the $J^{\text{CLIP}}(\theta)$ shown in Eq. (2.15) modifies the surrogate objective where it removes the incentive for moving r_t outside of the motivation objective [Schulman et al., 2017].

$$J^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\varepsilon, 1+\varepsilon\right)\hat{A}_t \right) \right]. \tag{2.15}$$

where ε is a small positive constant used in the clipping function to define the range $[1-\varepsilon, 1+\varepsilon]$.

2.2 Environment Considerations for Reinforcement Learning Research

The arena of reinforcement learning research is vast, often interplaying with various computational engines to train and evaluate agents. The choice of an engine, be it a game platform, a toolkit, or a specialised environment, plays a pivotal role in the overall efficacy of the research. Engines provide the foundation for agent interactions and shape the quality of feedback, the backbone of reinforcement learning. This section delves deep into the intricacies of potential engines for the project, weighing their strengths, weaknesses, and suitability for specific tasks.

RL algorithms can be broadly classified into two categories: *model-free* and *model-based* algorithms.

Model-Free vs. Model-Based Algorithms

Model-free algorithms are defined as those that do not rely on explicit knowledge of the environment functions which govern state transitions and deliver rewards. Even if such algorithms may contain instances of these environment functions, their design primarily facilitates the representation of an agent exploring its environment. Based solely on observations, the agent can infer received costs or state transitions without the need for direct knowledge of the underlying functions [Pong et al., 2018].

In contrast, a model-based algorithm necessitates explicit knowledge of the model and cost functions. Some model-based algorithms explicitly use the derivatives of the model and cost functions, evident, and the derivatives are not deducible from singular observations by the agent [Sun et al., 2019].

This distinction defines what differentiates a model-free algorithm from a model-based one—the ensuing subsections detail how both algorithms are used within ADPRL and the researchers that employ them.

Model-Based RL

Model-based RL, as described by Kaelbling et al. [1996], is distinct from the basic classification of an algorithm being model-based. It is defined as "Learn a model, and use it to derive a controller", sometimes referred to as "planning in RL" or "indirect RL" [Sutton et al., 1998, Chapter 9].

Using neural networks within this approach involves a supervised learning method to understand the model and cost functions. The objective is to converge closely to the deterministic expectations of the target functions. These functions can be dissected into deterministic and noise aspects, with the latter ideally converging to an expectation of zero.

The advantage of model-based RL, as outlined by Sutton [1988], lies in its efficient use of experience. This leads to a refined policy developed with fewer interactions with the environment. Nonetheless, a potential pitfall is that if the model is imperfectly learned, optimisation can inadvertently adhere to this flawed model.

Highlighting a practical merit of model-based RL: consider a robotic aerial vehicle learning primarily from the negative reinforcement of crashes. It is vastly preferable for these mishaps to transpire in a simulated environment rather than in reality. Furthermore, several simulated trajectories can be analysed in the time it takes for a real robot to complete a single trajectory. This efficiency underscores the advantages of model-based methodologies over their model-free counterparts.

2.2.1 Differences in Environmental Dynamics and Observability

Understanding the nuances between different types of environments is crucial for designing and evaluating intelligent systems. This subsection delves into the distinctions between partially observable and fully observable environments and between dynamic and static environments, highlighting their implications for agent behaviour and decision-making processes.

Partially Observable vs. Fully Observable Environments

Partially Observable Environments In partially observable environments, agents cannot access complete state information. Such limitations can stem from imperfect sensors, occlusions, or the intrinsic complexity of the environment. Agents under these conditions must employ sophisticated strategies, including maintaining internal state representations or using historical data to inform their decisions.

Fully Observable Environments Conversely, fully observable environments always provide agents complete access to state information. This transparency simplifies the decision-making process, as agents can make more informed choices based on the immediate observation of their surroundings and the consequences of their actions.

Partially Observable State On the other hand, a partially observable state refers to the segment of the environment's state that an agent can perceive or has information about at a particular moment. It highlights the agent's immediate perspective or understanding of the environment, which is constrained by the limitations of its sensory apparatus or the information made available to it. This concept emphasises the granularity of what is directly knowable by the agent versus what must be inferred or estimated.

Key Differences The key difference between these concepts lies in their scope and focus. While a partially observable environment characterises the general condition of observability within the environment, a partially observable state refers to the specific elements of the environment's state that are accessible to the agent at any time. This distinction is crucial for agent design, as it affects how agents process information, make decisions, and plan actions based on the knowledge they can acquire about their environment.

Dynamic vs. Static Environments

Dynamic Environments Dynamic environments are characterised by their capacity to change over time, often independently of the agent's actions. Such environments pose significant challenges, necessitating continuous monitoring and adaptation from the agent to effectively achieve its goals.

Static Environments In contrast, static environments remain constant unless acted upon by the agent. This stability allows for more straightforward planning and decision-making, as agents can operate under the assumption that the environment will not change unexpectedly between actions.

Conclusion: The differentiation between partially and fully observable environments, alongside the dynamic versus static nature of environments, forms a foundational concept in artificial intelligence and robotics. These environmental characteristics profoundly influence agents' strategies and effectiveness, making their understanding essential for designing robust and adaptive intelligent systems.

This thesis adopts a dynamic environment as a testing ground to challenge the agents' adaptability and decision-making processes under uncertainty. Dynamic environments, characterised by their ever-changing nature, present a realistic and rigorous setting for evaluating the efficacy of the proposed Memory-Based Backpropagation Through Time (MBPTT) algorithm. In such environments, agents are continually faced with new scenarios that require the ability to quickly adapt and make decisions based on incomplete and evolving information. This complexity mirrors real-world situations more closely than static environments, making it an ideal choice for this research.

In addition to the challenges posed by dynamic environments, this thesis specifically focuses on the complexity introduced by partially observable states. A partially observable state, where agents have limited access to the environment's full state information, necessitates the development of sophisticated strategies for effective decision-making. This limitation simulates conditions where sensory data are restricted, or agents must operate with incomplete knowledge of their surroundings, a common scenario in many real-world applications.

The integration of memory mechanisms into the RL and ADP paradigms, as facilitated by the MBPTT algorithm, addresses these challenges head-on. By enabling agents to remember and leverage past experiences, the algorithm significantly enhances their ability to operate in environments where full observability cannot be guaranteed, and conditions change dynamically. This approach allows for a nuanced understanding of the environment over time, fostering a level of adaptability and strategic planning that traditional RL algorithms struggle to achieve.

The decision to use a dynamic environment to challenge the agents with a partially observable state underscores the thesis's commitment to advancing the field of artificial intelligence. It reflects a deliberate move towards creating more robust, flexible, and intelligent systems capable of navigating the complexities of real-world environments. Through this rigorous testing ground,

the thesis demonstrates the potential of memory-augmented learning algorithms to revolutionise how agents learn, adapt, and make decisions in uncertain and ever-changing conditions.

2.2.2 Tackling Partially Observable Environments

Navigating the intricacies of partially observable environments has historically necessitated agents to maintain a belief state—a probability distribution over potential states—which updates with each received observation, reward, and executed action. Foundational insights into these concepts were provided by seminal researchers such as Littman et al. [1995].

In non-stationary environments, Lane et al. [2007] argued for exploiting the environment's topology to facilitate the agent's adaptation, critiquing the conventional use of atomic state-space representations in classical reinforcement learning (RL) algorithms. They proposed envelope-based navigation to leverage spatial characteristics and predict likely encounter locations, effectively focusing on these areas while excluding less relevant ones. However, this approach faced challenges in generalisation due to computational demands, as noted by Finney et al. [2013].

Classical RL's limitations, particularly when interfacing with atomic state-space representations in dynamic goals settings, prompted Lane et al. [2007] to advocate for relational reinforcement learning, introducing "pseudo-relational" learning methods. This approach aimed to mitigate the pitfalls of partial observability inherent in Markov Decision Processes (MDPs) that rely heavily on robust native topology.

The challenge of atomic state representation, which binds MDPs closely to specific policies, complicates the translation between unique state IDs across different MDPs. While relational representations offer a potential solution, Lane et al. [2007] used a restricted form of this representation in their research. Addressing domains where agents must map a history of partial observations to actions, Partially Observable Markov Decision Processes (POMDPs) offer a sophisticated approach. A significant advancement in training POMDP policies came through Recurrent Neural Networks (RNNs), as discussed by Wierstra et al. [2007] and Wierstra et al. [2010]. The latter introduced the Recurrent Policy Gradient (RPG) algorithm for learning memory-based policies, leveraging RNNs to back-propagate estimated return-weighted "eligibilities" through time. This approach demonstrated superior performance compared to other RL algorithms in game-like benchmarks.

Building on this, Hausknecht and Stone [2015] proposed the Deep Recurrent Q-Network

(DRQN), incorporating Long Short Term Memory (LSTM) with a DQN to address POMDPs. They evaluated this approach in flickering Atari domains, observing that RNNs significantly improve performance in scenarios with partial observations.

Wierstra et al. [2010] and Hausknecht and Stone [2015]'s findings underscore the efficacy of RNNs in managing partial state observations, suggesting that simply stacking observations could yield similar performance under certain conditions. This revelation aligns with the exploration of BPTT within this thesis, emphasising its potential to enhance RL strategies for partially observable environments. **Implications for BPTT Comparison:** The success of RPG and DRQN in navigating partially observable environments underscores the relevance of exploring BPTT and other memory-augmented learning methods within this thesis. The comparative analysis with BPTT aims to delve into how memory mechanisms, integral to BPTT, can further advance the field of RL, particularly in complex, dynamic settings where traditional methods face limitations. This section sets the stage for a comprehensive discussion on the integration of BPTT in tackling the challenges posed by partially observable environments, highlighting its potential to contribute novel insights and solutions in advanced reinforcement learning techniques.

2.2.3 Standardisation and Differentiability: Key Pillars in Modern Reinforcement Learning

The evolution of Reinforcement Learning (RL) has spotlighted two foundational concepts critical to its advancement: standardisation and differentiability. Standardisation facilitates a unified approach across diverse RL research and applications, promoting consistency in methodologies, techniques, and terminologies. This uniformity ensures the replicability, comparability, and scalability of research findings and practical solutions. Conversely, differentiability has become paramount, particularly for integrating deep learning within RL frameworks. The ability to apply gradient-based optimisation methods hinges on the differentiability of functions and models, enabling efficient backpropagation processes that enhance agent performance. Together, these principles underscore the ongoing refinement and efficiency of RL developments.

Challenges and Solutions in BPTT Implementation:

One of the primary hurdles in applying Backpropagation Through Time (BPTT) within RL is the necessity for a differentiable environment, as noted by Fairbank et al. [2014a]. While simulation engines such as those developed by Beattie et al. [2016] and Tassa et al. [2018] represent significant advancements, their operators may not always align with the differentiability

requirements of standardised research environments. A practical approach to circumvent this challenge involves creating a differentiable environment that closely replicates the dynamics of conventional research setups. An exemplar of this solution is the adaptation of the "Cartpole" physics to a simplified, differentiable form, as illustrated by Beattie et al. [2016].

Implications for Reinforcement Learning Research:

The emphasis on standardisation and differentiability within modern RL underscores the discipline's progression towards more sophisticated and generalisable methodologies. For BPTT, navigating the challenge of differentiability enhances its applicability across various settings and exemplifies the intricate balance between theoretical innovation and practical implementation. This thesis explores the intersection of BPTT with the core principles of standardisation and differentiability, aiming to contribute to the broader discourse on refining RL techniques for complex, dynamic environments.

Through this exploration, the thesis seeks to highlight the pivotal role of these concepts in the evolution of RL, particularly their impact on the development and application of advanced algorithms like BPTT. The subsequent discussions will delve into the technical intricacies of ensuring differentiability within RL models and the strategic approaches to maintaining standardisation across research and application domains, setting the stage for future innovations in the field.

2.3 Deep Reinforcement Learning: Challenges and Evolution

The advent of Deep Reinforcement Learning (DRL) marked a revolutionary shift in the way agents learn and make decisions, bridging the gap between high-capacity neural networks and reinforcement learning paradigms. DRL uses deep neural networks to approximate previously challenging or infeasible functions with classical methods, enabling agents to process high-dimensional sensory data and learn intricate policies. However, as with any rapidly advancing field, DRL brought challenges. These range from stability issues in training due to the non-stationary nature of data to the problem of exploration in vast state spaces. Additionally, the complexity of neural networks introduced nuances in convergence, optimisation, and generalisation. This section aims to trace DRL's evolutionary trajectory, highlighting its monumental achievements and the hurdles it faced. This section delves into the seminal works, contemporary solutions, and ongoing research to refine the DRL paradigm.

2.3.1 Addressing Deep Reinforcement Learning Challenges in High-Dimensional Sensory Environments

One of the most persistent challenges in reinforcement learning, learning policies directly from high-dimensional sensory input, is addressed by Mnih et al. [2013].

Deep learning has advanced the state of computer vision by extracting high-level features from raw sensory data, as indicated by Krizhevsky et al. [2012], Sermanet et al. [2013], Mnih [2013]. Similarly, breakthroughs in speech recognition have been marked in studies done by Dahl et al. [2011], Graves et al. [2013]. These works used a variety of neural network architectures in supervised and unsupervised settings, including but not limited to the Boltzmann machine [Hinton, 2007], convolutional networks, multi-layer perceptrons, and recurrent neural networks. In particular, the works of Krizhevsky et al. [2012], Sermanet et al. [2013], Mnih [2013] suggest deep learning's potential in both supervised and unsupervised domains. Building on this, Mnih et al. [2013] theorised the benefits of these architectures in extracting high-dimensional sensory data for reinforcement learning.

In contrast to reinforcement learning, many successful deep learning applications leverage vast amounts of hand-labelled training data. However, as Sutton and Barto [2018] elucidates, reinforcement learning primarily learns from scalar reward feedback, which is often noisy, sparse, and delayed.

In supervised learning, the association between inputs and targets is more immediate. The only delays are in reinforcement learning, which fluctuates based on environmental configurations and underlying logic. While Caruana and Niculescu-Mizil [2006] establishes that data samples in supervised learning are independent, Sutton and Barto [2018] reveals that environmental interactions lead to trajectories of events where each is contingent on its predecessor. This chain of highly correlated states can disrupt deep learning processes. Mnih et al. [2013] notes that this is especially problematic when an agent adopts new behaviours, leading to challenges in deep learning frameworks typically suited for fixed data distributions.

Mnih et al. [2013]'s primary approach to address these complications involves deploying convolutional networks, mirroring other works by Jaderberg et al. [2016a], which use convolutional neural networks combined with auxiliary prediction techniques. Their skewed replay buffer, for instance, is adept at forward-predicting rewards for unobserved steps.

In their work, an unsupervised reinforcement and auxiliary learning agent was developed. This agent, powered by a convolutional neural network, interprets pixel data, subsequently

processed by a Long short-term memory [Hochreiter and Schmidhuber, 1996] (LSTM) network to dictate the agent's motions. Various incentives, like wall images or 3D apple-like objectives, guide the agent to apply pixel control on data from sensory observations [Jaderberg et al., 2016a].

Mnih et al. [2013] highlights that while deep learning's prowess is evident in many domains, its efficacy dwindles when deriving control policies from raw video feeds in intricate RL settings. To surmount this, they employed a modified Q-learning [Christopher, 1992] algorithm and updated weights via stochastic gradient descent. Emulating techniques from Jaderberg et al. [2016a]'s work, an experience replay mechanism was instituted to manage correlated data and shifting distributions. For practical evaluation, Mnih et al. [2013] used an Atari game framework to implement and test their solutions.

2.3.2 Overcoming Catastrophic Forgetting By Continual RL

Catastrophic forgetting is a phenomenon where newly acquired knowledge can abruptly overwrite previously learned information. As noted by Kaplanis et al. [2018], one potential culprit of this occurrence is an intense reinforced reward shock or noise. Neural network parameters have scalar values, distinguishing them from brain biochemical models.

A synaptic model incorporating this biological intricacy in tabular and deep reinforcement learning has been demonstrated by Benna and Fusi [2016]. Further expanding on this concept, Kaplanis et al. [2018] contends that merging such biological complexity with reinforcement learning can potentially mitigate the issue of catastrophic forgetting across multiple timescales substantially. Their work posits that continuous learning might diminish the dependence on experience replay databases, addressing within-task forgetting. They argue that "the incorporation of different timescales of plasticity can correspondingly improve behavioural memory over distinct timescales" [Kaplanis et al., 2018].

To validate these ideas, Kaplanis et al. [2018] designed three experimental setups. The initial experiment investigated the feasibility of continual reinforcement learning in a simplified grid-world environment. For this purpose, they implemented the model designed by Benna and Fusi [2016] on tabular Q-values. The experiment involved tasks determining the goal's location within two predefined areas. Constraints like a maximum step count and whether the agent reached its goal were set to dictate the episode's termination. The goal's location was periodically adjusted every 10,000 episodes to foster continuous learning in the agent. Results revealed that the method derived from Benna and Fusi [2016]'s work allowed for more rapid adaptation.

Building on this, a subsequent experiment applied Benna and Fusi [2016]'s approach to deep RL agents. Once again, agents leveraging Benna and Fusi [2016]'s methodology exhibited a marked acceleration in learning and adaptability post-training.

2.3.3 A multi-objective deep reinforcement learning framework

Nguyen [2018] introduced a novel multi-objective reinforcement learning (MODRL) framework that builds upon the principles of DQN. In developing this framework, Nguyen [2018] incorporated linear and non-linear techniques, encompassing single-policy and multi-policy strategies. An essential claim of their research is the system's capability to converge effectively toward optimal Pareto solutions.

To substantiate their approach, Nguyen [2018] contrasted tabular Q-learning with deep reinforcement learning. They emphasised that while tabular Q-learning demands substantial memory and becomes inefficient and impractical in scenarios with large state spaces, deep reinforcement learning primarily uses memory for node and network storage. By comparison, in tabular Q-learning, the Q-table predominantly occupies the memory.

In settings with multiple objectives, deep reinforcement learning can often inadvertently prioritise one objective at the expense of others, causing significant weight shifts that hinder solving for secondary objectives. Addressing this challenge, Multi-Objective Reinforcement Learning (MORL) [Hayes et al., 2021] was developed, allowing for the simultaneous consideration of multiple objectives. This is facilitated by feeding MORL with non-scalar reward types.

However, Nguyen [2018] identified that MORL might experience conflicts among its objective layers. To mitigate this, they proposed the MODRL framework. Their approach entails "learning a single-policy based on a linear scalar of the objective using a fixed set of weights" [Nguyen, 2018].

2.3.4 Reinforcement Learning in Gaming: A Retrospective

Over the past few decades, RL has emerged as a powerful tool for understanding and optimising decision-making processes. The gaming industry, characterised by its interactive and dynamic environments, provides an ideal testbed for RL algorithms. From mastering board games like Backgammon to navigating the complex environments of video games such as Atari, RL has showcased its prowess and adaptability. This retrospective delves deep into the evolution, challenges, advancements, and practical applications of reinforcement learning in gaming.

Evolution of Reinforcement Learning in Game Play

Similar to Mnih [2013], Mnih et al. [2013]'s work, Tesauro [1995] shows one of the famous reinforcement learning success stories where the algorithm was able to learn the backgammon playing the program from scratch by just playing it alone. However, Mnih et al. [2013] claims that the earlier performance and changing the environment setting of reinforcement learning have made the TD-gammon underperform. It was widely believed that TD-Gammon was a particular case for the game setting backgammon.

Moreover, Mnih et al. [2013] believes that the dice roll has helped the agent in exploring the more expansive state space; moreover, Pollack and Blair [1997] claimed that the value function of the backgammon environment was implemented and intended to be smooth, which caused the TD-gammon to perform better.

Practical Applications: Atari Game Experiments

Mnih et al. [2013]'s experiment was on seven different Atari games where the same network architecture was used to learn and work with this variety of games without incorporating game-specific information. Combining stochastic mini-batch updates with experience replay has resulted in state-of-the-art results in six of the seven games introduced by Mnih et al. [2013] with no adjustments in hyper-parameters for each game.

2.3.5 Deep Reinforcement Learning Across Multiple Tasks

When discussing multi-task deep reinforcement learning, a prevailing assumption is that an agent can learn one objective and apply it to multiple tasks using a single learning algorithm. However, as posited by Hessel et al. [2018], while the algorithm can be general enough for learning individual tasks, solutions tend to be specific. Balancing the needs of concurrent tasks becomes critical significantly when resources are constrained.

As noted by Hessel et al. [2018], issues arise when specific tasks overshadow others, leading the agent to prioritise one task over the others. A primary cause is a disparity in reward distribution among tasks. To counter this, Hessel et al. [2018] recommends an automated adaptation of each task's contribution to the agent's updates.

Learning in Discrete and Continuous Environments

Deep reinforcement learning has been effectively applied in relatively simple environments like Go and Chess, as evidenced by Silver et al. [2016, 2017]. Yet, these successes were primarily within discrete space environments. When transitioning to continuous control tasks, notable works include those by Mnih et al. [2015, 2016], Schulman et al. [2015, 2017], Hessel et al. [2018]. However, a standard limitation, as highlighted by Hessel et al. [2018], is the emphasis on mastering individual tasks, necessitating the retraining of agents for different tasks.

Strategies for Enhanced Multi-Task Learning

Various authors have ventured into multi-task control in reinforcement learning, bringing forth novel strategies [Yang et al., 2017]. Notably, the strategy of distilling task-specific expertise into a shared model has been explored, where an agent is trained over diverse tasks and later combined to be responsive across multiple objectives [Parisotto et al., 2015]. Moreover, other techniques, like the off-policy learning of many predictions about the same stream of experience, have been proposed by authors such as Schmidhuber [1990], Pilarski et al. [2011], Jaderberg et al. [2016b]. Additionally, parallel learning strategies, as presented by Sharma and Ravindran [2017], have caught the attention of researchers like Hessel et al. [2018], mainly due to the potential for improved performance across various tasks.

Multi-Agent Reinforcement Learning (MARL)

Beyond the multi-task challenge, an agent often interacts with other agents in real-world scenarios, which the same or different algorithms might drive. This interplay introduces complexities, mainly when agents operate in a shared environment. The Independent Reinforcement Learning (InRL) [Tan, 1993] paradigm treats an agent's experiences as part of its inherently non-stationary environment. The "joint-policy correlation" metric, as introduced by Lanctot et al. [2017], sheds light on the inherent difficulties of policy generalisation across agents.

Reinforcement learning thrives on repetition, where each iteration seeks to refine an agent's policies through feedback from the environment [Sutton, 1988]. In MARL, multiple agents concurrently learn and interact within a shared environment [Lanctot et al., 2017, Shoham et al., 2007, Busoniu et al., 2008]. The environment dictates the agents' relationships: competitive, as seen in Go [Silver et al., 2016] and Poker [Moravčík et al., 2017, Yakovenko et al., 2016, Heinrich

et al., 2015], or cooperative, as in studies presented by Foerster et al. [2016], Catacora Ocana et al. [2019].

Advancements in Multi-Agent Reinforcement Learning

In pursuit of enhancing MARL, various strategies have been introduced. Classic methods involve approximating additional information, such as joint values, as employed by Greenwald et al. [2003], Littman [1994], Kuhn and Tucker [1953], Claus and Boutilier [1998]. While some improvements stem from adjusting update frequencies [Bowling and Veloso, 2002], significant strides have been made in strategies that dynamically respond to other agent actions online [Littman, 2001, Kleiman-Weiner et al., 2016]. Nevertheless, most solutions focus on repeated matrix games and cases with full observability. Tackling partially observable environments, Moravčík et al. [2017] introduced the poker AI ("DeepStack").

Lanctot et al. [2017] has advanced the discourse by introducing a novel metric to quantify policy correlation effects and the subsequent severity of over-fitting. Applying this to partially observable settings using deep reinforcement learning, Lanctot et al. [2017] showcased innovative policy and strategy distributions devoid of gradient sharing among agents. Their experiments effectively diminished the challenges faced by independent reinforcement learners, fostering a setting conducive to both cooperation and competition.

2.4 Memory and Adaptability in Reinforcement Learning

The intertwining roles of memory and adaptability have become defining features in the evolving landscape of reinforcement learning. Historically, agents primarily relied on immediate feedback, navigating static environments and optimising for singular tasks. However, as real-world scenarios grew in complexity and unpredictability, it became evident that agents needed more than just current perceptions; they required a mechanism to remember, reflect upon, and leverage past experiences. This is where memory steps in, providing agents with a rich tapestry of past interactions and aiding in decision-making and strategy formulation. Coupled with adaptability, memory allows agents to seamlessly adjust to changing environments, transitioning between tasks and effectively capitalising on cumulative learnings. This potent combination empowers agents to tackle unforeseen challenges with resilience and agility. This section illuminates the crucial interplay between memory and adaptability in reinforcement learning, charting the

evolution, significance, and methodologies that epitomise their combined strength in shaping robust RL agents.

2.4.1 Implications for This Thesis

The exploration of memory and adaptability in reinforcement learning not only forms the theoretical foundation of this thesis but also directly informs its practical contributions. The integration of memory mechanisms, particularly through the use of Backpropagation Through Time (BPTT), represents a pivotal area of investigation within this research. By examining how memory enhances the ability of RL agents to leverage past experiences for future decision-making, this thesis contributes to advancing the state of the art in RL methodologies.

Moreover, the emphasis on adaptability underscores the practical significance of developing RL agents capable of navigating dynamic environments. The methodologies explored in this thesis, including the application of BPTT and the examination of memory-augmented learning algorithms, are directly aimed at enhancing agent adaptability. This focus aligns with the broader objective of creating RL systems that can effectively transition between tasks, adjust to new environments, and apply learned strategies to novel situations.

2.4.2 Relation to Memory-Augmented Learning

The investigation into memory-augmented learning algorithms, such as those incorporating RNNs and LSTMs, serves as a cornerstone of this thesis. These algorithms exemplify the critical role of memory in enabling RL agents to perform in complex, partially observable environments. By detailing the implementation and evaluation of these algorithms, the thesis showcases how memory mechanisms can significantly improve learning efficiency and decision-making processes in RL agents.

Additionally, the adaptability facilitated by these memory mechanisms is demonstrated through the agents' ability to apply past learnings to new and changing task demands. This adaptability is crucial for agents operating in real-world scenarios, where the ability to assimilate new information and adjust strategies accordingly quickly is a key determinant of success.

2.4.3 Future Directions and Broader Impact

The findings and methodologies presented in this thesis have implications for the field of reinforcement learning but also set the stage for future research. The demonstrated effectiveness

of memory-augmented learning algorithms in enhancing adaptability and performance in RL agents opens up new avenues for exploration, particularly in applications requiring sophisticated decision-making under uncertainty.

Furthermore, the relationship between memory, adaptability, and agent performance elucidated in this research contributes to a deeper understanding of how RL systems can be optimised for various applications. From autonomous vehicles navigating dynamic environments to intelligent systems making strategic decisions in unpredictable market conditions, this thesis's principles of memory and adaptability offer valuable insights for developing advanced RL solutions.

In conclusion, the exploration of memory and adaptability within the context of this thesis not only highlights their intrinsic value to reinforcement learning but also emphasises their potential to revolutionise how RL agents are designed and implemented. By advancing the integration of memory mechanisms and fostering adaptability in RL systems, this research contributes to the ongoing evolution of the field, paving the way for the development of more robust, efficient, and intelligent learning agents.

2.4.4 Adaptability in Reinforcement Learning

In the dynamic and often unpredictable realm of Reinforcement Learning (RL), an agent's ability to adapt becomes a cornerstone for its success. Adaptability is not just about an agent's capability to learn; it's about its resilience in changing environments, flexibility in shifting scenarios, and agility in adjusting to new information. As RL models find applications in increasingly complex and real-world settings, the demand for adaptability escalates. From fluctuating financial markets to robots navigating ever-changing terrains, agents that can seamlessly adjust their strategies are paramount. This section delves into the significance of adaptability in RL, exploring the various methodologies, techniques, and advancements proposed to foster flexible and robust learning mechanisms. This section journeys through memory-based approaches, meta-learning, and the challenges and opportunities they present to ensure that RL agents remain competent and versatile in their learning endeavours.

2.4.5 Usage of Memories to Adapt

The Semi-parametric Topological Memory (SPTM) introduced by Savinov et al. [2018] offers a novel approach for agents navigating unfamiliar environments. It melds a non-parametric graph

and a parametric deep network. Within this system, nodes in the graph symbolise environment locations, while the parametric deep network operates in tandem with the graph, retrieving nodes anchored to specific observations [Savinov et al., 2018]. Notably, this graph-based memory eschews metric information storage, retaining only the connectivity of locations associated with the nodes.

Savinov et al. [2018] delineates the SPTM's role as a planner. Demonstrations in three-dimensional settings showcase agents introduced to an unknown maze equipped solely with a recording of a walkthrough. As the agent traverses the maze, it assimilates information, forging an internal representation. This internal map aids in pinpointing goal locations. The memory graph incorporates an exploration sequence offered to the agent, with an observation o_{T_e} and its explored counterpart denoted as $o^e_{T_e}$. A vertex, say v_i, in the graph houses an observation $o_{v_i} = o^m_i$. Two vertices in this graph are interconnected if:

- Their associated observations are proximate.
- They are chronologically sequential.

To optimise graph quality, Savinov et al. [2018] implemented shortcuts, preventing the formation of inconsequential edges. Performance metrics indicate the SPTM agent's superiority over feed-forward NNs [Zell, 1994] in terms of efficacy.

Elucidating Savinov et al. [2018] methodology:

- Use a graph to capture location data within nodes.
- Provide the agent with a 5-minute maze walkthrough.
- Employ the SPTM system to direct points to another network for action determination.
- Subject current agent observations to localisation, invoking the k-nearest neighbours technique in the vicinity of the prior localisation or applying it globally in case of failure. This can be likened to maze folding, a tactic of folding the maze to discern what captivates the agent.
- Post observation and localisation, the process segues into planning. Here, the Dijkstra algorithm is deployed to discern the shortest path, leading to the final creation of the memory graph.

The Learning Classifier System (LCS) [Urbanowicz and Moore, 2009] and the XCS [Wilson, 1995] (a specific type of Learning Classifier System (LCS)) algorithm are two related techniques in rule-based machine learning. Grounded in genetic algorithms and reinforcement learning, LCS uses two main components: a population of compacts and classifiers, directing inputs to the desired outputs [Zang et al., 2015]. Described as a "condition-action-payoff rule", the classifier system adapts to new and changing environments [Holland et al., 1975]. However, the XCS focuses on prediction accuracy, distinguishing itself from the LCS, where fitness is based on the prediction itself [Wilson and Goldberg, 1989].

According to Zang et al. [2015]'s study, XCS has addressed certain limitations of LCS but still struggles with the absence of a memory mechanism. This restriction confines XCS to optimise policy in a Markov environment, limiting its application in non-Markov settings where complete environmental information is absent. Zang et al. [2015] resolves this issue by enhancing XCS with a memory condition, integrating it with non-Markov property detection.

Experiments conducted with memory-enhanced XCS (XCSMD) [Zang et al., 2015] led to innovative classifications, reducing search space complexity and improving efficiency. Despite this advancement, the technique faced challenges with highly complex aliasing clones.

2.4.6 Deep RL with Successor Features

Zhang et al. [2017] introduced a deep reinforcement learning algorithm based on successor features. The primary advantage of this algorithm, as outlined by the authors, is its ability to learn from past experiences in navigational tasks and subsequently transfer this knowledge to newly introduced challenges. Notably, after mastering the initial task, the algorithm exhibits a reduced learning time for subsequent tasks and demonstrates adaptability in changing environments.

The successor representation is central to the proposed method, which effectively separates the estimation of task-specific rewards from the prediction of expected feature occurrences under certain policy dynamics. This separation positions successor feature-based RL as an optimal choice for knowledge transfer across related tasks.

To validate their approach, Zhang et al. [2017] conducted an experiment where visual input was provided to the Selective Federated Reinforcement Learning (SFRL) [Fu et al., 2023] model. Concurrently, a supervised learning approach was employed to train a Convolutional Neural Network (CNN) [Venkatesan and Li, 2017] to predict actions determined by an A^* planner. Their network architecture resembles the CNN used in the SFRL model, with a notable distinction at

the output level. In their design, the output from 512 units feeds into a concluding softmax layer.

The experimental setup involved a robot tasked with navigation using the proposed method, equipped only with raw sensory data for decision-making. The DQN approach was also evaluated as a baseline for comparative analysis. In their conclusion, Zhang et al. [2017] affirmed the capability of their algorithm to facilitate knowledge transfer across related tasks, thereby promoting more efficient learning.

2.4.7 Meta-Learning

Meta-learning, a former informal notion of cognitive psychology also referred to as "learning to learn", was more recently developed into a formal notion of machine learning [Schaul and Schmidhuber, 2010]. Schaul and Schmidhuber [2010] define meta-learning as learning how to learn in the context of machine learning. Informally, a meta-learning algorithm alters a learning algorithm or the learning process based on experience. The updated learner is more adept at gaining knowledge from new experiences than the original learner.

Meta-learning is approached differently by different researchers. In a setup where a series of different tasks with a shared underlying set of regularities, Thrun and Pratt [1998] state that the rapid improvement of the agent's performance in each new task can be identified as meta-learning. At the architectural level, meta-learning is conceptualised as involving two learning systems: a lower-level (or "inner") system that learns quickly and is primarily responsible for adapting to each new task and a higher-level (or "outer") system that learns more slowly and works across tasks to improve the lower-level system [Prokhorov et al., 2002, Younger et al., 2001].

Some researchers interpret meta-learning in a much stronger sense, requiring it specifically to mean being able to replicate the adaptive weight processes that occur when training neural networks. For example, Cotter and Conwell [1990]'s early work showed how fixed-weight RNNs could approximate an adaptive-weight learning algorithm. They demonstrated this idea by transforming a backward error propagation network into a fixed-weight system. Younger et al. [1999] expanded upon this idea by using an RNN to store knowledge analogous to short-term memory. This method allowed the neural network to learn dynamically, enabling learning to occur continually as part of the net's behaviour.

Younger et al. [2001] and Prokhorov et al. [2002] used the term meta-learning to describe adaptive behaviour with fixed weights in RNNs. However, Lo and Bassu [2001] used alternative terminology by referring to fixed-weight adaptive RNN behaviour as "accommodative" neural

networks. This thesis uses the term "meta-learning" from the definitions by Younger et al. [2001], Prokhorov et al. [2002].

Similarly to Thrun and Pratt [1998], Hochreiter et al. [2001] presented an RNN that was trained on a series of interrelated tasks using standard backpropagation, where the network received an auxiliary input indicating the target output for the preceding step.

Compared with Prokhorov et al. [2002], Hochreiter et al. [2001] focuses on the knowledge-transfer mechanism using RNNs. They argued that using an RNN to represent a differentiable version of a Turing machine was possible. Furthermore, they hypothesised that gradient-based optimisation approaches could derive a learning algorithm from a random starting point.

Santoro et al. [2016] suggested that augmented-memory neural networks (MANNs) can perform meta-learning in tasks that require short- and long-term memory. They used gradient descent to learn an abstract method slowly to obtain valuable representations of raw data (meta-representation). In addition, they incorporated an external memory module to bind never-before-seen information rapidly. Their method was based on Neural Turing Machines [Graves et al., 2014], architectures with augmented memory capacities. These architectures offer the ability to encode and retrieve new information quickly.

The previously mentioned works on meta-learning (i.e. by Cotter and Conwell [1990], Younger et al. [1999], Hochreiter et al. [2001], Prokhorov et al. [2002] and Santoro et al. [2016]) only addressed their algorithm's usage in supervised learning.

One of the recent popular approaches to meta-learning in RL (known as "meta reinforcement learning") is made by Finn et al. [2017], Nichol and Schulman [2018] suggesting a method at the validation time to learn an initialisation of the model-agnostic meta-learning (MAML) [Finn et al., 2017] model. Finn et al. [2017] showed that MAML learns a model that can quickly adapt with a single gradient update, which resulted in a few gradient steps to achieve a good performance. However, the methods provided by Finn et al. [2017] and Nichol and Schulman [2018] do not explicitly consider the necessity of exploring the initial policy. This problem is addressed by Stadie et al. [2018], where they considered per-task sampling distributions as extra information for exploration; exploration for model-agnostic meta-learning (E-MAML) typically consists of a feed-forward policy. This method optimises the per-task sample distributions about the anticipated future returns generated by the post-adaptation policy explicitly during adaptation.

Ortega et al. [2019] proposed a method to recast memory-based meta-learning within a Bayesian framework [Strens, 2000]; this memory-based meta-learning translates the complex problem of probabilistic sequential inference into a regression problem. According to Ortega et al.

[2019], this is accomplished by amortising data that has been Bayes-filtered, with the adaptation being implemented in the memory dynamics as a state-machine with adequate statistics.

2.4.8 Advanced Memory Models in Recurrent Neural Networks

In deep learning, the challenge of sequential data processing has led to the development of various memory models designed to capture temporal dynamics and dependencies. Traditional RNNs paved the way for these advancements, but they suffered from limitations, primarily the vanishing gradient problem, which hindered their ability to model long-term dependencies. Various advanced memory models have emerged over the years to overcome these challenges. These models, equipped with specialised mechanisms, aim to retain information over extended sequences, ensuring that both short-term and long-term patterns within the data are effectively captured. This section delves into some of the most notable advanced memory models that have significantly impacted the field, highlighting their unique architectures, functionalities, and applications.

Full long short-term memory

Full long short-term memory (LSTM) [Hochreiter and Schmidhuber, 1996] is a type of RNN architecture introduced to address the problem of vanishing gradients in standard RNNs. LSTM is designed to learn long-term dependencies by introducing memory blocks and gating mechanisms, allowing the network to selectively retain or discard information over long sequences selectively. Each LSTM cell contains a cell state, a hidden state, and three gating mechanisms: an input gate, a forget gate, and an output gate.

The input gate determines which elements of the current input should be added to the cell state, while the forget gate determines which parts of the previous cell state should be retained or discarded. The output gate controls the flow of information from the cell state to the hidden state. These gates are implemented using sigmoid activation functions, and the cell state is updated using a hyperbolic tangent activation function.

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i), \tag{2.16}$$

$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f), \tag{2.17}$$

$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o), \tag{2.18}$$

$$\tilde{c}_t = \tanh(W_{cx}x_t + W_{ch}h_{t-1} + b_c), \tag{2.19}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t, \tag{2.20}$$

$$h_{t+1} = o_t \odot \tanh(c_t). \tag{2.21}$$

The equations governing the behaviour of an LSTM cell are shown above, where i_t, f_t, and o_t represent the input, forget, and output gates, respectively. $\tilde{c}_t$ is the candidate cell state, c_t is the cell state, h_t is the hidden state, x_t is the input at time t, h_{t-1} is the hidden state at time $t-1$, W and b are weight matrices and bias vectors to be learned during training, and σ and tanh are the sigmoid and hyperbolic tangent activation functions, respectively.

The input gate and forget gate act as filters that selectively update the cell state, while the output gate controls the flow of information from the cell state to the hidden state. By controlling the flow of information, LSTM can selectively retain or discard information over long sequences, making it practical for tasks involving long-term dependencies.

LSTM has been widely used in various applications, including speech recognition, natural language processing, and image captioning. Its ability to learn and retain long-term dependencies makes it an attractive option for tasks that involve sequential data.

Gated recurrent unit

Gated Recurrent Units (GRU) [Cho et al., 2014] is another RNN variant introduced to solve the vanishing gradient problem. It is a simplified version of the LSTM but has proven effective in many tasks. The GRU uses two gates, the reset gate and the update gate, which work together to determine how much past information needs to be passed to the future. The GRU does not have a cell state like the LSTM; it only has a hidden state.

The reset gate helps the model determine how much past information to forget, while the update gate defines how much of the previous hidden state to keep. Combining these gates allows the GRU to capture dependencies over different time scales.

$$z_t = \sigma(W_{zx}x_t + W_{zh}h_{t-1} + b_z), \tag{2.22}$$

$$r_t = \sigma(W_{rx}x_t + W_{rh}h_{t-1} + b_r), \tag{2.23}$$

$$\tilde{h}_t = \tanh(W_{hx}x_t + W_{hz}(r_t \odot h_{t-1}) + b_h), \tag{2.24}$$

$$h_{t+1} = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t. \tag{2.25}$$

GRUs have been found to perform comparably to LSTMs in specific tasks but with the advantage of being computationally cheaper due to the reduced number of parameters.

Content-adaptive recurrent unit

The Content-Adaptive Recurrent Unit (CARU) memory model [Chan et al., 2020b] is a recent development in recurrent neural networks designed to combine the best features of both LSTM and GRU models. The main advantage of CARU over the standard RNN architecture is the ability to capture both short-term and long-term dependencies in sequential data. The model achieves this through an update gate and a reset gate. The update gate controls the information retained from the previous hidden state, while the reset gate controls the amount of information to be reset.

The equations below that govern the behaviour of the CARU memory model consist of the computation of the update gate z_t and reset gate r_t, the candidate hidden state $\tilde{h}_t$, and the updated hidden state h_t. The update and reset gates are computed using sigmoid activation functions, and the candidate hidden state is calculated using the hyperbolic tangent activation function. The final hidden state is added as a weighted sum of the previous and candidate hidden states, controlled by the update gate. Combining these components allows the CARU model to effectively capture and retain relevant information over long sequences, making it a valuable tool for modelling sequential data.

$$z_t = \sigma(W_{zx}x_t + W_{zh}h_{t-1} + b_z), \tag{2.26}$$

$$r_t = \sigma(W_{rx}x_t + W_{rh}h_{t-1} + b_r), \tag{2.27}$$

$$\tilde{h}_t = \tanh\left(W_{hx}x_t + W_{hz}(r_t \odot h_{t-1}) + b_h\right) \tag{2.28}$$

$$h_{t+1} = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t. \tag{2.29}$$

2.5 Exploring Reinforcement Learning Platforms and Engines

The advancement of reinforcement learning (RL) hinges on the algorithms and strategies deployed and significantly on the platforms and engines that support these computational processes. With the maturation of RL, the development of robust, scalable, and user-friendly platforms capable of accommodating a wide array of RL scenarios has become increasingly vital. These platforms form the backbone of RL research, facilitating the development, training, and deployment of RL agents across diverse domains such as gaming, robotics, and financial forecasting.

2.5.1 Evaluation Criteria for RL Platforms and Engines

In assessing the suitability of RL platforms and engines for this thesis, particularly for BPTT in complex neuro-control tasks, several key criteria were considered:

- **Differentiability:** Essential for leveraging gradient-based optimisation methods, particularly in deep learning integration.
- **Environment Complexity:** Support for continuous and dynamic environments closely mimicking real-world scenarios.
- **Scalability and Performance:** Capability to efficiently handle large-scale simulations and parallel computations.
- **Customisability:** Flexibility to modify environments and integrate low-level physics for detailed control simulations.

2.5.2 Selected Platforms and Engines

OpenAI Gym [Brockman et al., 2016]: Offers a comprehensive toolkit for RL research with a wide range of environments and direct access to game physics, making it particularly suitable for experiments involving BPTT. Its openness and flexibility for customisation align well with the requirements of conducting granular-level neuro-control tasks.

DeepMind Lab [Beattie et al., 2016]: Provides a 3D game-like platform designed for AI and ML research, focusing on pixel-to-action interactions. While it offers an intuitive API, its

level of access may not fully meet the granular control needed for BPTT-based neuro-control experimentation.

Brax [Freeman et al., 2021]: Emerges as a high-performance engine optimised for rigid body simulations, built on JAX for enhanced parallelism and scalability. Its ability to simulate thousands of environments concurrently and integrate with JAX-based RL algorithms presents a compelling case for experiments requiring high computational efficiency and differentiability.

2.5.3 Comparative Analysis and Fit for Purpose

Upon comparing these platforms against the specific requirements of this thesis, several conclusions can be drawn:

- **OpenAI Gym** stands out for its extensive environment library and the ability to customise these for specific research needs, making it an ideal choice for BPTT applications in neuro-control tasks.
- While **DeepMind Lab** offers rich interactive environments, its focus on pixel-level interactions may limit its applicability for tasks requiring direct manipulation of environmental physics at a more detailed level.
- **Brax**'s focus on performance and parallelism, supported by its foundation on JAX, makes it a strong candidate for simulations requiring high computational efficiency. However, its specific suitability would depend on the level of environmental complexity and control granularity needed in the research.

2.5.4 Conclusion

The journey through various platforms and engines underscored the realisation that while existing solutions offer substantial capabilities, they fell short of the specific requirements for this thesis. This gap led to the development of custom physics environments tailored to the intricate demands of employing Backpropagation Through Time (BPTT) in complex neuro-control tasks. The creation of these environments represents a pivotal achievement of this research, allowing for unprecedented control and specificity in simulating continuous and dynamic scenarios essential for exploring BPTT.

This bespoke approach enabled differentiability and environment complexity that pre-packaged solutions could not offer, facilitating granular-level manipulations and adjustments

critical for the neuro-control experiments conducted. The custom environments were designed to focus on scalability and performance, ensuring that the computational demands of deep RL, coupled with the intricacies of BPTT, were adequately met.

While necessitating additional effort and resources, the decision to develop proprietary environments underscores this thesis's key contribution to the reinforcement learning field. It highlights the importance of closely aligning the computational tools and frameworks with the research objectives, especially in pioneering areas where off-the-shelf solutions may not suffice.

Future work stemming from this research will explore further optimisations of these custom environments, potential open-sourcing to benefit the wider RL community, and the exploration of integrating advanced computational techniques to enhance simulation efficiency and learning outcomes. This thesis lays the groundwork for such endeavours, demonstrating the feasibility and value of custom solutions in advancing the frontiers of reinforcement learning research.

2.6 Summary and Addressing the Gaps

This thesis revisits the fundamental paradigms of Deep Reinforcement Learning (DRL) and the integration of memory mechanisms to tackle the inherent challenges of stability, exploration, and adaptability in dynamic and partially observable environments. The literature review highlights several key advancements in DRL yet reveals persistent gaps that impede the realisation of fully adaptive and efficient learning agents. This thesis endeavours to bridge these gaps by reintroducing a novel Backpropagation Through Time (BPTT) application within the RL framework, offering new perspectives and methodologies for enhancing agent performance.

2.6.1 Reintroduction of BPTT in RL

- **BPTT and RL Integration:** Unlike traditional approaches, this thesis successfully reintegrates BPTT into RL, providing a robust framework for capturing temporal dependencies and enhancing learning efficiency in dynamic scenarios. This strategic integration addresses the critical challenge of learning optimal policies in environments where the state space evolves.

2.6.2 Improvements in Dynamic and Partially Observable Environments

- **Adapting to Partial Observability:** By implementing memory mechanisms in conjunction with BPTT, the thesis demonstrates significant advancements in managing partially observable states. This combination proves instrumental in improving the decision-making capabilities of RL agents, enabling them to recall and use past experiences more effectively.
- **Performance in Dynamic Environments:** The empirical investigations presented in this thesis showcase the enhanced adaptability and learning efficiency of the proposed MBPTT framework. Tested in dynamic and complex environments, the memory-augmented BPTT approach outperforms traditional RL algorithms, highlighting its potential to navigate the intricacies of real-world applications.

2.6.3 Contributions to Enhancing RL Agent Performance

Integrating memory mechanisms with BPTT represents a pivotal shift in how agents learn and adapt in environments characterised by uncertainty and change. This thesis not only reaffirms the importance of temporal dependencies and memory in learning processes but also:

- Provides a methodological innovation in the form of the MBPTT algorithm, enhancing the capacity of RL agents to learn from sequences of events and actions over time.
- Demonstrates through rigorous experimentation the superior performance of memory-augmented agents in dealing with challenges posed by dynamic and partially observable environments.
- Contributes to the broader field of artificial intelligence by offering a scalable and efficient framework for future developments in agent learning and adaptability.

This thesis lays the groundwork for future research to develop more resilient, adaptive, and intelligent learning systems by addressing the identified gaps in the literature and pushing the boundaries of current methodologies.

Chapter 3

Finding Eulerian Tours in Mazes Using a Memory-Augmented Fixed Policy Function

This chapter explores a novel memory-augmented tabular Q-learning approach designed to navigate mazes with dynamic exit locations. This scenario presents significant challenges to traditional Q-learning algorithms. The primary contribution of this chapter is the development and validation of a memory-augmented tabular Q-learning algorithm that enhances the adaptability and efficiency of maze-solving agents without the computational complexity typically associated with recurrent neural network (RNN) structures.

Maze navigation represents a fundamental problem in RL, offering a clear framework to investigate the exploration-exploitation trade-off. Classical Q-learning, while powerful in many respects, often falls short in environments where conditions within the maze change dynamically, such as in the randomisation of exit locations. The inability of traditional Q-learning methods to adapt their policies post-learning phase [1] limits their effectiveness in navigating through mazes that require continuous adaptation and exploration of new paths.

Freezing Q-values marks the transition to the post-learning phase, where the Q-table remains unchanged, and the agent's policy becomes static. This condition implies that the agent no longer updates its knowledge based on interactions with the environment, which can limit its adaptability in dynamic settings. Unlike vanilla Q-learning approaches, where the policy's adaptability ceases once learning concludes, the memory-augmented strategy proposed here

[1] The "post-learning phase" in this context refers to the stage in tabular Q-learning when the learning process is deemed complete, and the Q-values, which represent the expected rewards for taking certain actions in specific states, no longer update. This phase signifies that the agent's policy no longer adapts based on new information or experiences.

aims to maintain adaptability through an external memory mechanism. This allows the agent to adjust its strategy based on stored experiences, even when the Q-values are frozen. It provides a distinct advantage in environments where conditions can change unpredictably.

Motivated by the need to overcome these limitations, this research introduces a memory-augmented Q-learning algorithm. This innovative approach integrates a memory mechanism that allows the agent to remember and adapt to the environment's dynamics, enabling efficient backtracking and exploration of alternative routes. This memory augmentation is particularly crucial in addressing the challenges posed by mazes with randomised exit positions, providing a robust solution that does not rely on the complexities of RNNs.

The primary distinction between the proposed memory-augmented tabular Q-learning algorithm and traditional (or vanilla) Q-learning lies in the former's enhanced adaptability to environmental changes. Once the learning phase concludes, vanilla Q-learning algorithms typically lack mechanisms to adjust to new information. In contrast, the memory-augmented approach incorporates an external memory that enables the agent to modify its behaviour based on past experiences, effectively allowing for continuous adaptation. This capability is particularly advantageous in maze environments with dynamic exit locations, where the ability to backtrack and explore new paths can significantly improve problem-solving efficiency. The integration of a memory system thus represents a novel solution that transcends the limitations of conventional Q-learning methods, highlighting the potential of memory augmentation in expanding the applicability of reinforcement learning algorithms.

The development of the memory-augmented tabular Q-learning algorithm aligns with the overarching theme of this thesis: enhancing the adaptability and decision-making capabilities of RL agents in dynamic environments. By focusing on the maze navigation problem, this chapter directly contributes to the thesis's exploration of methods that improve agent performance in settings characterised by uncertainty and change. The proposed algorithm exemplifies how augmenting traditional RL approaches with memory mechanisms can significantly extend their applicability and effectiveness, underscoring the potential of such innovations in broader RL challenges.

Furthermore, the solution presented in this chapter addresses a critical aspect of RL research: the ability of an agent to operate efficiently in environments where traditional models and algorithms encounter limitations. By integrating a memory function within the Q-learning framework, this research not only provides a novel approach to maze navigation but also showcases the use of a tabular form of a tabular form of memory augmentation in enhancing RL

agents' flexibility and problem-solving capabilities.

In summary, this chapter significantly contributes to the field of reinforcement learning by presenting a memory-augmented tabular Q-learning algorithm that successfully navigates mazes with dynamic exit locations. This advancement not only solves a specific problem within the domain of maze navigation but also offers insights and methodologies that apply to a wide range of RL challenges, furthering the thesis's goal of developing more adaptable and capable reinforcement learning agents.

3.1 Advancements and Challenges in Memory-Based Learning Systems

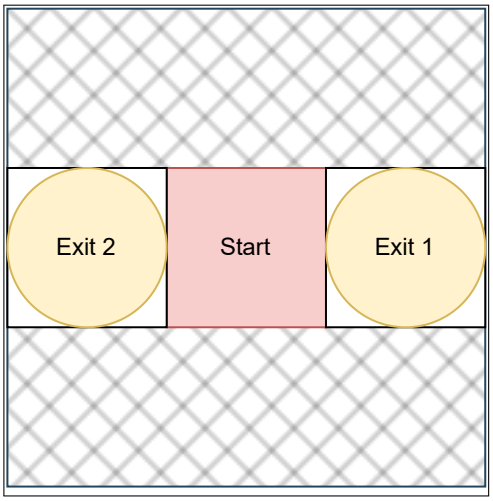
Integrating memory-based learning systems with reinforcement learning (RL) marks a significant advancement in the field, addressing the critical need for efficient information storage and retrieval to enhance learning. With the emergence of Deep Reinforcement Learning (DRL), the landscape of RL has expanded, bringing to light new challenges and opportunities, particularly in complex and dynamic environments where conventional methods struggle.

3.1.1 Memory-Based Systems in Reinforcement Learning

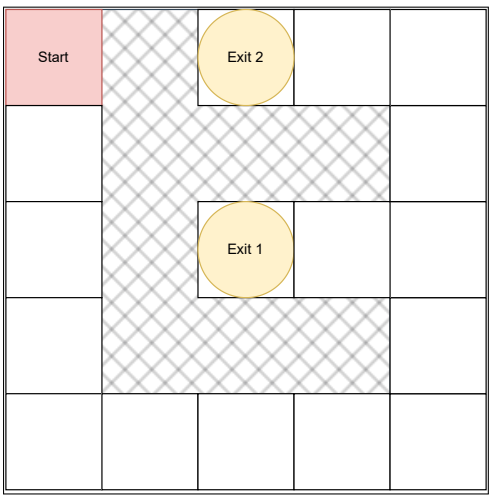
Memory systems have become indispensable for navigating the complexities of RL environments. They provide a mechanism for agents to recall past experiences, facilitating decision-making processes that are more informed and adaptive. The significance of memory-based systems is further highlighted in tasks that require a nuanced understanding of the environment, such as maze navigation, where the ability to remember previously explored paths directly influences the agent's success.

- **Nearest-Neighbor Searches:** This strategy is pivotal for optimisation, allowing agents to find the most relevant experiences in their memory, thereby guiding future actions more effectively [Cayton, 2008].

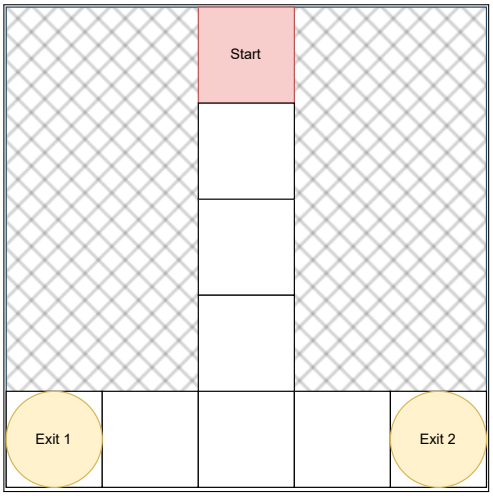
- **Space Decomposition Methods:** By breaking down complex environments into simpler sub-problems, these methods enable a more manageable exploration and exploitation process [Alexopoulos, 1995].



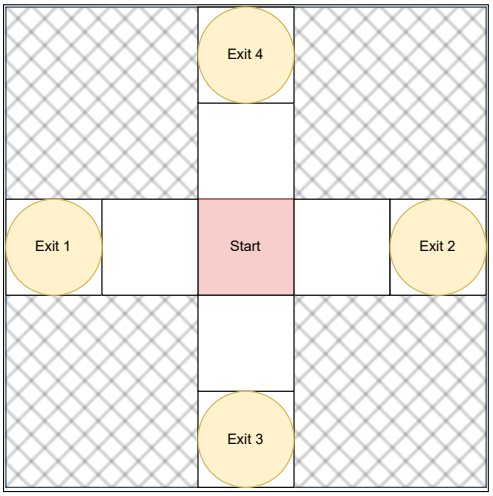
(a) Small corridor maze



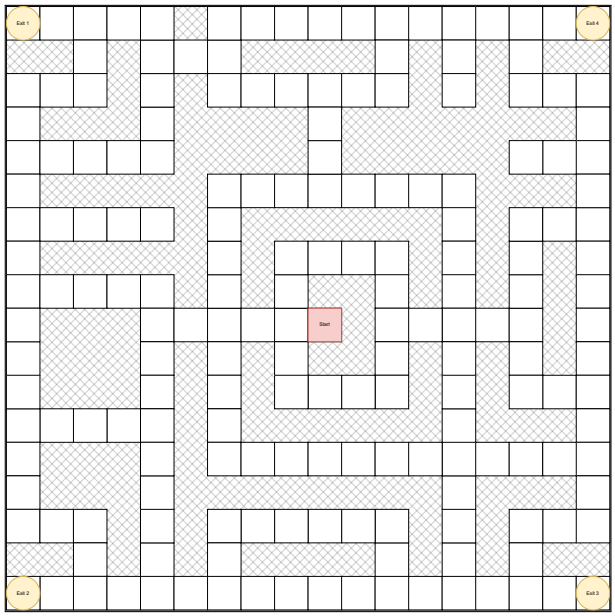
(b) Long corridor maze



(c) T-shaped maze



(d) Cross maze



(e) The complex looped maze with 20 by 20 overall size.

Figure 3.1: Visual representation of the mazes purposed for the experiment.

- **Hierarchical Clustering (HCA):** Employing HCA allows for the organisation of experiences in a way that reflects their hierarchical relationships, enhancing the efficiency of information retrieval [Nielsen, 2016].

The approach detailed in this chapter leverages an external memory system, offering a stark contrast to neural network-embedded memory systems. This distinction is crucial as it underscores the adaptability and simplicity of the proposed method, making it a viable solution even in scenarios where computational resources are limited or where the intricacies of neural networks pose a challenge.

3.1.2 Deep Reinforcement Learning in Partially Observable Environments

The advent of DRL has transformed the RL field, providing tools for learning optimal policies directly from complex, high-dimensional data. Nonetheless, the challenge of partial observability remains a significant hurdle. In environments where agents cannot fully perceive their surroundings due to sensor limitations or data quality issues, memory systems play a pivotal role in compensating for these gaps [Meng et al., 2022, Teh et al., 2020].

This chapter introduces a memory-augmented tabular Q-learning algorithm that directly addresses the challenges posed by dynamic maze environments with randomised exit locations. Unlike traditional deep Q-learning approaches that struggle with changing conditions and partial observability, the proposed system employs a memory table to record and use the agent's historical positions. This innovation enables the agent to dynamically adjust its policy in response to changes, showcasing a significant leap in adaptability and problem-solving capability.

3.1.3 Contributions to the Reinforcement Learning Domain

Introducing a memory-augmented learning approach represents a pivotal contribution to the reinforcement learning domain, particularly in navigating complex mazes with dynamic exit locations. By implementing an external memory mechanism within the tabular Q-learning framework, this work demonstrates a novel method for enhancing agent adaptability and efficiency, circumventing the computational complexities typically associated with recurrent neural network (RNN) architectures.

However, it is important to acknowledge the limitations associated with this approach. While the memory-augmented system significantly improves the agent's ability to adapt to changing

environments, introducing an external memory model increases state space. This enlargement can result in a more extended learning process and slower convergence rates due to the agent's requirement to learn and differentiate between a larger number of states. Such challenges highlight the need for further research to optimise the learning process and improve convergence rates in memory-augmented reinforcement learning models.

Moreover, the current implementation assumes a constant starting point and maze structure, limiting its applicability to environments where such conditions vary. Future work could explore the potential of this approach in more dynamic settings, where both the maze layout and the agent's starting position can change, presenting new challenges for memory management and state representation.

In summary, the contributions of this chapter extend beyond solving a specific maze navigation problem, providing valuable insights into the integration of memory systems within RL. Despite the identified limitations, the advancements presented pave the way for future research to develop more sophisticated, efficient, and adaptable reinforcement learning algorithms capable of tackling the complexities of real-world environments.

3.2 Environment and Agent Definitions

The maze built is a standard discrete-valued reinforcement learning problem where the discrete state space is denoted by $\mathbb{S}$, and the discrete action space is denoted by $\mathbb{A}$. The following concept is implemented and shown in Appendix 1.

The maze is represented by a matrix M of binary values 0 or 1. The matrix (maze) M is of size height $\times$ width. Each matrix element represents a maze cell (1=blocked or 0=open).

For example, the maze in Fig. 3.1b is represented by the matrix:

$$M = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

At time step $t = 0$, the agent starts from a given fixed start point, the requirement for a constant starting point in the context of this memory-augmented tabular Q-learning approach stems from

the need to establish a stable baseline for evaluating the adaptability and learning efficiency of the agent. A constant starting point ensures that variations in the agent's performance are attributable to its learning and memory capabilities rather than changes in initial conditions. This constancy is crucial for accurately assessing how memory augmentation influences the agent's ability to navigate mazes, especially under dynamic exit conditions., e.g. as shown in Fig. 3.1b. Then, at each time step t, the agent has state $s_t \in \mathbb{S}$ and chooses action $a_t \in \mathbb{A}$. Finally, the environment responds by moving the agent to a new state s_{t+1} and giving a real-valued reward r_t.

The agent attempts to move to an open cell by applying a valid action a_t from $\mathbb{A} = \{\text{north}, \text{south}, \text{east}, \text{west}\}$. However, the new state's positional values will not be updated if the arriving cell's $M(y,x)$ returns the value of 1, indicating that the agent collided with the wall.

The arriving cell's value from $M(y,x)$ will determine the agent's new position. To illustrate this explicitly, the standard update equations for moving around a discrete-cell maze are:

$$(y_{t+1}, x_{t+1}) \leftarrow \begin{cases} (y_t - 1, x_t) & \text{if } a_t = \text{North and } M(y_t - 1, x_t) = 0 \\ (y_t + 1, x_t) & \text{if } a_t = \text{South and } M(y_t + 1, x_t) = 0 \\ (y_t, x_t + 1) & \text{if } a_t = \text{East and } M(y_t, x_t + 1) = 0 \\ (y_t, x_t - 1) & \text{if } a_t = \text{West and } M(y_t, x_t - 1) = 0 \\ (y_t, x_t) & \text{otherwise} \end{cases} \tag{3.1}$$

At every time step t, on taking action a_t, the agent receives an instantaneous reward of:

$$r_t = -1, \tag{3.2}$$

regardless of whether the action led to bumping into a wall.

The episode terminates when the agent reaches the exit (where $(y_t, x_t) = (y_{\text{exit}}, x_{\text{exit}})$) or runs out of steps.

The agent repeatedly takes actions, accumulating rewards, until a terminal state is reached (i.e. until the maze's exit is found or the time expires). An episode is a state transition sequence from the start to the terminal state. The total discounted reward accumulated by the agent is:

$$R = \sum_t \gamma^t r_t, \tag{3.3}$$

where $0 < \gamma \leq 1$ is a discount factor.

As purely negative reward is accumulated over the episode (via Eq. (3.2) and Eq. (3.3)), it is to the agent's advantage to get to the exit of the maze as quickly as possible to maximise the total reward received.

Actions are chosen by a policy function $\pi : \mathbb{S} \to \mathbb{A}$. The learning objective is to find a policy function that maximises the expectation of R.

A deterministic policy function π will always specify which unique direction to move. Upon entering a dead-end corridor with a deterministic policy function, it is impossible to backtrack out of that corridor. A memory-based solution is defined for this problem in the following subsection.

3.2.1 Maze environment with memory modification

In each cell of the maze, identified by coordinates (y,x), a memory vector $\vec{c}_{(y,x)}$ is created. This vector is part of a larger table, called C, which has the dimensions $(\text{height}, \text{width}, 4)$. Each element of this memory vector corresponds to one of the four possible actions: north, south, east, or west. Specifically, $C(x,y,a)$ will have a value of one if the agent has previously taken action a from the cell located at (y,x), and zero otherwise. This memory vector is further illustrated in Eq. (3.4).

$$\vec{c}_{(y,x)} = \begin{bmatrix} C(y,x,\text{north}) \\ C(y,x,\text{south}) \\ C(y,x,\text{east}) \\ C(y,x,\text{west}) \end{bmatrix}. \tag{3.4}$$

The table C is initialised with zeros. Then, as shown in Eq. (3.5), the table's vector value $C(y_t,x_t,a_t)$ is updated after the agent takes action a_t from state (y_t,x_t).

The table C will retain the changed values throughout the episode, and when a new episode begins, the table values will revert to 0.

$$C(y_t,x_t,a_t) \leftarrow 1. \tag{3.5}$$

In the above section, the agent's state was described by two numbers, (y_t,x_t). In the case of a maze environment with memory modification, the state is extended to hold additional values so that the state is now described by:

$$s_t = (y_t,x_t,C(y_t,x_t,\text{north}),C(y_t,x_t,\text{south}),C(y_t,x_t,\text{east}),C(y_t,x_t,\text{west}))\,. \tag{3.6}$$

In short, the state is described by:

$$s_t = (y_t, x_t, \vec{c}_{(y_t,x_t)}). \tag{3.7}$$

The policy function without the memory modification was defined by $\pi(y_t, x_t)$. The policy function for the environment with memory arguments is now expanded to hold $(y_t, x_t, \vec{c}_{(y_t,x_t)})$. The policy function for the environment with memory modification is described by:

$$a_t = \pi(y_t, x_t, \vec{c}_{(y_t,x_t)}). \tag{3.8}$$

The C table serves as a memory store by keeping track of the corridors the agent has previously explored. When the agent enters a corridor, the corresponding value in $C(y_t, x_t, a_t)$ is updated to 1, indicating that the action a_t was taken from state (y_t, x_t). Suppose the agent reaches a cell in the maze that has been visited. In that case, the state representation changes as the agent reverses out of the corridor [2].

For example, if the agent only moves east in a corridor, the table $C(y_t, x_t, \text{east})$ is updated to hold 1. When the agent decides to exit the corridor, its state representation changes to $s_t = (y_t, x_t, 0, 0, 1, 0)$, distinct from its state when it entered the corridor $s_{t'} = (y_{t'}, x_{t'}, 0, 0, 0, 0)$.

The change in state representation influences the Q-values associated with actions taken from that state. Consequently, the policy, which is greedy with respect to the Q-values, can differ when moving down the corridor compared to backtracking out of it. This policy change allows the system to effectively backtrack out of dead ends, a feature not otherwise possible without this form of memory.

Comparatively, more straightforward solutions could be considered appealing; for instance, simply remembering whether the agent has previously visited a cell. This would only require a table of shape $(\text{height}, \text{width})$.

It is important to note that this more straightforward design becomes inadequate if the agent accesses the same cell more than twice. The agent can explore only two corridors due to the binary value constraint for each cell, becoming ineffective should the agent need to return to the same cell to explore additional branching paths.

[2] While the C model enhances the agent's ability to navigate mazes by incorporating the memory of previously visited states, it introduces a trade-off in terms of state space complexity. Specifically, incorporating this memory mechanism significantly enlarges the state space, potentially leading to a more extended learning process and slower convergence rates. This expansion arises because the agent must learn a policy that accounts not only for the physical layout of the maze but also for the history of visited corridors, effectively increasing the number of states the agent must differentiate between. As a result, achieving optimal performance may require more time and computational resources, highlighting the importance of considering efficiency improvements and optimisation techniques in future iterations of memory-augmented reinforcement learning models.

3.2.2 Integrating Memory in Tabular Q-learning

Incorporating memory into tabular Q-learning involves extending the standard Q-learning algorithm to work with a new state vector $s_t = (y_t, x_t, \vec{c}_{(y_t,x_t)})$. Here, $\vec{c}_{(y_t,x_t)}$ represents the history of visits for the four neighbouring cells from position (y_t, x_t), as described in Eq. (3.4).

The greedy policy for this augmented Q-function is defined as follows:

$$a_t = \arg\max_{a'} Q(y_t, x_t, \vec{c}_{(y_t,x_t)}, a'), \tag{3.9}$$

where a' denotes all possible actions available to the agent.

The Q-learning update rule for interacting with this augmented memory maze environment is shown below:

$$\Delta Q(y_t, x_t, \vec{c}_{(y_t,x_t)}, a_t) = \alpha \left(r_t + \gamma \max_{a'} Q(y_{t+1}, x_{t+1}, \vec{c}_{(y_{t+1},x_{t+1})}, a') - Q(y_t, x_t, \vec{c}_{(y_t,x_t)}, a_t) \right), \tag{3.10}$$

where $\alpha > 0$ is the learning rate.

Eqs. (3.5), (3.9) and (3.10) together illustrate how the standard Q-learning algorithm is adapted to interact with the newly defined state vector.

3.2.3 Simplified Wall-Following Algorithm

One may argue that established algorithms, such as the "wall-follower" algorithm [Del Rosario et al., 2014], offer an alternative to memory-augmented tabular Q-learning. However, it should be noted that the wall-follower method cannot solve mazes that include loops.

To emulate the behaviour of the "wall-follower" algorithm within a Q-learning framework, the observation state is modified to $\vec{o}_t = (w(\text{west}), w(\text{east}), w(\text{north}), w(\text{south}), d_{\text{entry}})$. In this representation, w values serve as binary sensory indicators, specifying whether a given direction is blocked. The variable d_{entry} denotes the direction from which the agent entered the current cell.[3]

The agent's policy responds to the observation vector $\vec{o}_t$. Given the limited scope of the agent's observations—restricted to its immediate surroundings and the entry direction—the primary strategy for maze navigation defaults to wall-following. This approach leans more toward exploitation rather than exploration.

[3]The concept of $\vec{o}$ is introduced to differentiate it from the s state used in the standard Q-learning framework. $\vec{o}$ is specific to the wall-following strategy.

To facilitate this strategy, one can train a Q-table with dimensions $(2,2,2,2,4,4)$.[4]

Subsequent experiments will aim to compare this wall-following approach with the memory-augmented method.

3.2.4 Proof of sufficiency for solving mazes with memory modification

Binary cell mazes come in various structures, each offering unique pathways to the target. Players must execute a valid action to reach unobstructed adjacent cells to move from one cell to its neighbour.

Tree graphs represent perfect mazes [5]. In these mazes, every cell (represented as tree nodes) is accessible, and there's a single, direct route from any one cell to another (described by tree branches). One can fully explore a perfect maze using an Euler tour, as depicted in Fig. 3.2 [Wikipedia, 2023].

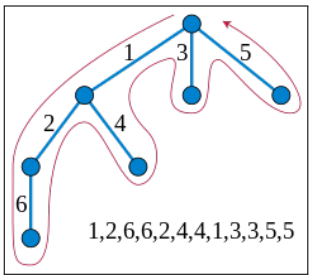


Figure 3.2: Euler tour of tree graph, taken from Wikipedia [2023]

In Fig. 3.2, a sequence of actions can be observed during the Euler tour. This sequence indicates that an agent travels downward until encountering a dead-end, then retraces its steps to the previous branch and explores another uncharted route. The agent recursively continues this behaviour, ensuring every path and leaf node in the tree is visited.

When the exit point in a maze shifts between the leaf nodes in Fig. 3.2 with each episode reset, the agent always starts at the root node; given enough time, the Euler tour guarantees the agent will find the exit, as it will eventually reach every tree leaf.

However, a fixed policy function receiving only (y,x) inputs cannot replicate the actions in Fig. 3.2. For instance, once an agent locates the exit on the tree's far left, it can't navigate back to explore other paths due to the deterministic nature of the policies.

[4] The dimensions $2,2,2,2,4,4$ correspond to binary indicators for wall presence in the west, east, north, and south directions, along with four possible entry and exit directions, respectively.

[5] A *perfect maze* is defined as a type of maze with precisely one path between any two points within the maze. This means no inaccessible sections, circular paths, or open areas are within the maze's structure. In other words, it is a maze that consists of a single maze path without any loops or closed circuits, ensuring that there is only one way to get from the entrance to the exit or between any two points within the maze.

In contrast, input $(y, x, \vec{c}_{(y_t, x_t)})$ enables the agent to differentiate between states, training them distinctly using the extended state values. This capability ensures the agent can backtrack recursively, probing other branches until all paths are explored. This behaviour closely aligns with the Eulerian tree exploration shown in Fig. 3.2.

Two distinct state-action pairs, including loops, might lead to the same cell for non-perfect mazes. For instance, in Fig. 3.2, connecting any two randomly chosen leaves would create a loop. Agents with a fixed policy function would be trapped in this loop because they cannot vary the (y, x) inputs. Hence, they cannot leave the looped maze section.

The ability to perform different actions to reach the same state allows the proposed method to update unique table entries, $C(y_t, x_t, d_t)$, each time it arrives at the same maze cell. This dynamic approach ensures agents explore every reachable cell, regardless of the maze's configuration.

These discussions substantiate that the greedy policy presented by Eq. (3.9) can represent a complete maze tour. This allows maze-solving even when exit locations change. In contrast, the Q-learning approach outlined in Section 2.1.4 requires additional learning to tackle mazes with variable exits.

3.3 Experiment Setup

Five distinct algorithms were intended to be compared to test on five different mazes; in each case, the augmented memory is added to the algorithm, and the results are compared. These algorithms are as follows:

- Conventional Q-learning algorithm.
- Proximal policy optimisation algorithm PPO.
- A synchronous, deterministic variant of Asynchronous Advantage Actor Critic (A3C) A2C.
- Deep Q Network (DQN) builds on Fitted Q-Iteration (FQI), using a replay buffer, a target network and gradient clipping.
- Conventional Q-learning trained to behave like a "wall-following" algorithm.

The PPO [Schulman et al., 2017], A2C [Mnih et al., 2016] and DQN [Mnih et al., 2013] were taken from Raffin et al. [2019]'s mentioned algorithms, and the specific hyper-parameters were

set to their default values. The PPO, A2C and DQN inputs will integrate the external memory, adding more details to their neural network input. For the conventional Q-learning algorithm, two inputs are acquired from the agent's current position (y,x) in case of no memory augmentation. However, for the memory augmented method, the input is expanded to contain six values to additionally hold a vector $\vec{c}_{(y_t,x_t)}$ shown in Eq. (3.7).

For the reinforcement learning algorithms such as PPO, A2C and DQN, the network architecture of Critic and Actor, where relevant for each algorithm, consisted of a multi-layer fully connected neural network. In case of no memory augmentation, the observation will hold the agent $o_t = (y_t, x_t)$ position, where it will be pre-processed into a one-hot encoded form. The input for each reinforcement learning algorithm included a one-hot encoded vector with the shape of the maze height added to the maze width.

In case of memory augmentation, the state is expanded to hold a vector $\vec{c}_{(y_t,x_t)}$ shown in Eq. (3.7) where contains four values in the range of 0 and 1 in addition to the agent's current position, therefore, after one-hot encode pre-processing, the input will hold the sum of maze's height and width added to eight extra values for the memory augmented part of the observation.

Two hidden layers and 64 nodes are designed for each algorithm's network architecture. The PPO and A2C algorithms used the tanh activation function; the DQN algorithm included the ReLU activation function in its network architecture.

In the PPO, A2C and DQN model, the agent's x and y coordinates and each element of the vector $\vec{C}$ are one-hot encoded. This means that for each coordinate or count table element, a binary vector represents it, with only one "1" and the rest being "0"s. For example, if y could be 0 or 1, its one-hot encoding would convert 0 to [1, 0] and 1 to [0, 1].

In this particular case, the neural network receives $(\text{height} + \text{width} + 8)$ inputs, where "height" and "width" are the dimensions of the maze, and 8 comes from one-hot encoding the elements of the $\vec{C}$ vector.

It is worth mentioning that there are alternative ways to one-hot encode the agent's position. One could directly one-hot encode the (y,x) pair as a single integer cell-number on the maze grid. While this approach may offer a more interpretable representation of the agent's state, it results in a less compact representation. Specifically, the length of the one-hot encoded vector becomes $width \times height$, as opposed to the $width + height$ length obtained when one-hot encoding the coordinates separately. Therefore, this method does not reduce the dimensionality of the input

space and could make it more challenging for the neural network to learn.[6]

The choice between these two methods depends on the specific requirements and constraints of the task at hand.

The last layer's activation function for all reinforcement learning algorithms was an identity activation function. The learning rate was set to 0.01, and the discount factor was set to $\gamma = 0.9$; off-policy was chosen for the Q-learning method.

The epsilon-greedy policy [Russell and Norvig, 2016] was used to apply randomness in choosing the actions with a 0.1 chance to occur. The epsilon-greedy policy allows exploration in training, which was removed in the validation phase of the experiments. Each algorithm was given 500 time steps to find the exit in one episode. At the start of each episode, the exit is moved to its next possible cell.

Five different maze structure environments were created, shown in Fig. 3.1. The exit is indicated with a yellow circle and named "EXIT". The starting cell is drawn with a red square; the empty cross-hatched space shows the maze wall where the agent cannot move into these areas. Each maze environment contains two or four possible exit cells (exits) depending on the maze's structure.

Each maze has a set of possible exits, and one exit is chosen for each episode. Hence, when the agent explores the maze, there is precisely one exit it seeks. At the start of each different episode, the exit is rotated through the set of possible exits for that maze, so each time the agent starts a new episode, the exit will have moved from the last time it explored. For each episode, the agent has 500 time steps to find the exit before the time runs out. After experimenting with all the possible exits, the total number of accumulated time steps to reach each exit is recorded.

These environments in Fig. 3.1 hold a common feature where the agent must decide to take one of the divided paths where only one can lead to the exit. The agent starts on the red square cell and must output actions to move into the allowed cells. In a collision with a wall, the agent will remain in the same cell, and one step will count towards the total steps.

The experiment was created to run on 10 different trials; in each trial, the agent was trained for 100,000 episodes, and the weights were updated after the agent took each action during each episode.

The agent must devise a movement strategy to find and visit all possible environmental exits. For instance, in the small 3-cell maze shown in Fig. 3.1a, the optimal number of steps to reach

[6]While converting the (y,x) coordinates to a single integer cell-number provides a unique way to represent the agent's position, it results in a less efficient encoding scheme regarding the vector's length.

the exit cell on the right side of the maze is 1. After reaching the exit cell, the agent's position is reset. Therefore, the total time to reach the exit on the left side of the maze through the already visited exit is 3. To reach both exits, 4 is achieved as the optimal number of steps to reach both exits.

In Fig. 3.1, five test mazes are shown and can be named and described in Table 3.1. First, the table introduces each maze environment, followed by the number of open cells and optimal steps to reach all the exits planned for the maze.

Table 3.1: List of mazes followed by the number of cells capable of being traversed.

Maze Name	Total Traversable Cells
Small Corridor Maze (Fig. 3.1a)	3
Long Corridor Maze (Fig. 3.1b)	18
T-shaped Maze (Fig. 3.1c)	9
Cross Maze (Fig. 3.1d)	9
Complex Looped Maze (Fig. 3.1e)	188

Since the "wall-following" Q-learning algorithm does not need to be trained on each maze, the algorithm will only train on the cross maze shown in Fig. 3.1d, and the algorithm will be validated on the rest of the mazes defined in Table 3.1.

3.4 Results

Learning algorithms such as tabular Q-learning, DQN, A2C, and PPO were tested on each map represented in Table 3.2; the agent performs well when it minimises the accumulated steps. The results are compared against the A^* searching algorithm to reach all the exits in the maze.

For the RL algorithms such as DQN, A2C and PPO, the observation state was hot-one-encoded into a flattened maze representation.

Fig. 3.3 shows the optimal path to reach the possible exits. The white cells indicate the unvisited cells, and the red cells correspond to the agent's path; the cross-hatched areas indicate the blocked areas where the agent cannot enter. Each maze's exits will rotate during the 100,000 training iteration; the policies are frozen after one episode. A fresh evaluation dedicated episode starts with the policies frozen and no epsilon-greedy.

3.5 Discussion

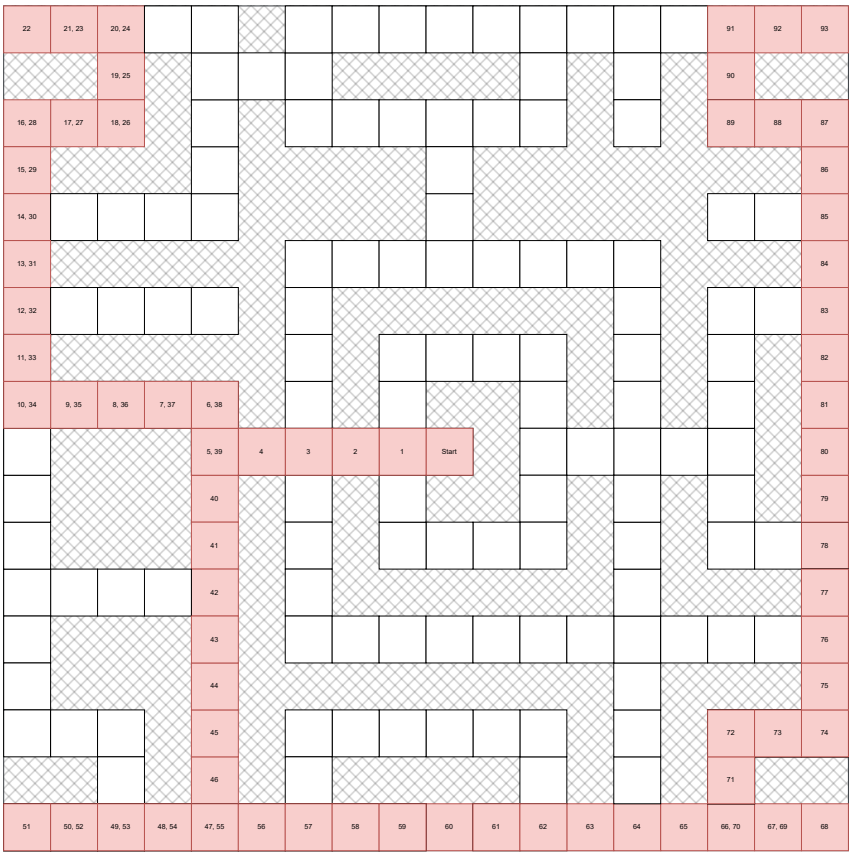


Figure 3.3: Optimal paths to reach the possible exits from the maze shown in Fig. 3.1e.

Table 3.2 shows each algorithm method's average total steps to reach all possible exits in each maze defined in Table 3.1.[7]

This uniform performance, indicated by a variance of 0.0, underscores the algorithm's robustness in navigating the mazes with high reliability. However, it is crucial to acknowledge that the deterministic nature of this result is partly due to the controlled conditions under which the experiments were conducted. In more dynamic or complex environments, where conditions can change unpredictably, a greater degree of variance in the steps required to reach exits may be observed, reflecting the challenges of adapting to new and unforeseen obstacles. The small corridor maze consists of 3 open cells, and the agent starts in the middle of the three open cells. To achieve an optimal accumulated step to reach both exits, the agent has to devise a movement strategy to reach one exit, perform a return movement to the centre, and move to the exit at the other end of the corridor. 4 is the optimal accumulated step to reach both exits on the small

[7]The reported variance of 0.0 in Table 3.2 indicates that, across multiple trials, the algorithm consistently required the same number of steps to reach all possible exits for each maze configuration listed in Table 3.1. This uniformity in performance suggests that the algorithm's behaviour is highly deterministic within the scope of the tested environments. Due to the controlled and static nature of the maze environments, such determinism can arise from the algorithm's ability to learn and apply an optimal or near-optimal strategy reliably, without deviation. It's important to note that while this level of consistency demonstrates the algorithm's effectiveness in these specific maze configurations, real-world applications may introduce variability that could affect performance.

corridor maze. It can be observed that the tabular Q-learning without memory accumulated 501 steps; this indicates that the agent did not find the second exit and ran out of time.

The accumulated total steps by the Q-learning with memory achieved better performance than other algorithm methods such as PPO, A2C and DQN, where the same memory architecture was used to help PPO, A2C and DQN algorithms.

The memory architecture designed significantly helped tabular Q-learning training for $100,000$ iterations. It can be seen in Table 3.2 that the tabular Q-learning with memory achieved 390.8 ± 124.9 average total steps to reach all exits in the complex looped maze, which shows that a maze as big as the complex looped maze requires more time to optimise.

Comparatively, PPO and A2C algorithms with augmented memory performed better than those without external memory. However, DQN's performance suffered from the augmented memory method. The state representation $s_t = (y_t, x_t, \vec{c}_{(y_t,x_t)})$ added unparalleled access to different state representations due to $C(y_t, x_t, a_t)$ update method shown in Eq. (3.5).

The performance of PPO, A2C and DQN did not reach the optimal accumulated steps. It can be assumed that PPO, A2C and DQN needed further adjustments, especially in their network structure, because these algorithms have proven sensitive to hyper-parameters [Olsson et al., 2022].

Table 3.2 shows that the tabular Q-learning agent learned to perform like a "wall-follower" algorithm and solved perfect mazes. However, it can be seen that the algorithm struggled with the complex looped maze as expected. Comparatively, the Q-learning with an external memory solution performed better and reached all exits.

Moreover, there are two potential exits for a maze. In that case, reaching the potentially closer exit is more efficient. The augmented memory tabular Q-learning follows this rule, whereas the "wall-follower" behaviour does not prioritise reaching the potentially more immediate exit first. The state representation given to the "wall-following" algorithm reveals the solution to the Q-table, and it will solve any given perfect maze with no loops. However, the state representation given to tabular Q-learning with memory only reveals the agent's location and neighbouring cells' visit history.

3.6 Chapter Conclusions

This chapter explored a novel memory-augmented tabular Q-learning approach designed to adeptly navigate mazes with dynamically shifting exit locations—a significant challenge for

conventional Q-learning methodologies. Our primary contribution has been the development and empirical validation of this innovative algorithm, which substantially enhances the adaptability and efficiency of maze-solving agents while avoiding the computational complexities typically associated with recurrent neural network (RNN) structures.

Through this investigation, this chapter challenged the traditional reliance on RNNs for managing sequential information in tasks such as maze navigation. RNNs, with their inherent feedback loops, have been presumed necessary for tasks requiring the retracing of steps, such as navigating complex maze structures. However, our findings suggest an alternative perspective, demonstrating that effective backtracking in maze-solving scenarios can be achieved without requiring RNNs. This revelation points to the potential for more straightforward, more computationally efficient solutions that can sidestep the notable challenges of training RNNs—such as computational demand, training difficulties, and the vanishing gradient problem.

Introducing memory augmentation within a tabular Q-learning framework has provided an efficient alternative to imbue maze-solving algorithms with memory capabilities, thereby circumventing the intricacies of RNNs. This memory-enhanced state representation has simplified the navigation process within mazes and showcased significant performance improvements compared to traditional maze-solving strategies, such as the “wall-follower” method.

The results presented in this chapter underline the feasibility and advantages of the proposed memory-augmented approach. Despite the initial hypothesis that RNNs might enhance the capability of agents to retrace steps in mazes, our empirical evidence suggests otherwise, indicating no substantial improvement with the incorporation of RNNs. This finding underscores the effectiveness of the memory-augmented solution and highlights the importance of data-driven conclusions in developing reinforcement learning strategies.

Future research avenues are broad and promising. Investigating modifications in deep reinforcement learning architectures such as PPO, DQN, and A2C, along with experimenting with diverse update techniques, could further reveal the strengths of memory augmentation in the broader realm of RL. Additionally, applying the memory-augmented solution to various discrete space environments could illuminate its broader applicability and potential limitations.

In subsequent chapters, the transition from discrete to continuous spaces will leverage deep learning methods and the Backpropagation Through Time (BPTT) technique to address the unique challenges of continuous environments. This shift is predicated on the insights gained from the current memory-augmented solution to explore and overcome the complexities inherent in such environments.

In summary, this chapter has contributed to reinforcement learning by offering a viable alternative to traditional RNN-based approaches for maze navigation. By demonstrating the efficacy of a memory-augmented tabular Q-learning algorithm in dynamic maze environments, this research addresses a specific challenge within the domain. It paves the way for future advancements in creating more adaptable and capable reinforcement learning agents.

Table 3.2: In the table, performance metrics are reported after 100,000 episodes for each algorithm, using a frozen policy and no epsilon-greedy exploration. The error bars indicate the standard error calculated over 10 trials. The best result for each test case is highlighted in bold. For instance, the bold value of 4.0 indicates that the proposed algorithm has successfully found the optimal path length, serving as a testament to its efficiency and accuracy.

Maze Name	A^* Accumulated Steps	Algorithm method	Accumulated total steps for each exit	
			With external memory	Without external memory
Small Corridor	4	Tabular Q-learning	**4.0** $\pm$ 0.0	501.0 $\pm$ 0.0
		PPO	77.22 $\pm$ 34.61	600.8 $\pm$ 124.47
		A2C	361.33 $\pm$ 72.94	800.4 $\pm$ 81.48
		DQN	445.88 $\pm$ 55.11	**47.2** $\pm$ 10.83
		Wall-Following Q-learning		4 $\pm$ 0.0
Long Corridor maze	30	Tabular Q-learning	**30.4** $\pm$ 0.4	**902.6** $\pm$ 64.93
		PPO	215.0 $\pm$ 67.33	1000.0 $\pm$ 0.0
		A2C	786.44 $\pm$ 84.44	1000.0 $\pm$ 0.0
		DQN	1000.0 $\pm$ 0.0	948.4 $\pm$ 35.07
		Wall-Following Q-learning		221 $\pm$ 0.0
T-shaped maze	16	Tabular Q-learning	**16.0** $\pm$ 0.0	**703.6** $\pm$ 80.67
		PPO	47.66 $\pm$ 8.51	1000.0 $\pm$ 0.0
		A2C	230.22 $\pm$ 105.02	1000.0 $\pm$ 0.0
		DQN	1000.0 $\pm$ 0.0	738.4 $\pm$ 93.82
		Wall-Following Q-learning		42$\pm$ 0.0
Cross maze	32	Tabular Q-learning	**32** $\pm$ 0.0	1701.2 $\pm$ 81.32
		PPO	652.22 $\pm$ 162.91	1352.6 $\pm$ 149.4
		A2C	1452.77 $\pm$ 49.97	1452.2 $\pm$ 156.6
		DQN	1557.44 $\pm$ 55.32	**887.0** $\pm$ 75.32
		Wall-Following Q-learning		32 $\pm$ 0.0
Complex Looped maze	232	Tabular Q-learning	**390.8** $\pm$ 124.9	1951.7 $\pm$ 48.3
		PPO	2000.0 $\pm$ 0.0	2000.0 $\pm$ 0.0
		A2C	1950.55 $\pm$ 49.44	2000.0 $\pm$ 0.0
		DQN	2000.0 $\pm$ 0.0	**1667.9** $\pm$ 77.10
		Wall-Following Q-learning		1731 $\pm$ 0.0

Chapter 4

Navigation in Continuous Space Environments

Embarking on a new chapter in exploring reinforcement learning (RL), This thesis pivots from the structured worlds of discrete environments to the unbounded complexity of continuous spaces. This transition marks a crucial juncture in this thesis, extending the discussion beyond the confines of memory-augmented methods to address the challenges inherent in navigating continuous spaces. Such environments, mirroring the real world's fluidity, demand a nuanced understanding and application of advanced machine learning techniques, including the pivotal BPTT algorithm.

The main contribution of this chapter lies in its rigorous examination of how BPTT, a cornerstone technique for training RNNs to model temporal dependencies, can be adeptly applied to the realm of continuous space navigation. By leveraging deep learning alongside BPTT, This chapter aims to showcase the adaptability and precision of RL agents in environments that more closely resemble the complexities and unpredictability of the real world.

Why does an exploration of environments ostensibly not requiring memory merit attention in a thesis devoted to BPTT and memory mechanisms? The answer lies in the foundational understanding BPTT provides in bridging temporal gaps in data, a critical aspect even in scenarios where memory is not explicitly tested. By investigating navigation in continuous spaces without direct memory requirements, this chapter establishes a baseline for BPTT's efficacy. It sets the stage for more intricate applications involving memory in subsequent studies. This approach allows us to isolate and understand the nuances of BPTT's application in continuous environments, providing a clearer pathway to integrating memory mechanisms in

future work.

Structured around two core sections, this chapter delves into distinct navigation tasks within continuous environments to evaluate BPTT's performance. This chapter defines each task's state space, action space, and reward function, implementing and assessing each algorithm's effectiveness in navigating these complex scenarios. This methodical approach highlights BPTT's versatility and underscores the algorithm's potential to enhance spatial intelligence and decision-making in continuous settings.

The environments selected for this investigation include a simulated 2D navigational agent and the challenge of balancing and navigating a bicycle, as introduced by Randløv and Alstrøm [1998]. By focusing on these continuous spaces, devoid of explicit memory tasks, this chapter lays the groundwork for a comprehensive exploration of BPTT. This strategic choice underscores the chapter's aim to deepen our understanding of BPTT in solving navigation problems within continuous environments, setting a solid foundation for integrating memory in navigation tasks. The subsequent chapter will thoroughly explore this theme (Chapter 5).

In summary, this chapter serves as a bridge, connecting the theory and application of BPTT in the context of continuous space navigation. By navigating the challenges these environments present, this chapter advances the understanding of BPTT. It sets the stage for future explorations into the synergy between memory mechanisms and RL in continuous, dynamic settings.

4.1 Backpropagation Through time

As this chapter delves into the application of Backpropagation Through Time (BPTT) in reinforcement learning and approximate dynamic programming, it is important to note that a comprehensive discussion on the fundamentals and principles of BPTT has been provided in the literature review section of this dissertation (see Section 2.1.3). Readers are encouraged to refer to that section for a more detailed explanation and background on BPTT and its typical applications in supervised learning tasks like time-series forecasting and natural language processing.

4.2 A Simulated 2D Navigational Agent

this research seeks to examine navigational strategies within continuous space environments by developing a bespoke simulation. This environment serves as a testbed for evaluating the

decision-making capabilities of reinforcement learning (RL) agents, specifically focusing on locating constant food sources in this chapter.

The constructed two-dimensional environment is designed to simulate a spatial navigation task, wherein an agent, conceptualised as a simulated organism, identifies the densest point of a food-pile. The food-pile within this environment is mathematically modelled using a Gaussian distribution, parameterised by $(x_{\text{food}}, y_{\text{food}}, z_{\text{food}}, \sigma_{\text{food}})$, where x_{food} and y_{food} represent the coordinates of the food-pile's centre, z_{food} denotes the peak density (or height) of the food-pile, and σ_{food} specifies the spread (or width) of the distribution.

The motivation behind creating this environment stems from the desire to explore how RL agents adapt to and navigate within continuous spaces, where the objectives and obstacles do not adhere to a fixed grid but rather vary fluidly. This environment allows for studying various navigational algorithms and their effectiveness in a context that mimics real-world scenarios more closely than discrete, grid-based environments.

Several considerations drove the choice to model the food-pile using a non-flattening Gaussian distribution. Firstly, a Gaussian distribution provides a smooth gradient of food density, which can be more challenging for an agent to navigate than a binary or flat distribution, offering a richer test scenario for evaluating the agent's navigational abilities.

Moreover, the non-flattening nature of the Gaussian ensures that the food density decreases gradually from the centre, mimicking natural phenomena where resources are not uniformly distributed but instead concentrated in certain areas. This aspect introduces the need for the agent to develop a nuanced understanding of the environment and to employ more sophisticated strategies to locate the area with the highest food density.

Lastly, employing a Gaussian distribution examines how agents deal with uncertainty and partial observability. As the agent moves away from the food-pile's centre, the decrease in food density can be subtle, requiring the agent to infer the food-pile's centre based on incomplete and indirect information. This setup is particularly useful for testing the efficacy of RL algorithms that incorporate memory or predictive models to navigate towards goals in environments with gradual changes and partial cues.

The details of the environment's implementation, including the mathematical model of the food-pile and the algorithmic approach for the agent's navigation, are provided in Appendix 3. This implementation serves as a foundational step towards understanding the dynamics of continuous space navigation and the potential of RL agents to adapt and thrive in such settings.

The development of this two-dimensional navigation environment represents a pivotal com-

ponent of our research into RL agents' adaptability and decision-making processes in continuous spaces. Exploring the challenges posed by a non-uniform food distribution modelled as a non-flattening Gaussian provides insights into the complexities of real-world navigation tasks and the capabilities required of RL agents to solve them effectively. Therefore, this environment serves as a tool for evaluating existing navigational strategies and a springboard for future innovations in RL research.

4.2.1 Environment Implementation

The food-pile's density at location (x,y) is described by the following Gaussian-type function:

$$d(x,y) = z_{\text{food}} + \exp\left(\frac{(x-x_{\text{food}})^2 + (y-y_{\text{food}})^2}{\sigma_{\text{food}}^2}\right). \tag{4.1}$$

Here z_{food} is a constant specifying the food-pile height, and σ_{food} is a parameter governing the width of the food bump. See Fig. 4.1.

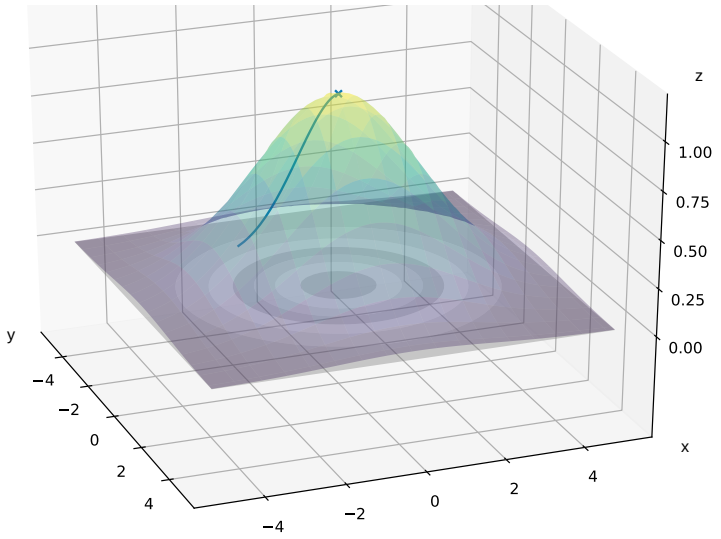


Figure 4.1: Simple food density distribution, where the height of the Gaussian bump indicates food density. Pathway shows an example solution trajectory found by the agent, starting at the bottom and finishing at the "x".

As the agent moves across the environment at discrete time step t, it has location (x_t, y_t), and it uptakes the food at its new position (x_t, y_t). The food supply at any location is deemed inexhaustible. Hence, the total food accumulated during an episode of length L steps is given by:

$$R = \sum_{t=0}^{L-1} \gamma^t r_t, \tag{4.2}$$

where $r_t = d(x_t, y_t)$ and $0 \leq \gamma \leq 1$ is a discount factor.

The objective of the problem is for the agent to learn to rapidly walk to the top of the food pile to maximise the food (reward) accumulated in a fixed finite episode length $L = 30$. Furthermore, the food pile's location and height are static; the agent's initial position is randomised around the food source.

4.2.2 Agent Properties

At each time-step t, the agent generates a two-dimensional action vector $\vec{a}_t \in \mathbb{R}^2$. The agent's velocity $\vec{v}_t$ is determined using the following equation:

$$\vec{v}_t = v_{\max} \tanh(|\vec{a}_t|)\widehat{a}_t. \tag{4.3a}$$

In this context, $|\vec{a}_t|$ represents the Euclidean length of $\vec{a}_t$. The normalised vector $\widehat{a}_t$ is obtained by dividing $\vec{a}_t$ by its magnitude $|\vec{a}_t| + \varepsilon$, with $\varepsilon = 10^{-6}$. This normalisation process scales the vector to unit length, ensuring its direction is preserved while avoiding division by zero. A maximum velocity value is $v_{\max} = 0.2$. The tanh function in Eq. (4.3a) ensures that the agent's speed remains below $v_{\max}$.

The construction of Eq. (4.3a) confines the velocity vector $\vec{v}_t$ within a circle of radius $v_{\max}$. This allows the agent to move freely in any direction.

The agent's position is updated based on the velocity vector $\vec{v}_t$ using the Euler method with a discrete time step $\Delta T = 1$:

$$\vec{p}_{t+1} = \vec{p}_t + \vec{v}_t \Delta T, \tag{4.3b}$$

where $\vec{p}_t = (x_t, y_t)$ is the agent's position at the specified time-step.

In this chapter, the food-pile was positioned at $(0,0)$ with parameters $z_{food} = 1$ and $\sigma_{food} = 8$. To prevent the agent from memorising a fixed solution, the initial positions of the agents were uniformly randomised within the range $x \in [-5,5]$ and $y \in [-5,5]$ at the beginning of each episode.

4.2.3 Agent brain for organism

A neural network with weight vector $\vec{w}$ chooses the actions and memory state of the agent. "Agent brain" refers to the neural network used in the algorithm, but this neural network is also known as the "actor" in the RL literature or the "action network" in the ADP literature.

At any time-step t, the agent receives an observation vector $\vec{o}_t$,

$$\vec{o}_t = (x_t, y_t). \tag{4.4}$$

The neural network structure used in the ADP/RL algorithms consisted of one hidden layer with 20 nodes (as depicted in Fig. 4.2). The tanh activation function was employed in the hidden layer, while no activation function was used in the output layer.

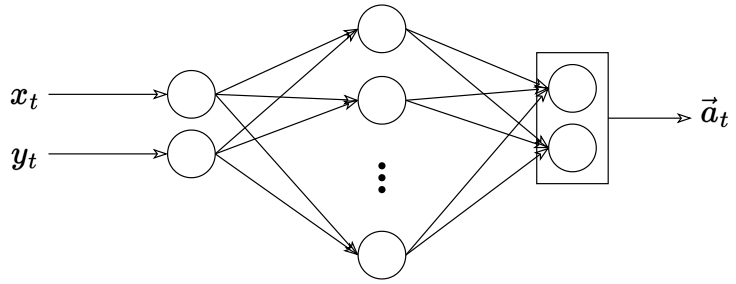


Figure 4.2: Neural Network structure used in the experiments conducted in "A Simulated 2D Navigational Agent" experiment.

4.2.4 Experiments

Four different selected RL algorithms and a BPTT algorithm were used in the experiment. The following RL algorithms were used in all experiments: Advantage Actor-Critic (A2C) (Section 2.1.7), Soft Actor-Critic (SAC) (Section 2.1.7), Deep Deterministic Policy Gradient (DDPG) (Section 2.1.7) and Twin Delayed DDPG (TD3) (Section 2.1.7). These RL algorithms' implementations came from the Stable-Baselines package [Raffin et al., 2019] (version 1.1.0). These state-of-the-art algorithms were selected considering their successful history in different benchmark environments [Brockman et al., 2016]. Additionally, the implementation of BPTT was created and customised personally. Each algorithm trained for 100,000 iterations and was tested across 20 separate trials. The hyper-parameters used by each algorithm were:

- The DDPG, SAC, and TD3 algorithms used 100 batches in replay memory with a buffer size of 10^6. Neural network weights were optimised with an Adam optimiser, with a learning rate of 0.001. The DDPG algorithm used the soft update coefficient ("Polyak" update) of 0.005, and its default discount factor was 0.99.

- In the A2C algorithm included a discount factor of 0.99 and a value function coefficient for the loss calculation of 0.5. This algorithm used RMSprop optimiser with $\varepsilon = 10^{-5}$. In

addition, the A2C algorithm used action noise exploration with an entropy coefficient of 0.001.

- BPTT used the Adam optimiser with a learning rate of 0.001.

In the Stable Baselines package, the four primary RL algorithms had the above hyper-parameters set as default values. However, for the experiment in this chapter, the learning rates were tweaked to achieve the best possible outcomes.

The batching method in BPTT implementation enabled the algorithm to process 100 complete trajectories per iteration (i.e. per update of the weights in the neural network). Since the length of each trajectory (episode) was $L = 30$, the BPTT algorithm allowed 3,000 environment steps for each weight update. To match this as closely as possible for the selected RL algorithms, the hyper-parameters were set to match the provided hyper-parameters in the Stable-Baselines package to be `train_frequency=3000` and `gradient_step=-1`. This ensures the algorithm updates the weights in the neural network for every 3,000 environment interactions taken by the agent. When it does so, these weight updates are accumulated over those 3,000 environment steps.

All the selected RL algorithms (DDPG, A2C, SAC and TD3) consist of *actor* and *critic* networks. The actor-network is structured as described above, i.e. with two inputs (x_t, y_t) and two output nodes $(\vec{a}_t)$.

All of the critic networks used by the RL algorithms were Q-networks. These implement a function $Q : (\vec{o}_t, \vec{a}_t) \to \mathbb{R}$. Hence, the critic network has four input nodes (two for $\vec{o}_t$, and two for $\vec{a}_t$) and one output. There is one hidden layer of 20 tanh nodes (like in Fig. 4.2) and no activation function on the final layer.

The network used in BPTT did not have a Critic network, however. Instead, it only required an action network with a structure and purpose identical to the above actor networks.

4.2.5 Results

The results below display the mean value and its 95% confidence intervals, calculated over 20 trials using the "Seaborn" software library.

Fig. 4.3 illustrates that BPTT (Backpropagation Through Time) effectively solved the simplified version of the problem, exhibiting stable and robust performance. However, the selected

state-of-the-art RL algorithms also achieved approximate solutions to the problem, although they exhibited slower learning progress, instability, and lower final total rewards than BPTT.

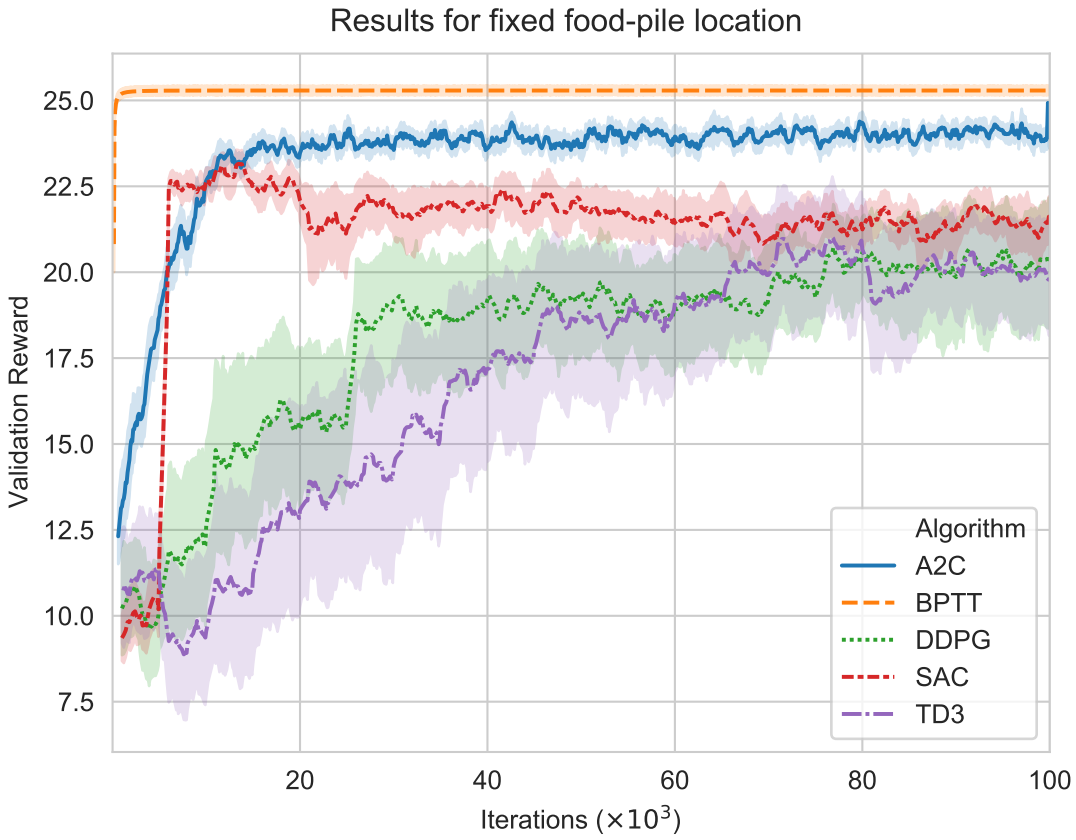


Figure 4.3: Algorithms' performance on fixed-food location validation environment over 100,000 iterations and averaged over 20 trials.

From the final iteration results depicted in Fig. 4.3, a sample of ten trajectories generated by the agent trained with BPTT was selected. These trajectories are visualised in Fig. 4.4, showcasing the agent's movement behaviour. The circles in Fig. 4.4 represent the initial positions of the agents, while the crosses indicate the goal points within the environment. The arrows indicate the direction of agent movement, clearly illustrating their direct paths toward the fixed food-source location.

Based on Fig. 4.4, it is evident that the BPTT agent successfully optimally navigated towards the goal. This observation further supports the significance of the average reward (25.5) achieved by the BPTT method, as demonstrated in Fig. 4.3.

4.2.6 Discussion

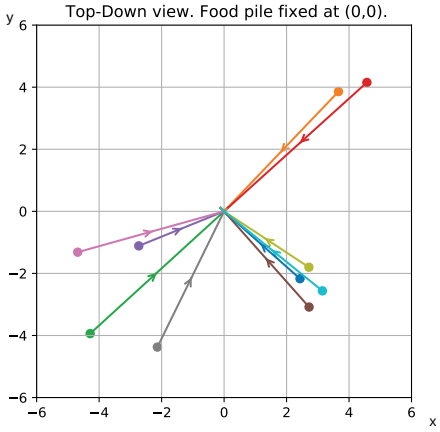


Figure 4.4: Top-down view of BPTT results for the simplified experiment (with no sensor or recurrent memory)

The experimental outcomes, as illustrated in Fig. 4.3, underscore the efficacy of the Backpropagation Through Time (BPTT) method in the navigation task within a continuous space environment. The BPTT algorithm's attainment of an average reward of 25.5 demonstrates its capability to foster significant learning progress and stability across trials.

While the comparison with other approaches reveals BPTT's robust performance, it is essential to contextualise these findings within the broader landscape of reinforcement learning research. The term "notably competitive outcome" is used here not to assert BPTT's superiority over state-of-the-art algorithms that may be highly specialised or tuned for specific tasks but to highlight its potential as a versatile and effective tool for navigation tasks, even in its more foundational or "vanilla" form.

The noteworthy aspect of BPTT's performance lies in its ability to achieve commendable results without extensive task-specific tuning. This characteristic is particularly valuable in reinforcement learning, where the adaptability of algorithms to varied tasks without requiring intricate adjustments can significantly expedite the development and testing of RL agents across different environments.

The findings suggest that BPTT, even when applied relatively straightforwardly, can optimise an agent's navigation strategy effectively, enabling efficient goal-reaching behaviours. This efficiency indicates BPTT's potential as a foundational technique that, while not necessarily outperforming highly specialised state-of-the-art methods, offers a solid baseline from which further enhancements and task-specific adaptations can be developed.

Given the performance of BPTT in this navigation task, future research could explore several avenues to build upon these findings. One potential direction is the integration of BPTT with other

deep learning enhancements or reinforcement learning strategies to further refine its efficiency and effectiveness. Additionally, investigating BPTT's application in more complex environments or tasks requiring sophisticated decision-making and adaptability could provide deeper insights into its capabilities and limitations.

Ultimately, such research would aim to push the boundaries of what BPTT can achieve in various contexts and contribute to the broader understanding of how similar "vanilla" algorithms can be leveraged or improved to meet the challenges of advanced reinforcement learning tasks.

In the comparative analysis of the selected algorithms, while BPTT demonstrated superior performance, it is essential to acknowledge the approximate solutions achieved by other state-of-the-art RL algorithms. Though these algorithms reached the goal, their learning progress was slower and more unstable than BPTT. This suggests the selected RL methods for this task needed more testing with hyper-parameters and network structure to find the optimal values.

Fig. 4.4 provides visual evidence of the BPTT agent's navigation behaviour. The direct paths taken by the agent towards the fixed food-source location indicate a successful optimisation process. This optimal behaviour further supports the significant average reward achieved by BPTT in Fig. 4.3.

4.3 Balancing and Navigating a Bicycle Introduced by Randløv and Alstrøm [1998]

Balancing and navigating a bicycle is a complex task humans have mastered through years of practice and experience. It involves a combination of sensory information processing, muscle coordination, and fine-tuned motor control. Although the physics of bicycle motion and control have been studied for over a century [Dieltiens et al., 2020], there is still much to learn about how humans achieve balance and navigate their bikes. Recent advances in deep learning and neural networks have provided new tools for studying the control of dynamic systems like bicycles. One such tool is the Backpropagation Through Time (BPTT) algorithm.

This section explores using Backpropagation Through Time (BPTT) to control the bicycle model proposed by Randløv and Alstrøm [1998]. The bicycle model is a mathematical representation describing a bicycle's motion dynamics. To render the model compatible with BPTT, necessary modifications are made to transform it into a differentiable system, allowing the computation of gradients concerning its input.

As indicated in Section 2.1.3, BPTT finds extensive application in sequence prediction tasks, particularly within recurrent neural networks (RNNs). However, in control tasks or traditional reinforcement learning (RL), BPTT is less common than alternative techniques like Q-learning or policy gradients.

Moreover, BPTT encounters other limitations. Handling long sequences poses challenges due to vanishing or exploding gradients, especially in tasks with extended time horizons. This may hinder the ability to effectively capture long-term dependencies and optimise control policies across extensive sequences.

Furthermore, this section studies the effects of reward shaping on BPTT performance. Reward shaping involves modifying the reward signals given to the agent during learning to provide additional guidance and promote faster convergence. The impact of different reward-shaping strategies on BPTT's ability to learn effective control policies for the bicycle model is investigated.

This section aims to use this differentiable Randlov's bicycle model to train a neural network to learn how to balance and navigate a bicycle in real-time. The hypothesis is that by using BPTT and a differentiable bicycle mode, a neural network can be trained accurately to predict the bicycle's motion and control inputs based on sensory information, such as the rider's body position and the bicycle's orientation.

4.3.1 Randløv and Alstrøm [1998] bicycle environment

The Randløv and Alstrøm [1998] bicycle model is a mathematical model that describes a bicycle's motion dynamics. It was developed by Danish engineer and computer scientist J. J. Lukkassen Randlov in 1998. The model is based on the physical principles of bicycle motion, such as the effects of gravity, gyroscopic forces, and friction. It has been used to study the stability and control of bicycles.

The bicycle implemented is a rigid body with two wheels and a frame. The bicycle's motion is described by a set of equations that govern the angular position, velocity, and acceleration of the bicycle's various components.

The model considers several factors that affect the bicycle's motion and control, such as the rider's body position, the bike's orientation, and the handlebar's orientation. These factors are all combined in the model to produce a set of equations that describe the motion and control of the bicycle in real-time. Appendix 2 will delve into the mathematical equations and detailed

explanations that elucidate the physics behind the bicycle mode. Please refer to Table 1 for detailed information about the symbols.

Modifications to Randlov's model for it to be differentiable

In the original code provided by Randløv and Alstrøm [1998], the if-else statement makes the code non-differentiable when the condition is checked. Specifically, the line:

```
if front_term > 1:
    front_term = sign(psi + theta) * 0.5 * np.pi
else:
    front_term = sign(psi + theta) * arcsin(front_term)
```

Listing 4.1: Original code for the front tyre position calculation

To address this, the conditional is transformed into a differentiable expression using the "TensorFlow" functions. The transformed code snippet is as follows:

```
r_f = tf.where(theta == 0., tf.constant(1.e8),
                            safe_divide(1, tf.abs(tf.sin(theta))))
front_term = psi + theta + tf.sign(psi + theta)
                            * tf.asin(v * df / (2. * r_f))
```

Listing 4.2: Differentiable code for the front tyre position calculation

In this transformed code, the if-else statement is replaced with the TensorFlow function "tf.where()", which performs element-wise selection based on a condition. The condition "theta == 0". checks whether the value of theta is zero. If it is, it sets the value of "r_f" to a large constant 1.e8, effectively removing the division by zero issues and ensuring differentiability.

In other cases, the value of "r_f" is calculated as "$\frac{l}{|sin(theta)|}$". The term "$2 * r\_f$" is used to scale the argument of the "asin()" function, ensuring that the function's input remains within the valid range of $[-1, 1]$.

In this transformed code, front_term is calculated using TensorFlow's "$tf.sign()$" and "$tf.asin()$" functions, allowing it to avoid the non-differentiable condition and ensure differentiability throughout the computation.

The "NumPy" functions and "if and else" conditions were converted by "Tensorflow" functions; please refer to Appendix 2.

Study in reward shaping of the environment

The reward segment of the task comprises both a positive reward and a penalty, motivating the agent to move towards the goal. Calculating the agent's progress towards the goal is achieved through the displacement of the agent to the goal. The normalised displacement of the agent concerning the goal ($\widehat{Displacement}$) is computed using the following two steps:

Step 1: Displacement Calculation

The first step involves computing the displacement vector $\overrightarrow{Displacement}$, which represents the difference between the agent's position before the action was applied $\overrightarrow{Bike_t}$ and the bicycle's position after the action was applied ($\overrightarrow{Bike_{t+1}}$):

$$\overrightarrow{Displacement} = \overrightarrow{Bike_{t+1}} - \overrightarrow{Bike_t}, \tag{4.5}$$

and computing the difference vector $\overrightarrow{Difference}$, which represents the difference between the goal position $\overrightarrow{goal}$ and the bicycle's current position after the action is applied ($\overrightarrow{Bike_{t+1}}$):

$$\overrightarrow{Difference} = \overrightarrow{goal} - \overrightarrow{Bike_{t+1}}. \tag{4.6}$$

Step 2: Normalised Displacement Calculation

The normalised displacement vector $\widehat{Difference}$ is then obtained by dividing the displacement vector by its Euclidean norm:

$$\widehat{Difference} = \frac{\overrightarrow{Difference}}{\|\overrightarrow{Difference}\|}. \tag{4.7}$$

Here, $\|\overrightarrow{Difference}\|$ represents the magnitude of the $\overrightarrow{Difference}$.

To calculate the current reward (r_t), the dot product is taken between the displacement vector $\overrightarrow{Displacement}$ and the normalised difference vector $\widehat{Difference}$:

$$r_t = \overrightarrow{Displacement}_i \cdot \widehat{Difference}_i. \tag{4.8}$$

By employing the dot product in the reward calculation, the agent is effectively rewarded for moving closer to the goal along the direction of the goal position. This encourages the agent to take actions that bring it closer to the goal location and promote successful navigation towards the desired objective.

In addition to the positive reward that motivates the agent to move towards the goal, the reward system also includes a penalty component. This penalty is a guiding mechanism to encourage the agent to exhibit desired behaviours and avoid undesirable actions during the

navigation task. In the context of riding a bicycle, the penalty is designed to shape the agent's conduct concerning three key parameters: θ (the angle of the bicycle's handlebar), ω (the roll orientation of the bicycle), and ψ (the orientation of the bicycle towards the goal).

The penalties used a flat-bottomed barrier function, which serves as a tool to apply penalties smoothly and controlled. It penalises conditions based on the penalty value, with the penalty increasing as the condition deviates from the desired behaviour. However, instead of creating steep penalties, the function introduces a flat-bottomed region, where the penalty remains constant until the condition crosses a certain threshold.

The function starts by normalising the penalty value with respect to "k_{width}" and then shifts and scales the penalty function to create a flat-bottomed region. The maximum value of the penalty is attained when the normalised penalty value exceeds 1, resulting in the penalty function rising sharply outside the flat-bottomed region. The "k_{power}" parameter determines the steepness of the penalty rise.

By employing the flat-bottomed barrier function, the penalties are enforced in a controlled manner, preventing sharp discontinuities in the optimisation landscape. This ensures stable and smooth convergence during the optimisation process, enabling effective exploration of the solution space and better handling of constraints in the task. The penalty calculation is defined as follows:

$$penalty_{FBBF}(value, k_{width}, k_{power}) = \max(value/k_{width} * 0.5 - 1, 0)^{k_{power}}. \tag{4.9}$$

Each parameter above was considered in calculating the total penalty; the penalties were calculated by:

$$\theta_{penalty} = penalty_{FBBF}(|\theta|, 1.25, 8), \tag{4.10}$$

$$\omega_{penalty} = penalty_{FBBF}(|\omega|, \frac{\pi}{15}, 8), \tag{4.11}$$

$$\psi_{penalty} = penalty_{FBBF}(|\psi|, \frac{\pi}{2}, 8). \tag{4.12}$$

The total reward applied at each time step was calculated by:

$$R = -1(\theta_{penalty} + \omega_{penalty} + \psi_{penalty}) + r_t. \tag{4.13}$$

This reward is applied at each time step and summed over the entire trajectory, effectively guiding the agent's learning process.

4.3.2 Agent brain used in navigating the bicycle

The agent used the similar structure and model used in Fig. 4.2; however, the neural network used had two hidden layers with tanh activation function and consisted of 24 nodes, the output layer used tanh activation function, and each layer was initialised with a normal distribution of 0.001; the learning rate was set to 0.001.

4.3.3 Experiment setup

As mentioned before, the observable values such as ω, $\dot{\omega}$,θ, $\dot{\theta}$,$\sin\psi_g$ and $\cos\psi_g$ are used for the network input.

The initial experiment considered an early termination approach that could improve the learning process in the bike balancing task; the experiment explored stopping the iteration when certain conditions were met.

Then, it was considered to let the bike move while it was touching the ground from its sides; it was then discovered that smoothly altering such penalty functions can greatly aid learning. The initial randomised weights can cause the action network to fail if Eq. (4.9) generates a trajectory with significant errors while the agent is touching the ground due to failing the balancing task; the accumulated significant penalties will prevent the agent from finding a solution.

This confuses the learning algorithm because the learning algorithm focuses on where the error gradients are most significant, which occurs only after the system has already lost control of the tracking reference target (i.e. at the end of the learning process).

To ignore the early termination approach, one solution was to apply a tanh around the penalties; the choice of the tanh provided these desired properties, such as smoothly binding the penalty between 1 and -1, and in case of the flattest part of the penalty curve derivatives are set to 1 or -1. The modified version of the reward function is calculated by:

$$R = -1\tanh\left(\theta_{penalty} + \omega_{penalty} + \psi_{penalty}\right) + r_t. \tag{4.14}$$

It was considered to randomise the x value of the agent's position and keep the $y = 0$; for each agent, a goal is placed 30 units in front of the agent's starting position.

The impact of the tanh method to ignore the early termination approach on the agent's performance was measured, and the experiment aimed to demonstrate whether this approach could lead to more stable and efficient learning, ultimately improving the agent's ability to navigate towards the goal effectively while avoiding significant learning pitfalls.

The experiments were set to allow the agent to balance the bike for 1000 updates to the physics model, and the results were gathered over 10 different trials. The weight update was done over 1000 iterations.

4.3.4 Results

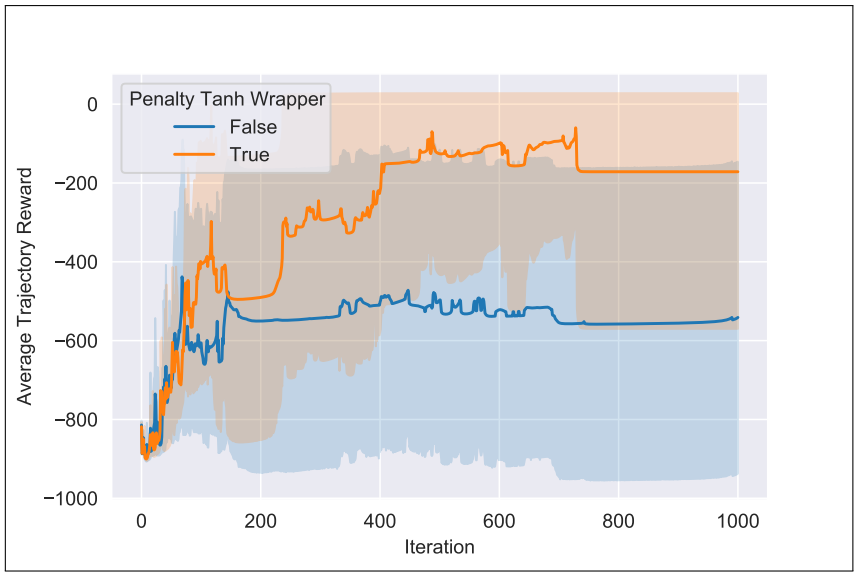
The following result graphs show the mean value and its 95% confidence intervals calculated over 10 trials by the "Seaborn" software library.

The results from both experiments are visualised in Fig. 4.5, one using an early termination method upon meeting a specific condition and the other without such termination, enabling a comparison of the efficacy of the tanh method explained above in each setting, which no significant difference was shown due to the overlap of error bars.

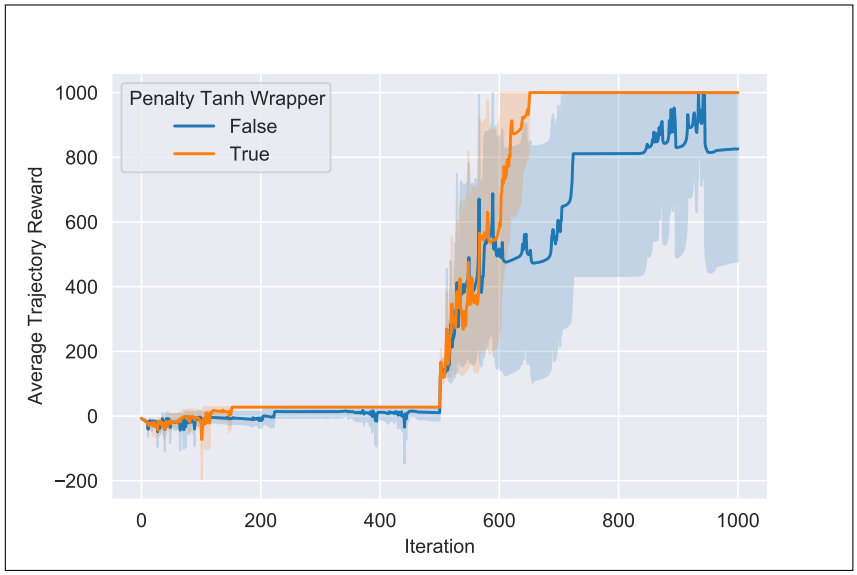
In Fig. 4.5a, the comparison is made regarding the effect of applying the tanh function to the penalty part of the reward function. Notably, the reset condition for the agent's roll orientation was omitted, enabling the agent to achieve a parallel orientation to the ground while retaining its ability to move. On the other hand, Fig. 4.5b presents the impact of employing the tanh function in the penalty section of the reward function while maintaining the reset condition for the agent's roll orientation. The game was reset in this scenario whenever the agent's roll angle ω reached either $\frac{\pi}{9}$ or $-\frac{\pi}{9}$. By examining these two experiments, the influence of the tanh activation function under different conditions can be assessed and gain valuable insights into its performance in the context of the reward function.

From the results in Fig. 4.5a, the trial with the highest performance was selected, and the paths taken by each agent to reach the goal, along with their corresponding roll values, were plotted in Fig. 4.6. Precisely, Fig. 4.6a displays the agent's path over 1000 updates of the physics model, with the axes representing the location. Concurrently, Fig. 4.6b depicts the ω value of the bike on the y-axis, indicating its balance. Moreover, in Fig. 4.6c, the goal is shown with a cross and 30 units away from the starting position of the agents. Fig. 4.6c, presents the agents that used the tanh wrapper around the desired penalties of the reward function, and their respective bike balancing values are shown on the y-axis in Fig. 4.6d. These visual representations enable a comprehensive analysis of the agent's paths and bike balance under the influence of the tanh method described above, providing valuable insights into this method's impact on the agents' performance.

Fig. 4.6 provides a visual representation of the navigation task within the continuous space



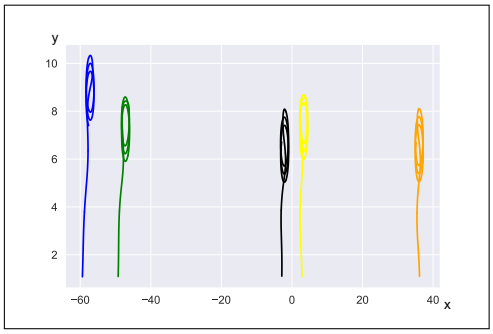
(a) The experiment setup where there are no reset conditions unless the agent reaches 1000 steps.



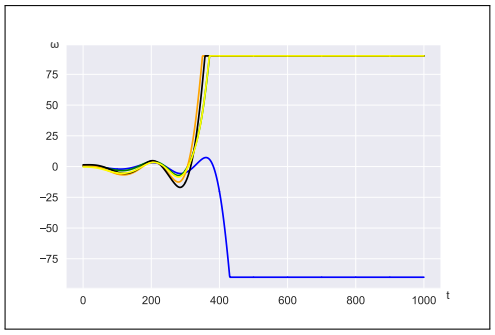
(b) The experiment setup where the game resets when the agent reaches $\frac{\pi}{9}$ on either side.

Figure 4.5: Visualisation of the agent's performance, comparing the tanh wrapper used around the penalty section of the reward function under both reset conditions explained. The results represent the agent's validation setup over 1000 iterations across 10 trials. The difference in the y-axis ranges between the top and bottom diagrams is due to using a tanh wrapper in one setup, which results in smaller penalty values. In contrast, the absence of a tanh wrapper in the other setup allows for larger penalty values to be recorded, thus causing the discrepancy in scale.

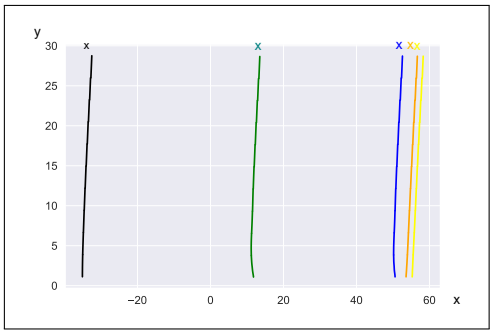
environment, specifically focusing on the bicycle balancing and navigation challenge introduced by Randløv and Alstrøm [1998]. Each coloured line in the figure corresponds to a distinct trajectory taken by an agent instance during the simulation. These trajectories are not merely different paths taken by the same agent in a static environment; instead, they reflect the outcomes of introducing randomness into the task's initial conditions or the environment itself.



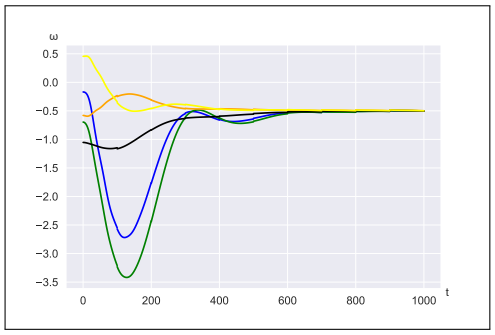
(a) Agent's path without tanh wrapper around the penalty section of the reward function



(b) Agent's roll value when a tanh wrapper is applied around the penalty section of the reward function. The x-axis represents the trajectory time-step t, and a roll value of 0 indicates that the bicycle is standing perpendicularly to the ground. Instances, where the roll value reaches $\omega = \pm 90^{\circ}$ indicate that the bikes are toppling over onto their sides. The use of the tanh wrapper moderates the penalties, affecting the stability and trajectory of the bicycle.



(c) Agent's path when a tanh wrapper is used around the penalty section of the reward function. In this visualisation, each bike is shown to be successfully navigating towards its corresponding goal point, denoted by an 'x' of a matching colour. The tanh wrapper moderates the penalties, guiding the agents more effectively towards their objectives.



(d) Agent's roll value with tanh wrapper around the penalty section of the reward function (0 indicates the bicycle is standing perpendicularly)

Figure 4.6: Visualisation of the agent's performance, comparing the tanh wrapper used around the penalty section of the reward function.

In the context of this experiment, randomisation plays a crucial role in testing the robustness and adaptability of the BPTT algorithm and other RL approaches under evaluation. Randomisation can manifest in various aspects of the task, such as:

- Initial positioning and orientation of the bicycle.
- The positioning of goals within the environment.

By introducing these random elements, each agent instance encounters a slightly different version of the task, compelling it to adapt its learned strategy to navigate the environment and reach its objective successfully. This approach ensures that the evaluation of the algorithms' performance is not limited to a single, potentially idiosyncratic scenario but encompasses a broader range of conditions that an agent might face in real-world applications.

The varied trajectories illustrated in Fig. 4.6 serve as a testament to the agents' ability to generalise their learned behaviour across different instances of the navigation task. The differences in the paths taken, denoted by the distinct colours, underscore the flexibility and dynamic decision-making capacity of the agents trained under the BPTT framework.

This visual evidence supports the conclusion that the algorithms under study can handle the complexities introduced by randomisation, thereby validating their potential for applications in environments where variability and unpredictability are common. Furthermore, analysing these trajectories provides insights into the challenges and opportunities for optimisation that might be addressed in future research to enhance the agents' performance in continuous space navigation tasks.

4.3.5 Discussion

As shown in the experiments, the agent with tanh wrapper around the penalties in the reward function managed to balance the bike and move towards their goal; in comparison, the agent without the tanh wrapper could not recover from the exceeding penalties it received after hitting the ground. In Fig. 4.6d, it can be seen that the bike managed to keep the balance of the bike by slightly moving its balance towards an angle to keep moving forward towards the goal.

The bicycle navigation task employs a polynomial objective function that the learning algorithm aims to maximise. This function is designed to quantify the agent's success in navigating towards a target while maintaining balance. However, as the agent's trajectories extend in length, the task of learning optimal control strategies becomes increasingly complex. This complexity arises from the nature of the cost function gradient and how it influences the learning process in scenarios involving extended trajectories.

In the context of long trajectories, the gradient of the cost function—which indicates the direction and magnitude of the steepest ascent towards the objective maximisation—exhibits pronounced steepness at points that may not be strategically advantageous for learning. Specifically, these points often occur after the agent has already entered an uncontrollable position,

meaning that the trajectory has deviated significantly from an optimal path or that the bicycle has lost its balance to the extent that recovery is unlikely within the current trajectory.

This characteristic of the gradient presents a significant challenge for the learning algorithm for several reasons:

- **Misguided Focus:** The steep gradient at inappropriate points misleads the learning algorithm into prioritising adjustments aimed at "saving" positions that are essentially lost causes. Instead of focusing on improving the overall strategy for earlier parts of the trajectory, the algorithm expends resources attempting to correct situations where recovery is highly improbable.

- **Inefficient Learning:** Concentrating on these lost-cause positions detracts from the learning process's efficiency. The algorithm might become bogged down in trying to optimise for scenarios with minimal potential for impact, thereby slowing overall progress towards learning effective navigation and balance strategies.

- **Confusion and Instability:** The disproportionate emphasis on correcting uncontrollable positions can confuse the learning process, leading to instability. The algorithm might oscillate between strategies without converging on a robust solution that consistently achieves the task's objectives.

To mitigate these issues, several strategies can be considered, including:

- **Regularisation and Reward Shaping:** Modifying the objective function to smooth out the gradient or incorporating reward-shaping techniques to guide the learning process effectively.

- **Early Termination of Trajectories:** Implementing mechanisms to terminate trajectories preemptively when the agent enters an uncontrollable position, preventing the algorithm from focusing excessively on lost causes.

- **Adaptive Learning Rates:** Using adaptive learning rates to dynamically adjust the magnitude of updates based on the trajectory's context helps the algorithm prioritise learning from more informative segments.

In summary, while the polynomial objective function in the bicycle model poses distinct challenges for learning in scenarios involving long trajectories, understanding and addressing

the underlying issues related to the cost function gradient can lead to more effective and efficient learning algorithms. By implementing strategic modifications and enhancements, it is possible to guide the learning process towards achieving optimal navigation and balance in continuous space environments.

For example, by smoothly applying tanh to the polynomial cost function, which enabled the learning algorithm to concentrate on the critical parts of the trajectory, i.e. where the trajectory's performance just started to deteriorate, which can be seen in Fig. 4.6b before the 400 physics model update, as opposed to when trajectory control has already been irreversibly lost, i.e. after the 400 physics model update shown in Fig. 4.6b.

It can be argued that the learning can be stopped after losing control, allowing the weights to be updated correctly. By stopping the trajectories, the agent ignores the gradients after the terminal state; it is believed that these errors caused by the agent are valuable assets for learning. Ignoring such penalties can damage effective learning. In errors, the agent learns; however, the exceedingly large number of penalties generated by the agent after hitting the ground in the experiment shown can be controlled by the proposed tanh function wrapped around the penalties, which allowed the agent to balance effectively and find its way towards the goal.

Therefore, according to Fig. 4.5a, in the case of the bike agent without the early termination condition, where the agent was allowed to reach a parallel orientation to the ground and still be able to move, the tanh wrapper proved to be highly effective. The tanh wrapper facilitated a more gradual and controlled learning process, allowing the penalties to accumulate smoothly. This resulted in improved stability and consistent progress throughout the experiment.

On the other hand, according to Fig. 4.5b, in the bike agent experiment with the early termination condition, the tanh wrapper also played a significant role. By introducing the termination condition when the agent's roll angle exceeded a certain threshold, the tanh wrapper helped stabilise the learning process. The wrapper effectively attenuated the penalties and prevented abrupt resets, allowing the agent to explore and learn in a more controlled manner.

Furthermore, the findings indicate that finding the right balance between rewards and stability is crucial when designing and optimising the BPTT algorithm.

Overall, the experiments with the bike agent underscore the significance of reward-shaping techniques, such as the tanh wrapper technique, in achieving more stable and robust learning outcomes. Further exploration and refinement of such reward-shaping techniques can lead to better performance and convergence in complex navigational tasks.

4.4 Chapter Conclusions

This chapter embarked on an empirical journey to scrutinise the applicability and efficacy of Backpropagation Through Time (BPTT) and other advanced reinforcement learning (RL) algorithms in the complex terrain of continuous space environments. Through methodical experimentation, including a navigational task with a fixed food location as detailed in Section 4.2, this chapter gleaned valuable insights into these algorithms' operational nuances and comparative performance.

The experiments revealed that BPTT stands out for its ability to guide agents through continuous spaces with a degree of efficiency and adaptability that may surpass traditional RL approaches. Specifically, the BPTT algorithm demonstrated promising results in steering agents towards objectives with precision and learning speed that suggest an optimal navigation strategy. This outcome validates BPTT's potential in continuous environments and underscores its relevance to the thesis's focus on enhancing RL agents' adaptability and decision-making prowess in dynamic settings.

In contrast, while other RL algorithms showed potential, they exhibited limitations such as slower learning rates, occasional instabilities, and sometimes lower overall rewards. These findings indicate a potential area for further research and development, emphasising the need to refine these algorithms to improve their performance in continuous space navigation tasks.

Using neural architectures, characterised by their simplicity yet effectiveness, further illustrated the dynamic interplay between algorithmic strategies and environmental complexity. The implementation details, such as choosing a hidden layer with 20 nodes and the tanh activation function, were pivotal in capturing the environment's subtleties, reinforcing the thesis's theme of integrating deep learning techniques with RL for advanced problem-solving.

In this chapter, it is evident that BPTT, with its distinctive advantages in speed, stability, and reward outcomes, marks a significant stride toward realising more sophisticated and capable RL agents. These insights enrich our understanding of BPTT's role within the broader RL landscape and lay the groundwork for future explorations into more complex environments and challenges.

Looking ahead, the next phase of our research will venture into the uncharted waters of Section 4.2 with randomised food-source locations, aiming to test the limits of BPTT further and explore its synergies with memory mechanisms in navigating the unpredictable terrains of continuous spaces. This future work promises to deepen our comprehension of BPTT and other RL algorithms, contributing to the advancement of artificial intelligence and the development of

autonomous agents capable of sophisticated decision-making and adaptation.

In summary, this chapter not only contributes to the thematic core of the thesis by showcasing BPTT's potential in continuous space navigation but also sets the stage for subsequent investigations into the integration of memory and adaptability in RL. Through this exploration, This thesis moves one step closer to the goal of developing RL agents that are not just reactive but truly adaptive, capable of thriving in the ever-changing landscapes of the real world.

Chapter 5

Adaptive Learning for Resource Exploitation and Navigation in Continuous Environments

The exploration of adaptive learning mechanisms in environments that simulate natural processes represents a significant stride toward understanding and replicating the cognitive abilities of living organisms. This chapter delves into the dynamics of a simulated organism tasked with exploring and exploiting resources within a continuous space. This scenario mimics the fundamental survival strategies living entities employ in natural ecosystems. Unlike the discrete, maze-like environments discussed in previous chapters, this setting presents a more nuanced challenge that closely mirrors real-world conditions, where resources like food are distributed across a continuous landscape.

This chapter's primary contribution lies in examining how adaptive dynamic programming (ADP) algorithms, specifically Backpropagation Through Time (BPTT), can be leveraged to navigate and exploit resources efficiently within such environments. This exploration is motivated by the need to understand the extent to which BPTT and related reinforcement learning (RL) algorithms can adapt to changes in environmental conditions and objectives, reflecting the adaptability inherent in living organisms.

In contrast to the previous focus on memory for navigation within discrete spaces, this chapter broadens the scope to include memory's role in navigation and resource exploitation in continuous environments. This shift is crucial for simulating more complex behaviours observed in nature, where organisms navigate their surroundings and make strategic decisions on resource

allocation and prioritisation based on memory and observation.

The scenario detailed in this chapter situates the agent within a minimally complex, partially observable environment (POE), a term that signifies a fundamental aspect of the challenges faced by the agent. Unlike fully observable environments, where the agent has access to complete and immediate information about its surroundings at all times, a POE restricts access to such comprehensive data. Instead, the agent receives limited sensory inputs that provide incomplete information about the state of the environment. This limitation necessitates the agent to infer or reconstruct the missing information to make informed decisions and navigate the environment effectively.

This concept of partial observability is critical in understanding the dynamics of natural and artificial intelligence systems, as discussed in the Section 2.2.1 on differences in environment observabilities. In real-world scenarios, living organisms rarely have access to all information about their surroundings. Instead, they rely on their sensory perceptions, memory, and cognitive abilities to coherently understand their environment, a process mirrored in this chapter's design of the simulated environment.

In this specific scenario, the agent, modelled as a simple simulated organism, is tasked with locating and moving towards a food source, the position of which is randomised at the start of each new iteration. The agent's sensory inputs are limited to the density of food at its current location, represented as a single scalar value. This setup epitomises a POE, as the agent does not have direct visibility of the entire environment or the exact location of the food source. Instead, it must explore the environment, sampling the food density at multiple locations, and use its internal memory to record these observations. Over time, the agent synthesises this fragmented information to deduce the probable location of the densest food concentration and devise a strategy to exploit this resource efficiently.

The challenge of navigating and exploiting resources in a POE highlights the importance of memory and adaptive learning strategies in overcoming the limitations posed by incomplete information. It also aligns with the broader thesis of enhancing reinforcement learning agents' adaptability and decision-making capabilities in dynamic environments. By investigating how agents learn to operate within a POE, this chapter contributes to the ongoing exploration of advanced learning mechanisms and their application in simulating complex cognitive processes observed in living organisms.

Addressing this problem necessitates using a recurrent neural network (RNN) with recurrent memory nodes. This setup enables the simulated organism to simultaneously explore its envi-

ronment and memorise salient features, facilitating the development of a nuanced movement strategy. In this context, the goal of employing ADP/RL algorithms is to discover an RNN configuration that optimally solves the exploration and exploitation challenge.

Furthermore, this chapter aims to distinguish between navigation and resource exploitation concepts. Navigation refers to the organism's ability to orient itself and move through the environment, while resource exploitation focuses on the organism's efficiency in locating and using resources within that environment. This distinction is vital for understanding the organism's adaptive learning processes in response to environmental changes.

By integrating ADP/RL algorithms with RNNs, this research seeks to solve a specific resource exploitation problem. It contributes to the broader understanding of adaptive learning and memory's role in continuous space environments. This endeavour aligns with the thesis's overarching theme of enhancing artificial agents' adaptability and cognitive capabilities, pushing the boundaries of current machine-learning approaches to better mimic the complex decision-making processes observed in living entities.

This chapter presents a scenario where a simple simulated organism must explore and exploit an environment containing a food pile. Building upon the foundational work discussed in Section 4.2, this chapter extends the investigation into how an organism learns to navigate its environment to seek food effectively. The organism learns to make observations of the environment, use memory to record those observations, and thus plan and navigate to the regions with the strongest food density. In particular, this chapter compares reinforcement-learning algorithms with an adaptive dynamic programming algorithm, specifically the BPTT algorithm, to demonstrate its relative efficiency. With the background of BPTT already established for the readers, the chapter concludes that backpropagation through time can convincingly solve this recurrent neural-network challenge. Furthermore, this algorithm successfully mimics a minimal organism's fundamental objectives and mental environmental-mapping skills while seeking a food pile distributed statically or randomly in an environment.

Existing Reinforcement Learning (RL) and Adaptive Dynamic Programming (ADP) algorithms have become more efficient in examining, learning, and solving new problems (e.g., a new game) [Mnih et al., 2013, Silver et al., 2018]. However, solutions can be computationally demanding as the algorithms slowly adapt to a new problem. The existing RL methods cannot efficiently adapt to an existing problem that has been learnt once but then changes its rules [Li et al., 2018]. This chapter presents a surprisingly simple POE problem where the goal is randomised, and many ADP/RL algorithms struggle to solve it.

Furthermore, in contrast to the maze-solving problem discussed in Chapter 3, where memory was solely used for navigation in a discrete space environment, this chapter explores the use of memory for both navigation and resource exploitation in a continuous space environment.

The proposed problem in this chapter is close-to-minimal in a POE, where an agent has to use observations and internal memory to learn to navigate and move toward a randomly positioned food-source – see Fig. 4.1. The food-source's location is randomised at every "game" reset, where the agent is re-initialised according to its original position. The agent receives sensory data of the density of food at its current location (this is the height of the Gaussian bump, i.e., a single scalar reading) at every time step of its journey.

To solve the proposed problem:

- The agent has to devise a movement strategy that quickly explores, taking samples of the food density at multiple nearby locations.

- It also must devise a memory strategy that quickly records observations to acquire the location information about the food-source. Then, an exploitation strategy quickly moves the agent towards the discovered densest food-location.

This solution requires a recurrent neural network (RNN) with recurrent memory nodes to act as an exploration and memorisation strategy. It also requires an ADP/RL algorithm to discover this RNN exploration algorithm.

In the context of this chapter, navigation and resource exploitation are distinct yet interrelated tasks that the agent must perform within a POE. Navigation refers to the agent's ability to orient itself and move through the environment towards a goal or area of interest. It involves understanding the spatial layout and deciding the best paths to reach a destination efficiently. On the other hand, resource exploitation focuses on the agent's ability to identify, evaluate, and use resources within the environment to its advantage. In the scenario presented, the primary resource of interest is the food pile, and exploitation involves not just locating this food source but also determining the most efficient way to consume or benefit from it.

This distinction is critical in understanding the agent's dual objectives: first, to navigate or find its way to the food source, and second, to exploit or maximise the gain from this resource. The complexity arises from the environment's partial observability, which demands that the agent develop a strategy to infer the location of the food source based on limited sensory inputs and then apply a separate set of strategies to optimise resource consumption.

Removing the analogy to poker, the concept of a POE can still be well understood without it. Such environments necessitate that the agent relies on indirect cues and memory to reconstruct the state of the world it cannot directly observe, a challenge that is central to the tasks of navigation and resource exploitation described.

Regarding Recurrent Neural Networks (RNNs), it is acknowledged within the machine learning and neuroscience communities that RNNs offer a powerful framework for modelling sequential data and temporal dependencies, attributes theoretically aligned with certain cognitive processes. The capacity of RNNs for universal computation, akin to Turing completeness, suggests that with adequate complexity and tuning, RNN architectures could potentially solve a wide range of problems, including those requiring the integration of observations over time to make informed decisions. This theoretical underpinning forms the basis for employing RNNs to address the challenge of navigating and exploiting resources in a dynamic POE. By exploring the weight-space of an appropriately sized RNN, the aim is to discover an algorithmic solution that enables the simulated agent to effectively navigate to and exploit a randomly positioned food source, embodying a form of learning that closely mirrors adaptive behaviour in uncertain contexts.

The RNN algorithm will require the formation of memories about the partially-explored environment and goal. Hence, the RNN algorithm must be capable of learning about its environment. It is the objective of the ADP/RL algorithm to find such a set of weights. If the ADP/RL algorithm manages this, it must have "learned how to learn" or performed meta-learning.

Hence, two levels of learning are required here: the inner learning algorithm executed by the RNN as it explores the food environment and the outer learning algorithm, i.e., the ADP/RL algorithm, which discovers an RNN exploration algorithm capable of solving this task.

Even though this food-exploration challenge is relatively easy and could easily be solved by many pre-existing hill-climbing algorithms from computer science, the RNN algorithm sought after in this work must be discovered by the ADP/RL algorithm instead of being hand-programmed. Furthermore, the RNN algorithm must be self-adaptive to any new randomised location for the food pile without requiring further tuning of its weights.

RL algorithms applicable to continuous action and state spaces were chosen from the Stable-Baselines package (version 1.1.0) [Raffin et al., 2019] to tackle the POE. These algorithms included Advantage Actor-Critic (A2C), Soft Actor-Critic (SAC), Deep Deterministic Policy Gradient (DDPG), and Twin Delayed DDPG (TD3) [Mnih et al., 2016, Haarnoja et al., 2017, Lillicrap et al., 2015, Fujimoto et al., 2018]. Additionally, the classic ADP model-based

algorithm, backpropagation through time (BPTT) [Werbos, 1990, Lillicrap and Santoro, 2019], was used. However, the experiments revealed that the selected state-of-the-art RL algorithms could not devise a policy that enabled the agent to develop a recurrent-memory-based exploration strategy.

In contrast, the implemented BPTT algorithm successfully devised an exploration strategy with recurrent-memory features. This outcome aligns with prior work by Fairbank et al. [2014a], demonstrating that BPTT could find RNN solutions for exploiting POEs and exhibit more robust convergence compared to classic RL algorithms [Fairbank et al., 2013]. Although BPTT can be argued not to be a "true" RL algorithm as it requires access to a known environment model, it was shown that using BPTT to train the agent was crucial in finding a solution. Furthermore, the advantage of BPTT in receiving model-based gradient information about the environment played a significant role in solving this task.

It is proposed that the navigational problem and the RNN algorithm necessary to solve it emulate, in the simplest sense, the memory-manipulation tasks performed by certain organisms to locate food (e.g., E-coli bacteria's sensory-based navigation [Hu and Tu, 2014]). Therefore, it is argued that this RNN algorithm is functionally equivalent to the food-seeking sentient behaviour of the simplest organisms. It encompasses the formation of memories describing the agent's environment and the goal-directed adaptive behaviour to exploit the environment and obtain food. The term "functionally sentient" refers to the fact that the external behaviours of the agent mimic those of a sentient creature. It is important to note that no claims are made regarding the actual sentience of the simulation or such simple organisms. This approach follows a long tradition in adaptive-behavior research of developing "minimal agents" that exhibit complex behaviour [Conway et al., 1970, Braitenberg, 1986, Beer, 2003].

One of the critical challenges for the agent in this particular task is that the single environment sensor returns only one scalar value. Consequently, several scalar readings are required to deduce the slope of the food gradient, necessitating memory. These readings must then be combined using some algorithm (e.g., triangulation) to deduce the food-density gradient. While a human programmer might need to design a solution algorithm, in this case, the RNN must carefully learn it through self-learning. Moreover, once the problem is solved, the RNN exhibits capabilities commonly associated with simple organisms assumed to be sentient:

1. It maintains an internal belief state about its environment, such as the current estimation of the food gradient and the agent's previous location.

2. The belief state changes as the agent explores the environment.

3. It executes a purposeful movement strategy to gather knowledge through exploration.

4. It performs computations on its memories (e.g., triangulation) to fully exploit the environment.

5. Once the goal is discovered, it moves towards it, demonstrating exploitation behaviour.

Contrasting the above features with the maze Q-learner discussed in Chapter 3 is worth contrasting. While both agents maintain some form of internal belief state and exhibit goal-oriented behaviour, there are several key differences:

1. The maze Q-learner's sensory input is generally more abundant and possibly multidimensional. In contrast, the agent in this chapter must make do with a single scalar sensor for food detection.

2. Memory usage in the maze chapter is primarily geared towards navigating a fixed maze with static rules. In contrast, the agent in this chapter needs to employ memory more dynamically to deduce the varying food-density gradient, which involves complex computations like triangulation.

3. The concept of 'exploitation behaviour' in the maze q-learner is more straightforward: it is consistently followed once the optimal biologically feasible; the agent in this chapter must continually adapt its exploitation strategies based on evolving beliefs about food density and location.

5.1 Environment and Agent Definitions

The environment physics and the reward function were taken from the physics described in Section 4.2. However, the agent definitions were altered to accommodate the RNN structure.

5.1.1 Agent's brain with memory

The observation vector received by the neural network is $\vec{o}_t$ defined by:

$$\vec{o}_t = (x_t, y_t, d(x_t, y_t), \bar{h}_t), \tag{5.1}$$

the $d(x_t, y_t)$ of $\vec{o}_t$ represents a scalar sensor reading the agent receives about the food density at their current location.

The memory state of the agent at time t is given by a vector $\vec{h}_t \in \mathbb{R}^m$, where m is the number of recurrent memory nodes.

The neural network implemented is shown in Fig. 5.1. It has $\dim(x_t, y_t, d(x_t, y_t) + m$ inputs and $\dim(\vec{a}_t) + 2m$ output nodes. This allows for recurrence in the neural network and allows the neural network to receive observations ($\vec{o}_t$) as input and to make a control action ($\vec{a}_t$) as output. At time t, the output vector $\vec{j}_t$ of the neural network is given by:

$$\vec{j}_t = \pi(\vec{h}_t, \vec{o}_t, \vec{w}), \tag{5.2a}$$

where π denotes the neural network, $\vec{o}_t$ is the observation vector at time t, and $\vec{h}_t$ is the recurrent memory state at time t.

The output vector $\vec{j}_t$ is partitioned into chunks defined by:

$$\vec{j}_t = [\vec{a}_t, \vec{h}_t^{\text{input}}, \vec{h}_t^{\text{gate}}], \tag{5.2b}$$

as shown in Fig. 5.1.

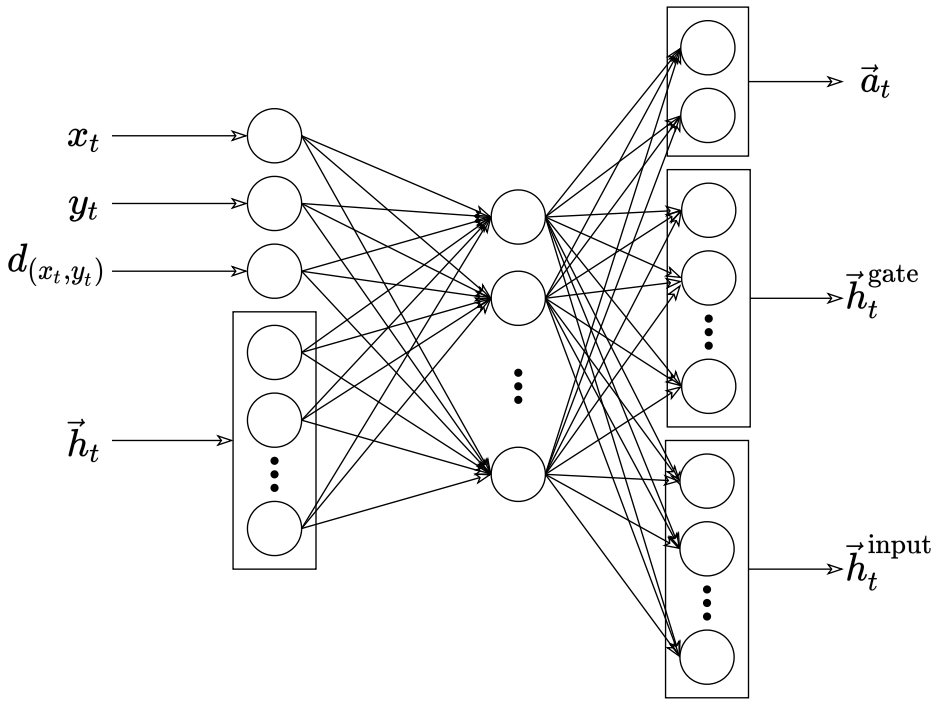


Figure 5.1: Main neural-network structure used in randomised food-location experiments.

In this context, $\vec{a}_t$ represents the action vector (of length 2) chosen by the agent through the deterministic policy function, while $\vec{h}_t^{\text{input}}$ and $\vec{h}_t^{\text{gate}}$ (each of length m) serve as memory gates and inputs for a GRU-style recurrent memory. The agent's motion adheres to the same model

described in Chapter 4, particularly Eq. (4.3), to maintain continuity and facilitate comparison. For the reader's convenience, the equations governing the agent's movement are reiterated below:

$$\vec{v}_t = v_{\max} \tanh(|\vec{a}_t|)\widehat{a}_t. \tag{5.3a}$$

$$\vec{p}_{t+1} = \vec{p}_t + \vec{v}_t \Delta T, \tag{5.3b}$$

where $\vec{p}_t = (x_t, y_t)$ represents the agent's position at a given time-step.

It is also pertinent to revisit Eq. (4.2) from Chapter 4, as this equation forms the basis of the reward function employed in the current chapter:

$$R = \sum_{t=0}^{L-1} \gamma^t r_t, \tag{5.4}$$

The recurrent memory is updated during each time step by the equation:

$$\vec{h}_{t+1} = \tanh(\vec{h}_t^{\text{input}}) \odot \sigma(\vec{h}_t^{\text{gate}}) + \vec{h}_t \odot (1 - \sigma(\vec{h}_t^{\text{gate}})), \tag{5.5}$$

where σ denotes the logistic sigmoid activation function, and $\odot$ denotes element-wise vector multiplication.

To fully apply Eqs. (5.3a) and (5.5), the neural network π must produce unbounded outputs, i.e. have no activation function on its final layer.

The decision to omit "forget" gates from the simplified version of gated memory in this study is grounded in the pursuit of computational efficiency and model simplicity. The original Gated Recurrent Units (GRUs) introduced by Chung et al. [2014] included forget gates as a mechanism to control the extent to which previous states influence current memory content. While this feature allows GRUs to model temporal dependencies with great flexibility, it also introduces additional complexity and computational overhead.

Building on the work of Zhou et al. [2016], this study simplifies GRUs by omitting forget gates, following a trend towards minimalistic neural architectures. The rationale behind this simplification is twofold:

1. **Increased Efficiency:** By reducing the parameters and operations required at each time step, the simplified model can be trained more rapidly. This efficiency is crucial when computational resources are limited, or the model needs to be scaled to handle large datasets or complex environments.

2. **Focus on Essential Dynamics:** Simplifying the memory mechanism allows the model to focus on capturing the most critical temporal relationships without the potential distraction of managing forgetfulness explicitly. This approach assumes that the essential dynamics of the task can be learned and represented even without the ability to forget past information selectively.

This simplified GRU model is inspired by the minimalist GRU variant proposed by Zhou et al. [2016], which demonstrated that GRUs could achieve competitive performance even without "forget" gates. In the context of this study, the simplified memory model is deemed sufficient for exploring the primary research questions related to navigation and resource exploitation in partially observable environments. The model's performance in these tasks provides empirical evidence supporting the viability of this simplification.

Please refer to Appendix 3 for the detailed implementation of the memory model.

In summary, the omission of "forget" gates in this study's gated memory model is a deliberate choice to simplify the architecture and enhance computational efficiency. This decision aligns with a broader research trend towards creating more streamlined and efficient neural network models capable of capturing the essence of complex tasks without undue complexity.

5.1.2 Backpropagation Through Time Algorithm

In contrast to the previous chapter's BPTT (Chapter 4) structure shown in Fig. 2.1, memory is introduced to enhance the model's capabilities. The enhanced structure involves the integration of both the physics model and the memory model, as illustrated in Fig. 5.2. In Figure 5.2, it elaborates on the enhanced structure that integrates the physics and memory models, enabling a more sophisticated approach to problem-solving in reinforcement learning scenarios. This integration facilitates a dynamic interaction between the agent's brain and its environment, allowing for the processing and retention of information over time. Key components illustrated in the figure include the agent's brain, which receives inputs from both the environment (observations) and its internal memory state. These inputs are processed to generate actions, update the memory state, and control the flow of information within the memory model. This architecture empowers the agent to make informed decisions based on both current environmental cues and past experiences, showcasing the utility of incorporating memory models in continuous space navigation tasks. The detailed description of this interaction and its implications for reinforcement learning strategies are further elaborated in the text. The Agent's brain receives two crucial pieces of

information: observations ($\vec{o}_t$) from the environment and memory information ($\vec{h}_t$) from the memory model. These inputs are combined and processed, resulting in the generation of three distinct sets of information: actions ($\vec{a}_t$), new memory information ($\vec{h}_t^{input}$), and memory gating information ($\vec{h}_t^{gate}$).

The action set ($\vec{a}_t$) represents the output of the Agent's brain, which will be executed in the environment. The new memory information ($\vec{h}_t^{input}$) will be fed back into the memory model to update and maintain the memory state. Lastly, the memory gating information ($\vec{h}_t^{gate}$) is used to control the flow of information within the memory model.

Combining the physics and memory models empowers the Agent to make more informed and contextually aware decisions. This enables the model to handle complex tasks and address previously unattainable challenges with a standard BPTT approach.

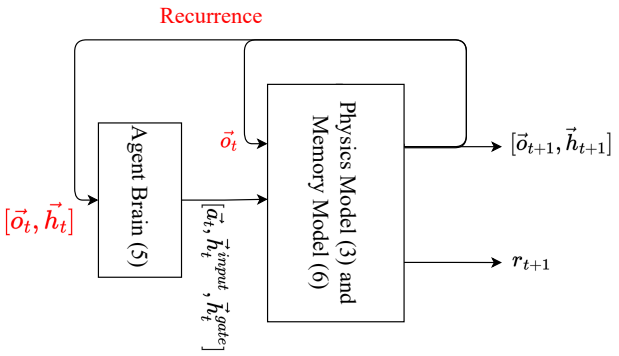


Figure 5.2: Recurrence between Agent Brain and Physics model allows the Agent's brain to produce new recurrent-memory data from the previous observations received by the Physics model.

In contrast to Fig. 2.2, internally, this unrolls the combined network of Fig. 2.1 "through time" to obtain the unrolled network shown in Fig. 5.3.

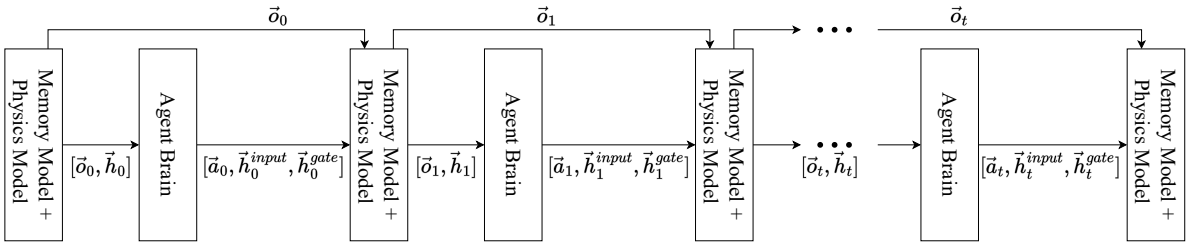


Figure 5.3: Unrolled combined network in BPTT.

BPTT will train the neural network to solve the exploration problem and use the recurrent memory nodes to execute the exploration algorithm necessary to find and exploit the food-pile.

Interestingly, even though knowledge of the food-pile distribution function Eq. (4.1) and physics model Eq. (4.3) is needed to be known during training (by the BPTT algorithm, not by the agent), once the agent has "learned how to learn", access to full knowledge of Eq. (4.1) and Eq. (4.3) is not needed. After training, the agent's RNN samples exploratory scalar values of the function Eq. (4.3) and then makes decisions based on those samples. It no longer gets or needs access to the derivatives of this function. In this sense, once trained, the "inner" component of the meta-learning system (i.e. the RNN) has learned to perform true model-free RL on the environment (albeit only on this specific food-pile environment). Once trained, and when the RNN is unleashed on the environment, it performs model-free RL to locate and exploit the peak of the food pile.

Under this scheme, the "environment" is treated as a more complicated function, which takes extra inputs $\vec{h}_t^{\text{gate}}$ and $\vec{h}_t^{\text{input}}$ (in addition to the usual state and action inputs) and emits extra output information $\vec{h}_{t+1}$ (in addition to the usual next state and reward, information). The neural network is considered feed-forward (Fig. 5.1).

While this change of view makes no functional difference to the way Eqs. (4.3b) and (5.5) behave, from an implementation point of view, it can be useful to put Eq. (5.5) into the environment because it then enables learning algorithms that were not written with recurrent neural networks in mind to be applied to the environment and to be able to instil the agent with the capabilities of gated recurrent memory.

In this chapter, the Partially Observable Environment (POE) primarily influences the agent's observability, not the BPTT algorithm's operation. Throughout the validation phase, the agent remains unaware of the underlying physics model and food-pile distribution; it only accesses its positional values and sensory data, which provide the food's current density at the agent's location. This setup underscores the agent's reliance on immediate sensory feedback to navigate and exploit resources within the environment. It showcases its ability to perform model-free reinforcement learning post-training, even without comprehensive knowledge of the environment's dynamics.

5.2 Experiment

Five selected RL and ADP algorithms were chosen for this experiment. A POE problem was designed to demonstrate these algorithms, where the location and height of the food source were randomised, and the agent must make exploratory observations to find the food. The general experiment setup was designed to match the experiment done in Chapter 4 and in Section 4.2.4.

The following result graphs show the mean value and its 95% confidence intervals calculated over 20 trials by the "Seaborn" software library.

The food location was re-randomised at the start of each episode, and the initial agent positions were always started from $(0,0)$. This is the POE version of the problem, which is necessary for sensing and memory capabilities. Each time an agent starts, the food-pile location was chosen with uniform random distribution such that $x_{\text{food}} \in [-5,5]$, $y_{\text{food}} \in [-5,5]$, $z_{\text{food}} \in [0.5,1.5]$ and $\sigma_{food} = 8$. In addition, the height of the food-source (z_{food}) was randomised within that range to apply further difficulty in this POE experiment.

The neural network used in this experiment is the same as in the previous experiment, except that there are now $m = 20$ recurrent nodes Eq. (5.2a) and also the sensor input $d(x_t, y_t)$ is used in Eq. (5.1).

In the context of this research, the decision to fix the memory size at 20 nodes was based on empirical evidence suggesting this was the minimum effective size observed during preliminary tests. This size balanced computational efficiency and the network's ability to capture and use relevant environmental information. Limited time and resources prevented extensive hyperparameter tuning, leading to the prioritisation of configurations that showed promise in early experiments. This approach is common in research where exploratory findings guide the selection of model parameters, aiming for a feasible balance between performance and resource constraints.

Compared to the previous experiment, this changed the actor-network to have an extra 20 input nodes for $\vec{h}_t$ and an extra node holding the sensory input ($d(x_t, y_t)$). The actor-network output was extended by 20 nodes for $\vec{h}_t^{\text{gate}}$ and a further 20 nodes for $\vec{h}_t^{\text{input}}$, compared to the previous experiment. See Fig. 5.1. 20 nodes also extended the critic network's input for $\vec{h}_t$ and the extra input node for $d(x_t, y_t)$, and the critic output remained a scalar for holding Q-values.

5.3 Results

The results are shown in Fig. 5.4 for this POE experiment. The selected RL algorithms struggled to perform well and devise a navigational strategy to find the peak of the food-pile.

Given the nature of the Gaussian distribution used in the environment, which inherently lacks discrete local or global optima due to its continuous and smooth profile, the observation that agents did not explore effectively and seemed to remain within a specific area may indicate limitations in their exploratory strategy rather than getting "stuck" in a local maximum. This

could suggest that the agents' exploration mechanisms were insufficient to motivate them to sample more broadly across the environment, potentially due to an over-reliance on areas of high immediate reward or a lack of effective exploration incentives in the algorithm's design.

However, the BPTT algorithm reached around 22.5 maximum average rewards, indicating that the BPTT agents solved the problem.

The performance of the BPTT agent in the current experiment, attaining a score close to the maximum of 25.5, is notably comparable to the results of the simpler experiment discussed in Section 4.2. This high score suggests that the BPTT agent operates near-optimally even in a POE. This inference is strengthened by a symmetry observed between the current and previous experiments, as evidenced by the reversed arrows in Figs. 4.4 and 5.5a. The symmetry suggests that the theoretical maximum reward should be analogous in both scenarios. However, it is important to note a slight decrease in performance (i.e., $22.5 < 25.5$). This reduction can be attributed to the time the agent in the more complex POE initially spends on exploration before efficiently navigating to the peak of the food pile.

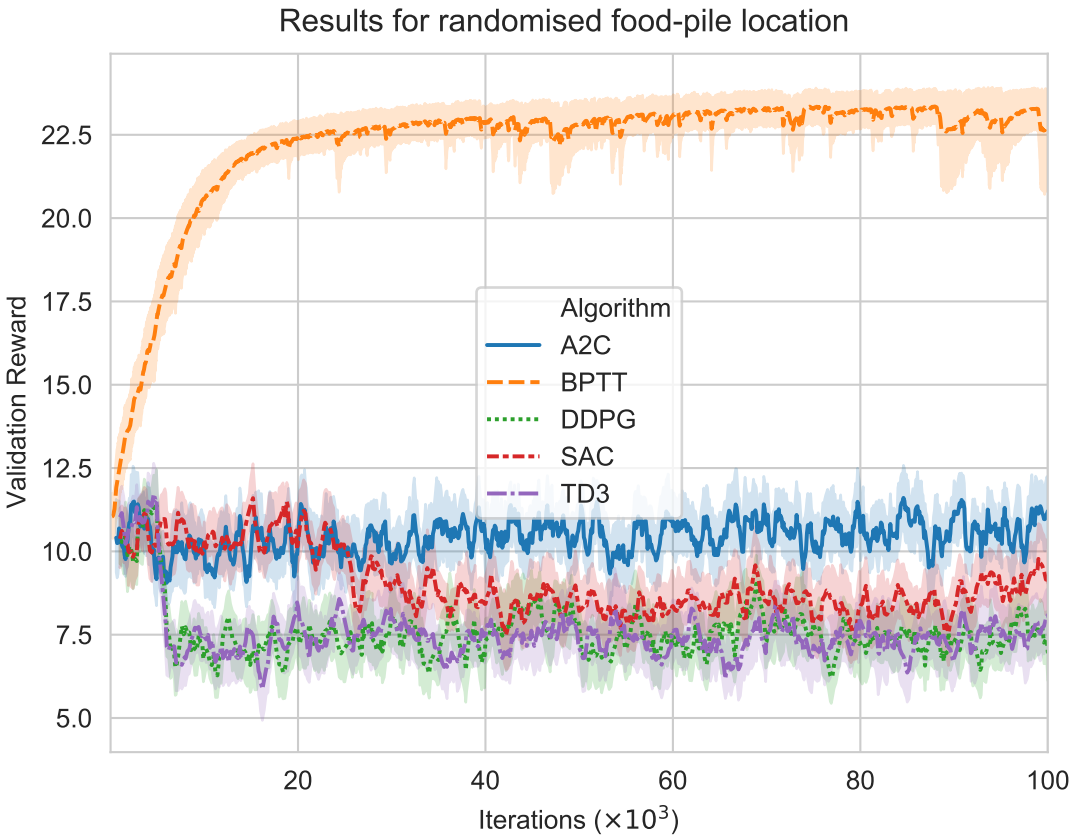
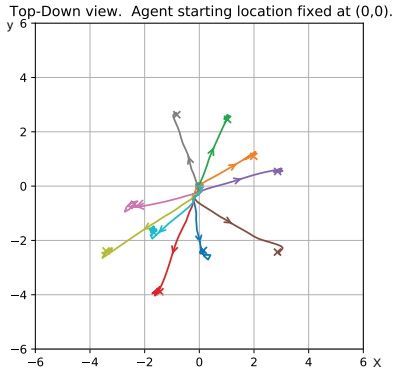


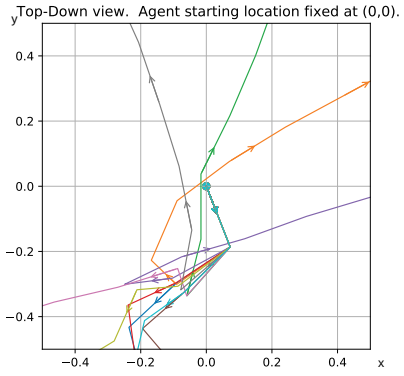
Figure 5.4: Algorithms' performance on randomised food location validation environment over 100,000 iterations and averaged over 20 trials. Each algorithm used sensory data input and 20 recurrent nodes.

Fig. 5.5a shows a sample of ten trajectories from the fully-trained BPTT agent's behavioural movement. Fig. 5.5b shows a close-up zoom of the central region, highlighting the agents performing an exploration strategy to reach the goal.

Fig. 5.5b shows that the RNN exploration algorithm learned to navigate and devise a movement strategy using input sensory data and memory. It illustrates that the ten sample trajectories created by the BPTT agents started from $(0,0)$, with a break-off point around $(0.1,-0.19)$. From that separation point, it showed that the RNN's memories of previous observations affect the agents' decisions after sampling multiple sensor results.



(a) High-level view of a sample of ten randomised food-pile environments.



(b) A close-up zoomed view of the region from x-range $[-0.5,0.5]$ and y-range $[-0.5,0.5]$, showing the initial exploratory actions the agents took in Fig. 5.5a.

Figure 5.5: Behaviour of fully-trained BPTT agents at solving the randomised food-pile problem on a test set. The x and y axes describe the location. Each coloured pathway represents a trajectory from the common start point at $(0,0)$. These show the agents exploring and calculating the direction of increasing food density and then travelling to the food-pile peaks. Each different coloured trajectory ends up at or near the centre of its own specific food-pile location (indicated by the coloured X symbols).

An ablation study was performed using three different versions of the neural-network architecture to clarify that recurrent memories and a sensor were required for the BPTT algorithm to solve the randomised food-location environment. These different versions involved removing the sensor and the recurrent memory. The results are shown in Fig. 5.6 and conclude that the recurrent memory nodes and sensor input are necessary for solving the navigational task.

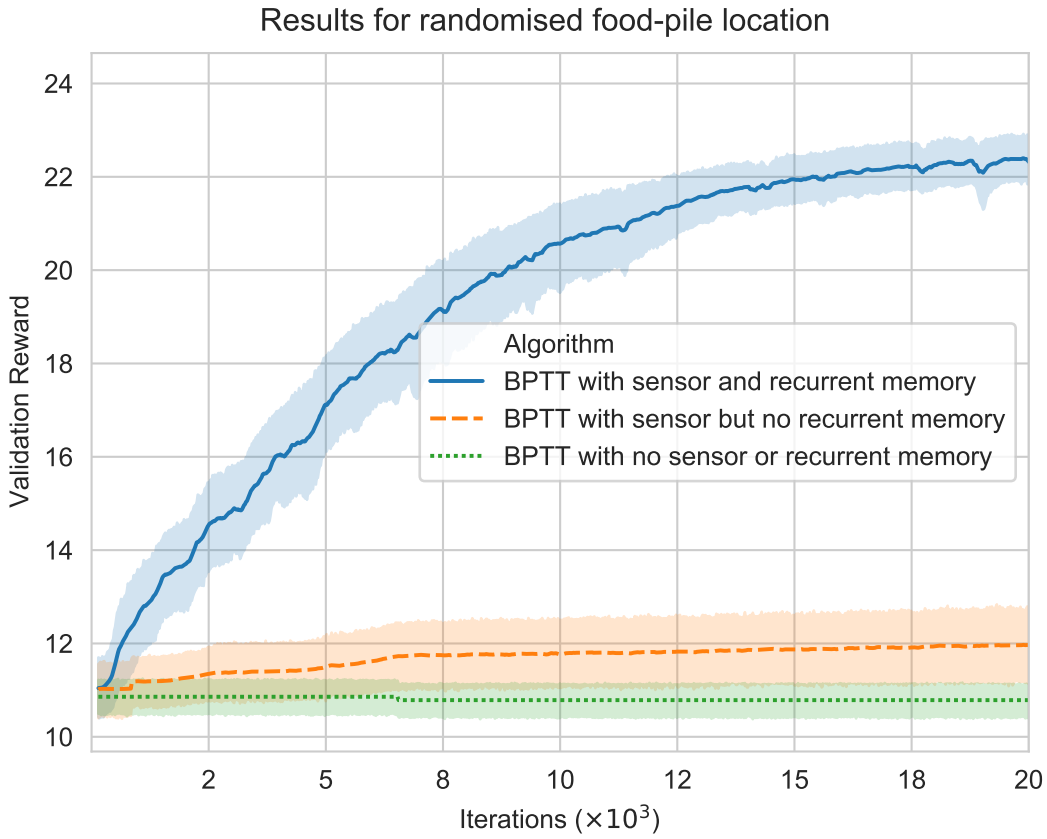


Figure 5.6: The effect of solving the randomised food-pile location problem with and without sensors and memory is that each algorithm setup is experimented with over 10,000 iterations and averaged over 20 different trials.

5.4 Discussion

BPTT showed competency in performance and stability in the POE. The BPTT algorithm discovered an exploration method that looks like the agent is wobbling as it moves towards the goal. This wobble at the start can be assumed to be the agent taking exploratory actions, i.e., it has successfully learned how to learn. The other RL algorithms failed to solve the randomised food-pile problem.

It appears in Fig. 5.4 that the agents receive an average validation reward of approximately 10 at the start of training. This is explained because, for an average randomisation of the initial agent and food positions, the agents were to stand still for 30 time-steps. Then, they would receive an average total reward of 10.7 (a value was found in a separate informal experiment).

Fig. 5.4 shows that the reward progression does not consistently show monotonic improvement, especially with the RL algorithms. This is likely because the RL algorithms are not proven to be true gradient ascent on any objective function, in contrast to BPTT [Barnard, 1993, Fairbank et al., 2013].

It is important to clarify the conceptual distinction between RL algorithms and gradient ascent methods like BPTT. RL algorithms, including policy gradient methods, optimise policies based on expected returns. Still, their updates do not necessarily follow the gradient of a well-defined, global objective function in a strict mathematical sense. This can lead to non-monotonic reward progressions due to exploration, variance in estimates, and the complex dynamics of the environment.

BPTT, by contrast, explicitly computes gradients of an objective function concerning model parameters over sequences, enabling more direct optimisation. This method is grounded in gradient ascent on a defined loss function, providing a clearer path to monotonic improvement in performance, assuming the gradient estimates are accurate and the learning rate is appropriately tuned.

The observation in the chapter aims to highlight these fundamental differences in how RL algorithms and BPTT approach optimisation, suggesting why RL algorithms might not exhibit consistent, monotonic improvement in rewards. It's not a matter of proving RL algorithms are not gradient ascent methods; rather, it recognises their operational and theoretical distinctions, which influence their behaviour and performance in learning tasks.

Fig. 5.4 shows that the selected RL algorithms failed to devise any movement strategy that improved on the initial average reward of 10.7. Their agents got stuck in the same region and did not successfully apply exploration strategies.

This might be because the recurrent memory and the sensory input features uniquely benefited the BPTT algorithm. However, the selected RL algorithms are sensitive to hyper-parameters chosen to solve RL-based problems.

The performance of RL algorithms in complex environments is highly sensitive to the choice of hyperparameters. In this study, while the SAC algorithm exhibited slow divergence in previous experiments, the decision to use a specific set of hyperparameters was informed by a balance

between computational feasibility and the pursuit of optimal performance within the constraints of the study. This approach was guided by preliminary experiments and literature precedents, aiming to establish a baseline for comparison. Recognising the limitations of this approach, future work could entail a more exhaustive hyperparameter search, leveraging techniques like grid search or Bayesian optimisation to explore the hyperparameter space and potentially enhance algorithm performance systematically.

One interesting trajectory in Fig. 5.5b is the purple line, which shows a complete turnaround at approximately $(-0.25, -0.3)$ as the agent "realises" it set off in the wrong direction. This systematic exploration method and multiple sampling of the food density from different locations allow the RNN to deduce the direction in which the food density increases the most. The agent (RNN) learns during an episode (the "inner" learning algorithm).

The pathways over the food-hill taken by the BPTT agents indicate that the agents' RNNs have discovered an exploration algorithm that samples and records food-heights in the environment and acts accordingly. Furthermore, the BPTT algorithm (the "outer algorithm") has chosen weights that enabled the RNN to behave like this. This adaptive behaviour is consistent with the minimal simulated "organism" approach favoured in this work.

In this chapter, the term "partially observable environment" (POE) emphasises the agent's limited access to the complete state of the environment. Initially, during the training phase, the Backpropagation Through Time (BPTT) algorithm requires knowledge of the food-pile distribution function and the physics model to guide the learning process. However, post-training, the agent operates without explicit knowledge of these underlying functions. It relies solely on its positional information and sensory data reflecting the immediate food density. This operational mode underscores the agent's transition to a model-free reinforcement learning approach, where decisions are made based on partial observations rather than a full understanding of the environment's state. Consequently, even though the BPTT algorithm uses comprehensive environmental models during training, the agent's subsequent independent operation aligns with the characteristics of a POE as it navigates and makes decisions based on limited information.

To solve the POE experiment, the ablation study in Fig. 5.6 shows that the implemented BPTT algorithm requires recurrent memory combined with sensory-based information. The other two variants of the implemented BPTT (BPTT with no sensor or recurrent memory and BPTT with sensor and no recurrent memory) failed to solve the problem.

In the POE experiment, all algorithms were forced to obey the recurrent-memory Eq. (5.5) by embedding those equations within the environment model. Although this combined system

(of physics plus memory) is a valid "environment", it turned out to be a particularly challenging one that the RL algorithms could not cope with.

In contrast, the BPTT algorithm exploits knowledge of the true derivatives which pass through the physics environment, through the memory model, and the neural network, i.e. back-propagating gradients right through the unrolled network shown in Fig. 5.3; and these derivatives seem to have been crucial in correctly solving this exploration problem. It seems a reasonable explanation that gradient-based algorithms such as BPTT can potentially extract more information quickly from an environment than scalar-based model-free RL methods [Fairbank and Alonso, 2012].

5.5 Chapter Conclusions

This chapter has provided significant insights into the challenges and opportunities presented by partially observable environments (POEs) and unexpected perturbations, such as the randomisation of food locations. By implementing a simplified form of the Gated Recurrent Unit (GRU), enhanced with sensory inputs, this study demonstrates the use and robustness of Back-propagation Through Time (BPTT) in navigating complex, dynamic environments. Despite its inception in 1990, this classic algorithm has showcased its potential to surpass some of the latest advancements in reinforcement learning algorithms by addressing tasks that necessitate memory manipulation and algorithmic discovery.

The core contribution of this chapter lies in its exploration of BPTT's application within simple, biologically plausible exploration scenarios. By simulating an "organism" tasked with food gathering in a POE, this work aligns with the principles of meta-learning and accommodative neural networks, underscoring the adaptability of BPTT in neural control, especially in tasks where observational signals are scarce. The efficacy of BPTT in managing the exploration-exploitation balance and its capacity to refine policies through reward-based updates highlights its suitability for complex navigational challenges.

Moreover, the chapter's ablation analysis and evaluation of trajectory behaviours emphasise the critical role of memory and sensory observations in task resolution, showcasing the sentient-creature-like attributes of the agent. The successful application of BPTT, yielding a functional RNN capable of self-learning solutions, illustrates the method's potential for broad applicability across diverse tasks without substantial modifications.

This study also differentiates BPTT, classified as an Adaptive Dynamic Programming (ADP)

algorithm, from other RL algorithms using the physics model and its derivatives for accelerated learning. This distinction underscores the advantages of integrating the learning model closely with the physical dynamics of the environment, as opposed to the more isolated approach seen in traditional RL methods.

Looking forward, this research opens avenues for further exploration into more intricate food distribution patterns, increased levels of partial observability, and the potential augmentation of the agent's sensory capabilities. The limited success of the evaluated RL algorithms suggests a need for an in-depth examination of advanced RL strategies, particularly those that could leverage gradient-based learning through recurrent memory nodes.

The next chapter will delve into alternative memory mechanisms, including the full Long Short-Term Memory (LSTM) and Content-Adaptive Recurrent Units (CARU), to further align with the overarching goals of enhancing adaptability, efficiency, and decision-making capabilities in dynamic environments. This direction not only supports the main objectives of the PhD thesis but also contributes to the broader field of artificial intelligence by advancing our understanding of memory integration in reinforcement learning.

Chapter 6

Advanced Memory Models for Control

This chapter represents a pivotal advancement in exploring memory models for enhancing the performance of Recurrent Neural Networks (RNNs) in control applications within POEs. Building upon the foundational work presented in the previous chapter on the simplified form of the Gated Recurrent Unit (GRU) [Chung et al., 2014], this investigation broadens the scope to include a suite of advanced memory models. Specifically, the analysis encompasses the full Long Short-Term Memory (LSTM) [Hochreiter and Schmidhuber, 1997], the Context-Adaptive RNN unit (CARU) [Chan et al., 2020a], and the Minimal Gated Unit (MGU) [Zhou et al., 2016]. Additionally, a modified version of a recurrent network with no gating mechanism, originally proposed by Jordan [1997], termed "*Identity*" in this context, is examined for its potential in neuro-control applications, particularly noting the replacement of the last activation function with a tanh function.

The main contribution of this chapter lies in its comprehensive comparative analysis of these advanced memory models, all implemented alongside Backpropagation Through Time (BPTT), to evaluate their performance and suitability for tackling the complex control tasks encountered in partially observable environments.

Motivated by the necessity to improve RNN performance in dynamic control scenarios, this chapter seeks to elucidate the comparative advantages and limitations of employing various memory mechanisms in conjunction with BPTT. The objective is to discern how these advanced models can enhance agents' adaptability, efficiency, and decision-making capabilities, directly aligning with the thesis's overarching goals. By delving into the nuances of memory model integration, this research endeavours to push the boundaries of what is achievable in reinforcement learning and adaptive dynamic programming, especially in contexts that demand a nuanced

understanding of temporal dependencies and environmental dynamics.

Each algorithm was implemented and evaluated in the context of the "EnvironmentAntfood" task, as shown in Appendix 3, providing a practical framework for comparison and analysis.

By examining these memory models' performance in a controlled, partially observable task, this chapter contributes significantly to our understanding of the role and optimisation of memory in BPTT-enhanced RNNs. The insights garnered from this study advance the theoretical underpinnings of memory model application in RNNs and offer actionable recommendations for their deployment in more sophisticated navigation and control tasks. The findings herein serve as a valuable resource for future research aimed at refining and expanding the capabilities of intelligent agents within the realms of reinforcement learning and adaptive dynamic programming.

6.1 Memory Integration

This section evaluates four distinct memory models for RNNs, chosen for their relevance to the research questions and their compatibility with the system's requirements, particularly focusing on differentiability and system compatibility. These models are detailed in Section 2.4.8 and are selected based on their potential to enhance the performance of RNNs in handling complex tasks within partially observable environments.

Jordan Recurrent Network and "Identity" Model: The first memory model revisits the classic "Jordan recurrent network" [Jordan, 1997], known for its pioneering role in storing and processing sequences. The adaptation termed the "*Identity*" model in this chapter introduces a significant modification: replacing the activation function in the last hidden layer. Unlike traditional RNNs, the "Jordan recurrent network" and the "*Identity*" model lack a gating mechanism, presenting a simpler architecture. This simplicity is intentional, aiming to assess the impact of minimal structural changes on system performance. The choice of these models allows for an examination of how basic recurrent architectures perform in tasks requiring memory without the complexity of gating mechanisms, emphasising differentiability and straightforward integration with our system.

Long Short-Term Memory (LSTM): The LSTM model [Hochreiter and Schmidhuber, 1997] is selected for its advanced architecture featuring memory blocks and three distinct gating mechanisms: input, forget, and output gates. These features enable selective information retention and processing, making LSTMs highly effective in applications with long-term dependencies.

The inclusion of LSTM in this comparison is based on its proven track record in complex tasks across various domains, from speech recognition to video analysis. Its differentiability and compatibility with backpropagation through time (BPTT) make it an ideal candidate for exploring the effectiveness of gated memory models within our system.

Context-Adaptive RNN Unit (CARU): CARU [Chan et al., 2020a] extends the basic RNN structure by introducing a context gate, which dynamically adjusts the flow of information based on the relevance of current inputs and previous states. This model is chosen for its ability to fine-tune attention to pertinent information, potentially enhancing performance in environments with fluctuating relevance of inputs. CARU's design aligns with the thesis's focus on adaptability and efficiency in dynamic settings, supported by its differentiability and the added value of context-sensitive processing.

Minimal Gated Unit (MGU): Lastly, the modified MGU [Zhou et al., 2016] is included for its streamlined approach, combining efficiency with effectiveness. MGU balances simplicity and computational power with only one gate, requiring less training data and simplifying architecture tuning. This model's selection is motivated by its minimalistic design, which maintains performance with fewer parameters, making it an intriguing option for comparison in terms of both differentiability and system adaptability.

By integrating and comparing these models, this research aims to identify optimal memory mechanisms for enhancing RNNs in control tasks in Partially observable and dynamic environments. Each model's selection is justified by its potential to contribute uniquely to understanding memory integration in RNNs, emphasising their differentiability and compatibility with the system's architecture and learning algorithms.

6.2 Integrating Memory Gates into the Physics Model

In training recurrent neural networks using backpropagation through time (BPTT), it is crucial to employ differentiable memory models. Memory models, such as the full LSTM, Minimal GRU, "*Identity*" or CRU models, possess a known and inherently differentiable structure. Specifically, the memory gates within these models were designed to facilitate differentiability.

The memory models mentioned were designed for static time sequences, and to apply to control algorithms, these memory models can be integrated into a physics model (please refer to Fig. 5.2 in the Section 5.1.2).

The general memory design in this chapter is organised as follows: the memory state of the agent at each time-step is represented by a vector $\vec{h}_t \in \mathbb{R}^m$, where m denotes the number of recurrent memory nodes used by the model. The neural network architecture employed takes an input vector of dimension $\dim(x_t, y_t, d(x_t, y_t) + m$ and produces an output vector of size $\dim(\vec{a}_t) + km$ (where action vector $\vec{a}$ in this context is the two-dimensional control action vector representing the direction the agent wants to move and the position of the agent is updated according to Eq. (5.3)). The value of k varies depending on the memory algorithm used, and it determines the number of additional output nodes dedicated to the memory nodes. Specifically, the architecture can be formalised as follows:

$$\text{Output Dimension} = \dim(\vec{a}_t) + km. \tag{6.1}$$

Compatible "*Identity*": In the "*Identity*" memory algorithm, the value of k from Eq. (6.1) is set to 1, resulting in calculating the hidden state h_{t+1} using the hyperbolic tangent function applied to the input vector $\vec{h}_t^{\text{input}}$. The resulting equation is as follows:

$$h_{t+1} = \tanh(\vec{h}_t^{\text{input}}). \tag{6.2}$$

Compatible Minimal GRU: The Minimal GRU is similar to the GRU, with only one gate controlling the flow of the information. The neural network outputs a gating value $\vec{h}_t^{\text{gate}}$ and an information value $\vec{h}_t^{\text{input}}$; in this memory model, the $k = 2$ is set in Eq. (6.1).

Similar to Zhou et al. [2016]'s MGU design, the simplified version is represented in Eq. (5.5).

Compatible Full-LSTM: The Full-LSTM memory model is already differentiable and can be directly used with BPTT. However, some modifications are required to use the LSTM memory model in conjunction with the physics model in this study.

In Full-LSTM, the output vector $\vec{j}_t$ is partitioned into chunks defined by $k = 4$, resulting in:

$$\vec{j}_t = [\vec{a}_t, \vec{h}_t^{\text{input}}, \vec{h}_t^{\text{gate}}, \vec{f}_t^{gate}, \vec{y}_t^{gate}]. \tag{6.3}$$

To incorporate the LSTM memory model's recurrent memory, memory gates and inputs are denoted by $\vec{h}_t^{\text{input}}$, $\vec{h}_t^{\text{gate}}$, $\vec{f}_t^{gate}$, and $\vec{y}_t^{gate}$, each of length m. These elements control the flow of information in and out of the memory cells, allowing the agent to retain and update information over time.

In an LSTM neural network, the cell state C_t and hidden state h_t are updated at each time step based on the current input, the previous cell state, and the gating mechanism of the LSTM. Specifically, the input gate i_t controls how much of the new candidate memory $\tilde{C}_t$ should be

added to the cell state, and the forget gate f_t controls how much of the previous cell state C_{t-1} should be retained. The new cell state C_t combines the retained previous cell state and the added recent memory:

$$C_t = f_t C_{t-1} + i_t \tilde{C}_t. \tag{6.4}$$

The output gate o_t is a sigmoid function determining how much of the cell state should be exposed as the output. Specifically, the hidden state h_t is computed as $h_{t+1} = o_t * \tanh(C_t)$, where the hyperbolic tangent function $\tanh(\cdot)$ squeezes the cell state into the range [-1, 1]. The output gate o_t is computed as $o_t = \sigma(W_{xo} h_{t-1} + W_{ho} h_t + b_o)$, where $\sigma(\cdot)$ is the sigmoid function and W_{xo}, W_{ho}, and b_o are the weights and bias for the output gate, respectively.

This study represented the memory state of the agent at each time step with a vector $\vec{h}_t \in \mathbb{R}^m$. The observation state holds the cell state C_t and the hidden state h_t to process the physics model. The LSTM algorithm will be calculated using the following equations:

$$i_t = \sigma(\vec{h}_t^{\text{gate}}), \tag{6.5}$$

$$f_t = \sigma(\vec{f}_t^{gate}), \tag{6.6}$$

$$o_t = \sigma(\vec{y}_t^{gate}), \tag{6.7}$$

$$C_t = f_t C_{t-1} + i_t \tanh(\vec{h}_t^{\text{input}}), \tag{6.8}$$

$$h_{t+1} = o_t \tanh(C_t). \tag{6.9}$$

Compatible CARU: CARU is a type of neural network architecture that uses constrained auto-regressive units to update the recurrent memory during each time step. It uses an update gate u_t to determine how much of the previous hidden state h_{t-1} should be retained and how much current input x_t should be added to compute the new hidden state h_{t+1}. The update gate u_t is calculated as a combination of the previous hidden state and the current input using intermediate variables z_t and r_t.

The partitioned output vector $\vec{j}_t$ in CARU is defined as $[\vec{a}_t, x_t, \tilde{h}_t, u_t]$ ($k = 3$ in Eq. (6.1)), where x_t, $\tilde{h}_t$, and u_t are the intermediate variables corresponding to the input x_t, the candidate hidden state $\tilde{h}_t$, and the update gate u_t in CARU, respectively.

The candidate memory state n_t is computed based on $\tilde{h}_t$ using the hyperbolic tangent activation function during each time step. The update gate l_t is calculated based on x_t and u_t using the sigmoid activation function. The new memory state $\tilde{h}_{t+1}$ is then computed using the CARU update rule, where $(1 - l_t) \odot h_{t-1}$ corresponds to the forgetting mechanism of the memory cell, and $l_t \odot n_t$ corresponds to the update mechanism of the memory cell.

$$n_t = \tanh(\tilde{h}_t), \tag{6.10}$$

$$l_t = \sigma(x_t) \odot \sigma(u_t), \tag{6.11}$$

$$\tilde{h}_{t+1} = 1 - l_t \odot h_{t-1} + l_t \odot n_t, \tag{6.12}$$

where σ denotes the logistic sigmoid activation function, and $\odot$ denotes element-wise vector multiplication. n is the candidate memory state that is computed based on $\tilde{h}_t$ using the hyperbolic tangent activation function. l is the update gate computed based on x_t and u_t using the sigmoid activation function. The new memory state $\tilde{h}_{t+1}$ is computed using the CARU update rule, where $(1 - l_t) \odot h_{t-1}$ corresponds to the forgetting mechanism of the memory cell, and $l_t \odot n_t$ corresponds to the update mechanism of the memory cell.

6.3 Experiment Setup

The preceding chapter employed a modified GRU memory model, using backpropagation through time for control within a partially observable environment. This environment served as a testbed to gauge the memory model's efficacy.

This chapter revisits the same environment as the previous chapter, extending the scope to evaluate various memory models: full long short-term memory (LSTM), "*Identity*", Minimal GRU, and CARU. The experimental setup for this evaluation comprises essential components for data collection and an accompanying analysis and interpretation methodology.

To ensure that model comparisons remain unbiased, identical hyper-parameters were maintained across all memory models. This consistency ensures that observed performance differences arise from the intrinsic capabilities of the models, not from any variations in experimental conditions.

Uniformity was also retained in the environmental conditions across all experiments, paving the way for a fair model comparison. Such a uniform approach ensures that any performance discrepancies between the models are attributable solely to the memory modification methods, cementing the reliability and significance of the results.

The neural network architecture, previously adopted in the prior chapter's experiments, remained unaltered for all memory models in this chapter. This consistency means that any performance differences observed can be traced back exclusively to the memory modifications

applied. However, there are some changes to the number of neural network's inputs and outputs due to the memory model nodes suggested in the above description of each memory model.

Data were collected from the validation environment after each network update. A randomised goal position environment was generated during this phase to showcase the updated network's prowess.

6.4 Results

The following result graphs show the mean value and its 95% confidence intervals calculated over 10 trials by the "Seaborn" software library. The shaded areas in the results show the standard error over the 10 trials.

Fig. 6.1 shows the performance of different memory models implemented. The models were trained and validated over 100,000 times, and the "Validation Reward" offers their performance in the validation phase where a complete randomised goal position was introduced to the agent.

Their performance was compared against the same algorithm with no memory, demonstrating a significant improvement in learning efficiency and adaptability. Algorithms equipped with memory mechanisms could maintain a higher level of performance across various tasks, particularly in environments characterised by dynamic changes and uncertainties. This enhancement was most notable in scenarios requiring recalling past experiences to make informed decisions, where the memory-augmented algorithms outperformed their non-memory counterparts. The inclusion of memory enabled the agents to develop a more sophisticated understanding of the environment, leading to more strategic planning and execution of tasks. Furthermore, the memory models contributed to reducing the time required for the agents to converge on effective strategies, showcasing the critical role of memory in accelerating the learning process and improving the overall robustness of the algorithm.

The Table 6.1 compares the average rewards achieved by different memory types at $10,000$ iterations, averaged over 20 trials. This data is crucial in understanding the impact of integrating various memory architectures within reinforcement learning models, especially in improving performance through enhanced memory capabilities.

Table 6.1: Reward at 10,000 iterations was averaged over 20 trials for each memory type used.

Memory type	CARU	Full-LSTM	MGU	Identity	No memory
Average Reward	19.56	23.88	22.74	17.35	10.76

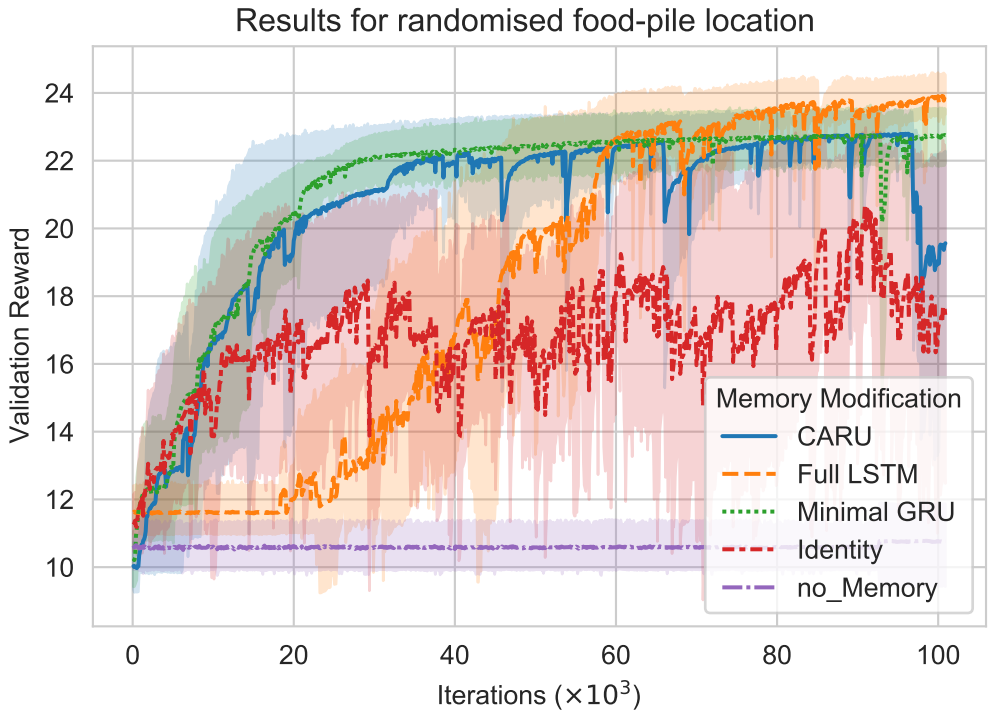


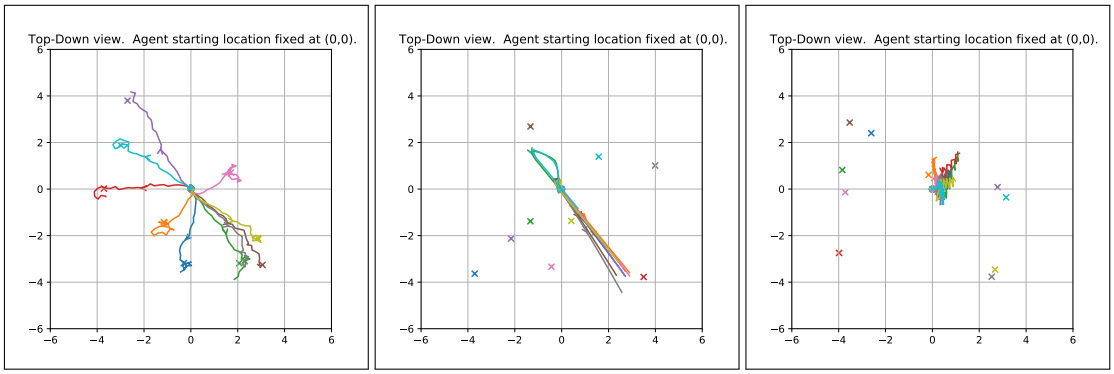
Figure 6.1: Performance of different memory modifications tackling randomised food location experiment, the results were averaged over 20 trials.

The Table 6.1 shows that the Full-LSTM model outperforms the other memory types with an average reward of 23.88, closely followed by the MGU model at 22.74. These results underscore the efficacy of more complex memory mechanisms, such as those found in Full-LSTM, in capturing and using information from the environment to make more informed decisions, thereby achieving higher rewards. The MGU model, despite its minimalistic design, also demonstrates significant effectiveness, suggesting that even simpler gating mechanisms can provide substantial benefits over traditional RNN architectures without memory enhancements.

The CARU model, with an average reward of 19.56, and the Identity model, at 17.35, show moderate performance. While CARU's context-adaptive gating mechanism gives it an edge over more basic models, it does not achieve the same level of performance as the Full-LSTM and MGU models. The Identity model, lacking a gating mechanism, performs better than the no-memory baseline but falls short compared to its counterparts with more sophisticated memory structures.

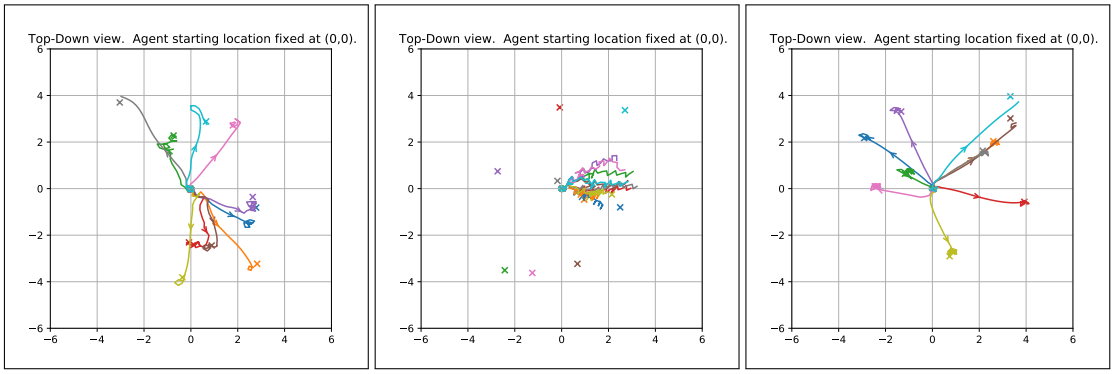
The no-memory baseline, with the lowest average reward of 10.76, clearly illustrates the importance of incorporating memory into reinforcement learning models. The significant performance gap between the no-memory baseline and the memory-equipped models highlights how memory mechanisms can enhance an agent's ability to learn and adapt, leading to better

decision-making and higher rewards. Figs. 6.2 to 6.5 shows the "*Identity*", CARU, Minimal GRU and full LSTM memory modification path at 100,000 iterations, visualising their route and movement strategy towards the goal. Each trial is trained separately, and the validation set of food positions is randomised, separating its value from the training set of food positions.



(a) Trial 1 (b) Trial 2 (c) Trial 3

Figure 6.2: First three trials of "*Identity*" memory model agent path captured from a top-down view at 1000000 iterations

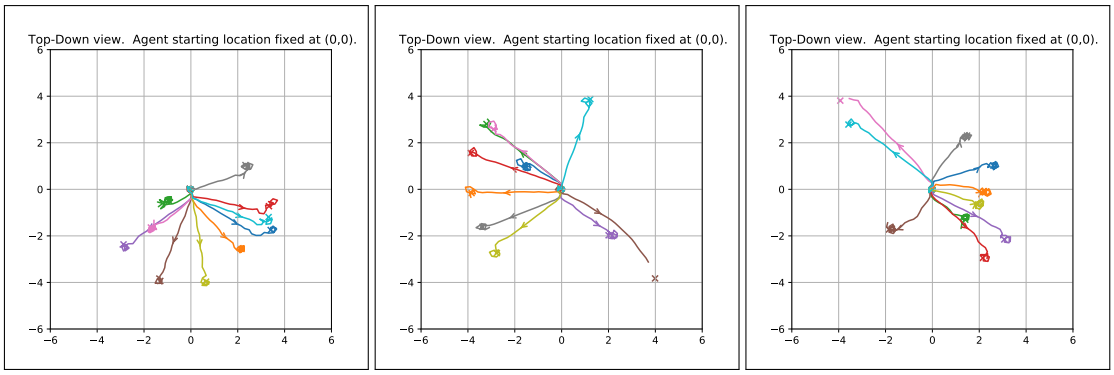


(a) Trial 1 (b) Trial 2 (c) Trial 3

Figure 6.3: First three trials of CARU memory model agent path captured from a top-down view at 1000000 iterations

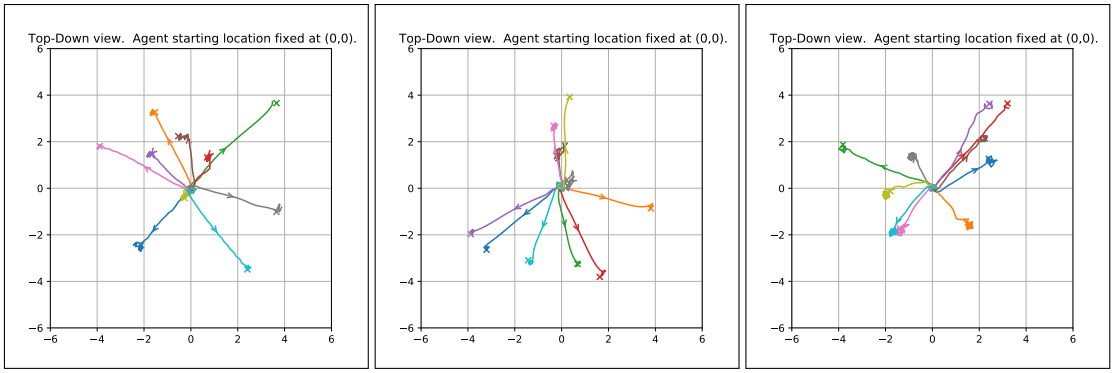
6.5 Discussion

In Fig. 6.1, it can be observed that the Full LSTM memory augmentation performance had a lower start until 20,000 iterations. Over time, this memory augmentation's performance improved, albeit slower than the other memory models. However, it eventually managed to converge with a higher average validation reward.



(a) Trial 1 (b) Trial 2 (c) Trial 3

Figure 6.4: First three trials of Minimal GRU memory model agent path captured from a top-down view at 1000000 iterations



(a) Trial 1 (b) Trial 2 (c) Trial 3

Figure 6.5: First three trials of Full LSTM memory model agent path captured from a top-down view at 1000000 iterations

In contrast, the "*Identity*" memory model had an unstable performance across the ten trials conducted in the experiment. This is likely because the identity memory model does not actively modify the network's memory, resulting in a lack of learning and instability in performance.

The minimal GRU and CARU models had relatively similar performances. Typically, the minimal GRU model may struggle with tasks requiring longer-term dependencies. On the other hand, the CARU model dynamically adapts the memory according to the task, making it more suited to tasks with varying memory requirements.

However, the results of the conducted experiment show that the CARU model had a consistent performance and failed to remain stable after reaching around 90,0000 iterations.

Overall, the results suggest that for these learning parameters and this problem, it seems that LSTM and Minimal GRU beat CARU and "*Identity*". It can be concluded that the choice of memory model is an essential factor in determining the performance of neural networks in the control task designed for this chapter. The "*Identity*" memory model may not be effective

in modifying memory and is therefore not recommended for the control task intended in this chapter. On the other hand, for a better learning rate (tuned for CARU), CARU might beat the full-LSTM memory model.

Moreover, this study observed that the Backpropagation Through Time (BPTT) algorithm could navigate using even a basic form of memory, referred to as the "*Identity*" memory modification. However, its performance was not as stable as that achieved with the full Long Short-Term Memory (LSTM) memory modification, as depicted in Figs. 6.2 and 6.5.

In the top-down view of the path taken by each memory model agent shown in Figs. 6.3 to 6.5, a joint movement strategy can be observed, where the agent moved towards a particular location at the beginning of each experiment consistently. Afterwards, the agent deduced a movement strategy that separated its path from other instances of food positions and moved towards the dedicated food (The same movement strategy can be seen in the previous chapter in Fig. 5.5b). It is important to note that these diagrams (Figs. 6.2 to 6.5) only show the best-performing trial and serve merely as illustrative examples of what can go wrong or right during the navigation process. For better comparison of the results and to draw more robust conclusions, readers should refer primarily to Fig. 6.1.

The data presented in the Table 6.1 provides compelling evidence of memory's critical role in enhancing reinforcement learning agents' performance. The superior performance of the Full-LSTM and MGU models over their counterparts demonstrates the value of integrating advanced memory architectures that offer nuanced control over information retention and processing. These findings suggest that incorporating memory not only aids in handling complex tasks but also significantly boosts the agent's learning efficiency and adaptability. Consequently, this study advocates for a deeper exploration of memory models and their integration within neural networks, emphasising the need for further research to harness the full potential of memory-enhanced learning algorithms in diverse and dynamic environments.

6.6 Chapter Conclusions

This chapter has undertaken a detailed exploration of the integration of memory models within the context of Recurrent Neural Networks (RNNs) trained through Backpropagation Through Time (BPTT), shedding light on the complexities and challenges associated with pairing different memory architectures with the physics model of the system.

One of the critical insights from this investigation is the role of differentiability in memory

models for effective application of BPTT. Memory architectures such as the Full LSTM, Minimal GRU, "*Identity*", and CARU, with their inherent differentiable structures, were examined for their suitability in training RNNs. These models were selected based on their theoretical capacity to enhance learning through their unique gating mechanisms or memory integration strategies.

However, it's important to acknowledge that the experimental comparisons conducted in this chapter, particularly those involving the "*Identity*" and CARU models, highlighted potential issues in memory integration rather than unequivocally demonstrating enhanced system efficacy. The absence of a robust no-memory baseline in Chapter 5 limited the scope for drawing definitive conclusions about the benefits of integrating memory into the system.

The nuanced behaviour observed in the "*Identity*" and CARU models raises critical questions regarding the optimal configuration and integration of memory within RNNs for control tasks. These findings suggest that while there is theoretical flexibility in integrating memory—directly into the physics model or as an independent module—the practical implications and effectiveness of such integration require careful consideration and further empirical validation.

The analysis of vector representations and the temporal evolution of the agent's memory state ($\vec{h}_t$) provided valuable perspectives on how memory, observations, and actions interact within an RNN framework. Yet, the performance disparities among the memory models underscore the necessity of a more granular understanding of how specific memory architectures influence neural network behaviour in control scenarios.

Regarding adaptability and performance, the Full LSTM model emerged as a relatively stable and effective option within the experimental confines of this study. This contrasts with the "*Identity*" model, which, despite its simplicity, highlighted challenges related to dynamic memory modification and its impact on network performance.

Reflecting on the journey from initial explorations with the modified GRU to the present evaluations, it is clear that the choice and integration of memory structures significantly influence the outcomes of neural network-based control systems. While this chapter has contributed to a deeper understanding of these dynamics, the observations underscore the importance of further research. Specifically, future work should aim to establish more comprehensive baselines, explore a wider array of memory models, and rigorously assess their efficacy in increasingly complex control environments.

The findings thus far pave the way for continued exploration into the role of memory in RNNs, particularly in the context of adaptive control systems. Investigating emerging memory models and rigorously evaluating their contributions to the field remains imperative, ensuring

that future advancements are grounded in solid empirical evidence and contribute meaningfully to our understanding of neural network optimisation and control.

Chapter 7

Conclusions

This thesis explored the integration of memory mechanisms within Reinforcement Learning (RL) and Adaptive Dynamic Programming (ADP), addressing the challenges of dynamic and uncertain environments. The primary aim was to investigate how incorporating memory cells via Backpropagation Through Time (BPTT) could improve agent adaptability and decision-making in control problems.

7.1 Synthesis of Research Findings

Memory-Augmented Approaches: Chapter 3 introduced a memory-augmented tabular Q-learning approach for navigating mazes with dynamically shifting exit locations, providing a simpler alternative to Recurrent Neural Networks (RNNs) for sequential information processing. The development and validation of this approach indicated the potential for more efficient solutions that avoid the computational demands and training challenges associated with RNNs. The findings showed that the memory-augmented tabular Q-learning approach consistently outperformed other algorithms in achieving optimal steps to reach all exits in various maze configurations, highlighting its robustness and efficiency even in controlled environments.

The experiments demonstrated that the tabular Q-learning with a customised memory mechanism achieved significant performance improvements. Specifically, compared to other experimented algorithms, it accumulated the least number of steps on average to reach all exits in the complex looped maze, showcasing its ability to handle more intricate navigation tasks. This approach's uniform performance, indicated by a variance of 0.0, underscores its robustness. However, it is crucial to acknowledge that the deterministic nature of this result is partly due

to the controlled conditions under which the experiments were conducted. In more dynamic or complex environments, a greater degree of variance in the steps required to reach exits may be observed, reflecting the challenges of adapting to new and unforeseen obstacles.

Continuous Space Navigation: Chapter 4 examined the performance of BPTT and advanced RL algorithms in continuous environments. The experiments highlighted BPTT's role in enhancing navigation efficiency and adaptability, offering potential improvements over traditional RL methods. This chapter detailed how BPTT facilitated smoother and more precise navigation in complex tasks involving long trajectories and intricate cost functions. The empirical results validated BPTT's potential in continuous environments, demonstrating its ability to maintain stability and achieve higher rewards than other RL methods.

The bicycle navigation task illustrated the challenges of learning optimal control strategies in continuous environments. The cost function gradient exhibited pronounced steepness at points that may not be strategically advantageous for learning. Specifically, these points often occurred after the agent had entered an uncontrollable position, meaning the trajectory had deviated significantly from an optimal path, or the bicycle had lost its balance. Techniques such as regularisation, reward shaping, early termination of trajectories, and adaptive learning rates were employed to mitigate these issues. These strategies effectively guided the learning process towards optimal navigation and balance in continuous space environments.

Challenges in Partially Observable Environments (POE): Chapter 5 demonstrated the effectiveness of BPTT, enhanced with a simplified Gated Recurrent Unit (GRU), in navigating complex POEs. This implementation highlighted the importance of memory cells and sensory observations in overcoming challenges posed by POEs. The ablation studies confirmed the necessity of recurrent memory and sensory-based information for solving POEs, showcasing the compatibility of BPTT with meta-learning and neural control principles. This chapter provided critical insights into integrating memory and sensory observations to overcome the limitations of POEs, underscoring BPTT's superiority in handling such environments compared to traditional RL algorithms.

The experiments in this chapter revealed that the BPTT algorithm could successfully navigate and make decisions based on limited information, which is crucial for real-world applications where complete state information is often unavailable. The BPTT agents exhibited a systematic exploration method, sampling and recording food heights in the environment to deduce the direction of the highest food density. This adaptive behaviour illustrated the algorithm's ability to handle partial observability and make informed decisions based on sensory input and memory.

Integration Challenges and Insights: Chapter 6 discussed the integration of memory models with RNNs trained through BPTT, revealing the complexities of pairing different memory architectures with the physics model of the system. This chapter provided insights into the nuances of memory models and the importance of differentiability for effective application, emphasising the need for empirical validation. The comparisons of various memory models, such as Full LSTM, Minimal GRU, and CARU, provided a deeper understanding of their performance and suitability for different control tasks, highlighting the challenges and opportunities in memory model integration.

The analysis of vector representations and the temporal evolution of the agent's memory state provided valuable perspectives on how memory, observations, and actions interact within an RNN framework. The performance disparities among the memory models underscored the necessity of a more granular understanding of how specific memory architectures influence neural network behaviour in control scenarios. The Full LSTM model emerged as a relatively stable and effective option, contrasting with the "Identity" model, which highlighted challenges related to dynamic memory modification and its impact on network performance.

7.2 Reflection on Research Questions and Objectives

Reflecting on the initial research questions, this thesis has demonstrated that integrating memory with BPTT significantly improves agents' learning efficiency and adaptability in complex environments. The development and evaluation of the BPTT algorithm have showcased its superiority over traditional RL algorithms in tasks requiring extended strategic planning and decision-making. These findings address the core research questions, affirming the hypothesis that memory mechanisms, when effectively integrated with BPTT, can enhance the capabilities of RL agents.

The objectives set forth at the outset of this research have been met with notable accomplishments:

- The reintroduction and implementation of the BPTT algorithm within the RL framework have laid a methodological foundation for memory integration, allowing agents to handle complex and dynamic environments more effectively.

- The integration of memory cells in the BPTT algorithm and its critical evaluation in dynamic environments have highlighted the effectiveness of memory mechanisms in

improving agent adaptability and performance in control tasks. The empirical results showed that memory-augmented agents could navigate challenging environments with higher efficiency and stability.

- The comparative analysis conducted in earlier chapters has quantitatively demonstrated the advantages of integrating memory, highlighting notable enhancements in adaptability, efficiency, and strategic planning. This analysis distinctly illustrates the flexibility of BPTT in integrating and effectively using memory cells.

- Through this research, the architectural and computational implications of incorporating BPTT and memory mechanisms into RL algorithms have been thoroughly analysed, providing valuable insights into their practical applicability. The detailed exploration of various memory models and their impact on learning algorithms has contributed to a more nuanced understanding of their roles in enhancing RL performance.

7.3 Novel Contributions Revisited

This thesis contributes to the field of reinforcement learning and adaptive dynamic programming by refining the application of memory within BPTT, specifically for control problems:

- The use of advanced memory cell manipulations with BPTT algorithm builds on existing BPTT frameworks that already incorporate memory functions, refining these ideas by focusing on the unique demands of control problem scenarios. This advancement allows for more precise and efficient handling of temporal dependencies and state transitions, often challenging in dynamic systems.

- Through an evaluation of memory-augmented agents in dynamic environments, this work provides a nuanced understanding of how memory integration influences adaptability and efficiency in such contexts compared to traditional RL methods. The findings suggest that integrating advanced memory techniques enables agents to better navigate environments with complex and changing dynamics, improving learning stability, speed, and overall performance.

- The insights gained contribute to the ongoing discourse on optimising learning agent designs, particularly highlighting the practical challenges and benefits of integrating memory cells in the BPTT framework. The comparison and analysis offer valuable

perspectives on how memory can effectively enhance learning algorithms, making them more robust and versatile in handling diverse and unpredictable environments.

7.4 Future Directions

The findings of this thesis pave the way for several promising avenues in further research, each with the potential to substantially advance the field of reinforcement learning and adaptive dynamic programming:

1. **Investigating Complex Environmental Dynamics**: Future studies could explore more intricate and variable environmental conditions that challenge current learning models. This includes environments with higher degrees of unpredictability or those that change in real-time, requiring agents to adapt continuously and dynamically. Research in this area could lead to more robust algorithms capable of handling real-world complexities.

2. **Optimising Memory Architectures**: There is significant scope for improving the efficiency and scalability of memory mechanisms within learning systems. Future work could optimise these architectures to reduce computational overhead while enhancing performance. This might involve refining existing memory cells, exploring new forms of short-term and long-term memory integration, or developing lightweight versions of complex memory systems that maintain performance while being more computationally efficient.

3. **Integration of Emerging Machine Learning Techniques**: As machine learning evolves, integrating the latest advancements with memory-augmented systems offers a fertile ground for research. This could include applying cutting-edge techniques such as federated learning, transfer learning, or generative adversarial networks within memory-based frameworks. Such integration could help tackle challenges like sample efficiency and transferability across different environments.

4. **Exploring Real-World Applications**: Applying the developed methodologies to real-world scenarios, such as autonomous vehicles, robotic control, and adaptive systems, can provide valuable insights into the practical challenges and benefits of memory-augmented RL algorithms. This can also help refine the algorithms to suit specific application needs better.

5. **Developing Hybrid Models**: Combining memory-augmented RL algorithms with other advanced AI techniques, such as reinforcement learning with evolutionary strategies or neuro-evolution, could create more robust and adaptable systems. Investigating these hybrid models could lead to breakthroughs in how learning algorithms can be optimised and applied to complex problems.

6. **Long-Term Learning and Adaptation**: Future research could focus on how memory-augmented RL algorithms perform over extended periods and how they adapt to long-term changes in their environment. This includes studying their ability to retain and use past experiences to improve future decision-making and adaptability.

7.5 Final Reflections

This thesis has contributed to enhancing the understanding of how memory mechanisms can be integrated into reinforcement learning (RL) and adaptive dynamic programming (ADP). By addressing the research questions outlined and achieving its objectives, this work offers a perspective that adds to the ongoing discussions in AI, particularly concerning agent navigation and exploration.

The research has demonstrated that incorporating memory mechanisms can improve the performance and adaptability of RL agents in dynamic and uncertain environments. This is important as it suggests that agents can operate more effectively and make decisions considering historical context, similar to human cognitive processes.

The methodologies developed in this thesis provide a framework that could be useful for future research in this area. As explored here, integrating memory into learning algorithms indicates a potential to enhance agent capabilities in complex environments. Such advancements could be relevant as AI systems become increasingly prevalent in critical applications such as autonomous vehicles and interactive technologies.

Theoretically, this thesis invites further exploration within the fields of RL and ADP by proposing alternative approaches and highlighting areas for additional research. Practically, the findings may help inform the development of more capable AI systems, which could impact various applications involving complex decision-making and interaction.

In conclusion, this thesis contributes to the discourse on BPTT by exploring how memory integration can improve AI agents' functionality. It lays a foundation for further studies to

explore and expand on these initial findings, potentially leading to broader applications and more profound understanding.

The research presented here underscores the transformative potential of memory-augmented learning algorithms. By advancing the capabilities of RL agents, this work contributes to the broader field of artificial intelligence and sets the stage for future innovations in adaptive learning systems. The exploration of memory integration in RL provides a pathway towards creating more intelligent, adaptable, and robust AI systems capable of addressing the complexities of real-world environments.

Appendix

Appendix 1: Tabular Q-learning Experiment to explore Eulerian Tours

This appendix provides the source code for an experiment that uses Tabular Q-learning to explore Eulerian Tours in mazes.

Listing 7.1: Source code for the Tabular Q-learning Experiment to explore Eulerian Tours, implemented in Python.

```
1  import random
2  import sys
3  import numpy as np
4
5
6  def environment_step(tempmaze, action, state, goal_state):
7      y,x = state
8      dy,dx=action_effects[action]
9      hit_wall = False
10     new_x = x+dx
11     new_y = y+dy
12     if new_x <0 or new_x>=maze_width:
13         # off grid
14         new_x = x
15     if new_y <0 or new_y>=maze_height:
16         # off grid
17         new_y = y
18     if tempmaze[new_y,new_x] == 1:
19         # hit wall
20         new_y=y
21         new_x=x
```

```
22          hit_wall = True
23      new_state = [new_y,new_x]
24
25      reward = -1
26      done = (new_state==goal_state)
27      return new_state, reward, done, hit_wall
28
29  def run_policy(currentState, ep,previous_action):
30      sy,sx= currentState
31      if np.random.uniform(0,1) < ep:
32          choice = np.random.choice(range(len(action_names)))
33      else:
34          q_values=Qtable2[current_maze[sy,sx+1],current_maze[sy,sx-1],
                current_maze[sy+1,sx], current_maze[sy-1,sx],previous_action, :]
35          assert len(q_values.shape)==1
36          assert q_values.shape[0]==num_actions
37          best_q_value=q_values.max()
38          best_q_indices=np.argwhere(q_values == best_q_value).flatten().
                tolist()
39          choice = np.random.choice(best_q_indices)
40          assert choice>=0 and choice<num_actions
41      return choice
42
43
44  def apply_q_update(state, action, previous_action, reward, next_state, done
        , timestep):
45      sy,sx=state
46      nsy,nsx=next_state
47      current_q_value =Qtable2[current_maze[sy,sx+1],current_maze[sy,sx-1],
            current_maze[sy+1,sx], current_maze[sy-1,sx],previous_action, action
            ]
48      target_q_value =  reward
49      if not done:
50          future_state_q_values = Qtable2[current_maze[nsy,nsx+1],
                current_maze[nsy,nsx-1], current_maze[nsy+1,nsx], current_maze[
                nsy-1,nsx],action, :]
51          assert len(future_state_q_values.shape)==1
52          assert future_state_q_values.shape[0]==num_actions
53          target_q_value += discount_factor * future_state_q_values.max()
```

```
54      Qtable2[current_maze[sy,sx+1],current_maze[sy,sx-1], current_maze[sy+1,
            sx], current_maze[sy-1,sx],previous_action, action] += learning_rate
            * (target_q_value - current_q_value)  # update
55      current_q_value = Qtable2[current_maze[sy, sx + 1], current_maze[sy, sx
            - 1], current_maze[sy + 1, sx], current_maze[sy - 1, sx],
            previous_action, action]
56      return current_q_value, state
57
58  action_names=["North","South","West","East"]
59  action_effects=[[-1,0],[1,0],[0,-1],[0,1]]
60  num_actions=len(action_names)
61  iterations = 500 *4 * 10
62  #iterations = 5000
63  learning_rate = 0.1
64  discount_factor=.9
65  maps = []
66  '''
67  name = 'smallCorridor'
68  SmallCorridor=np.array([
69          [0,0,0],])
70  SmallCorridor_goal_state = [[0,0],[0,2]]
71  SmallCorridor_start_state = [0,1]
72  maps.append([name,SmallCorridor,SmallCorridor_start_state,
        SmallCorridor_goal_state,3])
73  '''
74  name = 'smallCorridor'
75  SmallCorridor=np.array([
76          [1,1,1,1,1,1,1],
77          [1,1,1,1,1,1,1],
78          [1,1,1,1,1,1,1],
79          [1,1,0,0,0,1,1],
80          [1,1,1,1,1,1,1],
81          [1,1,1,1,1,1,1],
82          [1,1,1,1,1,1,1]])
83  SmallCorridor_goal_state = [[3,2],[3,4]]
84  SmallCorridor_start_state = [3,3]
85  maps.append([name,SmallCorridor,SmallCorridor_start_state,
        SmallCorridor_goal_state,3])
86
```

```
87  #
88  name = "Tshaped"
89  Tshaped =np.array([
90          [1,1,1,1,1,1,1],
91          [1,1,1,0,1,1,1],
92          [1,1,1,0,1,1,1],
93          [1,1,1,0,1,1,1],
94          [1,1,1,0,1,1,1],
95          [1,0,0,0,0,0,1],
96          [1,1,1,1,1,1,1]])
97  Tshaped_goal_states = [[5,1],[5,5]]
98  #Tshaped_goal_states = [[5,1]]
99  Tshaped_start_state = [1,3]
100 maps.append([name,Tshaped,Tshaped_start_state,Tshaped_goal_states,10])
101 #
102 name = 'LongCorridor'
103 LongCorridor=np.array([
104         [1,1,1,1,1,1,1],
105         [1,0,1,0,0,0,1],
106         [1,0,1,1,1,0,1],
107         [1,0,1,0,0,0,1],
108         [1,0,1,1,1,0,1],
109         [1,0,0,0,0,0,1],
110         [1,1,1,1,1,1,1]])
111 LongCorridor_goal_state = [[1,3],[3,3]]
112 LongCorridor_start_state = [1,1]
113 maps.append([name,LongCorridor,LongCorridor_start_state,
        LongCorridor_goal_state,18])
114 #
115 name = 'crossmap'
116 cross=np.array([
117         [1,1,1,1,1,1,1],
118         [1,1,1,0,1,1,1],
119         [1,1,1,0,1,1,1],
120         [1,0,0,0,0,0,1],
121         [1,1,1,0,1,1,1],
122         [1,1,1,0,1,1,1],
123         [1,1,1,1,1,1,1]])
124 cross_start_state = [3,3] # center square
```

```
125 cross_goal_states = [[1,3],[3,5],[5,3],[3,1]]
126 maps.append([name,cross,cross_start_state,cross_goal_states,12])
127 '''
128 #
129 '''
130 name = 'complex_looped'
131 complex_looped = np.array([
132 [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
133 [1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1],
134 [1,1,1,0,1,0,0,0,1,1,1,1,0,1,0,1,0,1,1,1],
135 [1,0,0,0,1,0,1,0,0,0,0,0,0,1,0,1,0,0,0,1],
136 [1,0,1,1,1,0,1,1,1,1,0,1,1,1,1,1,1,1,0,1],
137 [1,0,0,0,0,0,1,1,1,1,0,1,1,1,1,1,0,0,0,1],
138 [1,0,1,1,1,1,1,0,0,0,0,0,0,0,0,1,1,1,0,1],
139 [1,0,0,0,0,0,1,0,1,1,1,1,1,1,0,1,0,0,0,1],
140 [1,0,1,1,1,1,1,0,1,0,0,0,0,1,0,1,0,0,0,1],
141 [1,0,0,0,0,0,1,0,1,0,1,1,0,1,0,1,0,1,0,1],
142 [1,0,1,1,1,0,0,0,0,0,0,1,0,0,0,0,0,1,0,1],
143 [1,0,1,1,1,0,1,0,1,0,1,1,0,1,0,1,0,1,0,1],
144 [1,0,1,1,1,0,1,0,1,0,0,0,0,1,0,1,0,0,0,1],
145 [1,0,0,0,0,0,1,0,1,1,1,1,1,1,0,1,1,1,0,1],
146 [1,0,1,1,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1],
147 [1,0,1,1,1,0,1,1,1,1,1,1,1,1,0,1,1,1,0,1],
148 [1,0,0,0,1,0,1,0,0,0,0,0,0,1,0,1,0,0,0,1],
149 [1,1,1,0,1,0,1,0,1,1,1,1,0,1,0,1,0,1,1,1],
150 [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
151 [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
152 ])
153 height= len(complex_looped)
154 width = len(complex_looped[0])
155 #complex_looped = np.genfromtxt("foo.csv",delimiter=',')
156 complexlooped_start_state = [10,10]
157 complexlooped_goal_states= [[1,1],[1,18],[18,1],[18,18]]
158 maps.append([name,complex_looped,complexlooped_start_state,
        complexlooped_goal_states,90])
159 uses_memory=False
160 trials = 10
161 mapstrain = [maps[-2]]
162 brains = ["wallfollower"]
```

```
163
164
165 for mapp in mapstrain:
166     print(mapp[0])
167     current_maze = mapp[1]
168     maze_shape = np.shape(current_maze)
169     maze_height = maze_shape[0]
170     maze_width = maze_shape[1]
171     for trial in range(0,trials+1):
172         print(trial)
173         reward_history = []
174         trajectory_length_history = []
175         qvalue_history = []
176         goal_history = []
177         Qtable2 = np.zeros((2,2,2,2,4,4))
178         epsilon_greedy = 0.1
179         for iteration in range(iterations+1):
180             time_step_history = 0
181             total_reward = 0
182             total_reward_eval = 0
183             c = 0
184             q_valuestep = []
185             goal_states = mapp[3]
186             goal_counter = 0
187             #np.random.shuffle(goal_states)
188             for goal_state in goal_states:
189                 # print("Starting trajectory with goal state",goal_state)
190                 state = mapp[2]
191                 done = False
192                 time_step = 0
193                 previous_action = 0
194                 action = 0
195                 while not done:
196                     # Choose an action
197                     action = run_policy(state, epsilon_greedy,
                            previous_action)
198                     next_state, reward, done, hitwall= environment_step(
                            current_maze, action, state, goal_state)
```

```
199                     apply_q_update(state, action,previous_action, reward,
                            next_state, done, time_step)
200                     if next_state == goal_state:
201                         goal_counter+=1
202
203                     previous_action = action
204                     state = next_state
205                     total_reward += reward * (np.power(time_step,
                            discount_factor))
206                     time_step += 1
207                     if time_step >= (500):
208                         done = True
209                 time_step_history += time_step
210         for valid_map in maps:
211             print(valid_map[0])
212             time_step_history = 0
213             current_maze = valid_map[1]
214             maze_shape = np.shape(current_maze)
215             maze_height = maze_shape[0]
216             maze_width = maze_shape[1]
217             total_reward = 0
218             total_reward_eval = 0
219             c = 0
220             q_valuestep = []
221             goal_states = valid_map[3]
222             goal_counter = 0
223             for goal_state in goal_states:
224                 state = valid_map[2]
225                 done = False
226                 time_step = 0
227                 previous_action = 0
228                 action = 0
229                 while not done:
230                     # Choose an action
231                     action = run_policy(state, 0., previous_action)
232                     # print("time_step",time_step,"state",state,"action",
                            action)
233                     next_state, reward, done, hitwall = environment_step(
                            current_maze, action, state,
```

```
234                                                                 goal_state
                                                                        )

235                     previous_action = action
236                     if next_state == goal_state:
237                         goal_counter += 1
238                     state = next_state
239                     total_reward_eval += reward * (discount_factor **
                            time_step)
240                     time_step += 1
241                     if time_step >= (500):
242                         done = True
243                 time_step_history += time_step
244             reward_history.append(total_reward_eval)
245             trajectory_length_history.append(time_step_history)
246             goal_history.append(goal_counter)
247             print("Iteration " + str(iteration) + " Reward " + str(
248             trajectory_length_history[-1]) + " Goal reached " + str(
                    goal_history[-1]))
249             np.save("runs/"+str(trial) +"_map_"+valid_map[0]+"_Brain_"+
                    brains[0]+"Goals_reached.npy",np.array(goal_counter))
250             np.save("runs/"+str(trial) +"_map_"+valid_map[0]+"_Brain_"+
                    brains[0]+"_reward.npy",np.array(total_reward_eval))
251             np.save("runs/"+str(trial) + "_map_"+valid_map[0]+"_Brain_"+
                    brains[0]+ "_step.npy", np.array(time_step_history))
```

Appendix 2: Randlov Bicycle Experiment

This appendix offers the source code for the Randlov Bicycle Experiment, which focuses on balancing a simulated bicycle.

Fig. 1 is the bicycle's representation as seen from behind, the bicycle's balance is ω, CM is the Centre of Mass of the bicycle and cyclist combined, and d is the agent's choice of the displacement of the CM perpendicular to the plan of the bicycle. Mg is considered the total weight of the agent (the rider and the bicycle combined), and F_{cen} is the centre of force.

The following equation describes the system's mechanics where the angle α is defined as the total angle of tilt of the centre of mass.

Table 1: Notation and values for the bicycle system taken from Randløv and Alstrøm [1998].

Notation	Description	Value
α	Total angle of tilt of the centre of mass	
c	Horizontal distance between the front wheel touches the ground and the CM	66 cm
CM	The Center of Mass of the bicycle and cyclist as a total	
d	The agent's choice of the displacement of the CM perpendicular to the plane of the bicycle	
d_{cm}	The vertical distance between the CM for the bicycle and the cyclist	30 cm
F_{cen}	The center of force	
h	Height of the CM over the ground	94 cm
l	Distance between the front tire and the back tire	111 cm
M_c	Mass of the bicycle	15 Kg
M_d	Mass of a tire	1.7 Kg
M_g	Agent's total weight (rider and the bicycle's weights combined)	
M_p	Mass of the cyclist	60 Kg
r	Radius of a tire	34 cm
$\dot{\sigma}$	The angular velocity of a tire	$\dot{\sigma} = \frac{v}{r}$
ω	Bicycle's balance	
T	The torque the agent applies on the handlebars	
v	The velocity of the bicycle	10 km/h
θ	The steer angle of the bicycle	

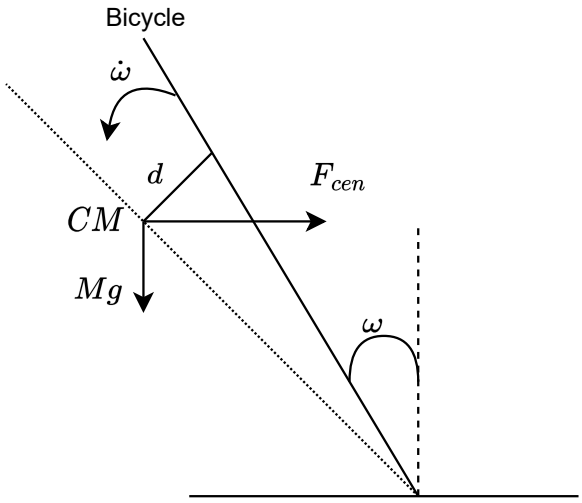


Figure 1: Bicycle's representation as seen from behind (This image is inspired from Randløv and Alstrøm [1998] paper.)

$$\alpha = \omega + \arctan\left(\frac{d}{h}\right). \tag{7.1}$$

The angular acceleration $\dot{\omega}$ can be calculated as:

$$\dot{\omega} = \frac{1}{I_{bicycle\ and\ cyclist}}(Mhg\sin\alpha) - \cos\alpha\left(I_{dc}\dot{\sigma}\dot{\theta} + sign(\theta)v^2\left(\frac{M_d r}{r_f} + \frac{M_d r}{r_b} + \frac{Mh}{r_{CM}}\right)\right), \tag{7.2}$$

this equation gives the mechanical equation for angular momentum. The physical contents of the right side are terms for gravitation, the effects of tyre angular momentum conservation, and the fictional centrifugal force.

The terms I_{dc}, $\dot{\omega}$, and $\dot{\theta}$ play a vital role in understanding why riding a bicycle is more accessible than maintaining balance steadily. The stabilising effects resulting from the conservation

of angular momentum in the tires are essential contributors to bicycle stability. These effects, commonly called gyroscopic effects, are influenced by the angular velocity of the tires $\dot{\omega}$ and, consequently, by the bicycle's velocity.

Gyroscopic effects arise due to the conservation of angular momentum in rotating bodies, such as bicycle tires. When the wheels of a moving bicycle experience a change in their orientation (caused by tilting or steering the bicycle), they are met with resistance, maintaining their plane of rotation. This resistance creates a stabilising influence, making it easier to balance the bicycle during motion compared to keeping it balanced while stationary.

The magnitude of the gyroscopic effect is directly proportional to the angular velocity of the tires $\dot{\omega}$, which, in turn, is related to the bicycle's velocity. As the bicycle's speed increases, the gyroscopic effect becomes more pronounced, contributing significantly to the bicycle's stability during motion.

Understanding the significance of I_{dc}, $\dot{\omega}$, and $\dot{\theta}$ and their relationship with gyroscopic effects provides valuable insights into the mechanics behind bicycle stability and why riding a moving bicycle is generally easier than maintaining balance when the bicycle is stationary.

The angular acceleration $\ddot{\theta}$ of the handlebars is:

$$\ddot{\theta} = \frac{T - I_{dv}\dot{\omega}\dot{\sigma}}{I_{dl}}. \tag{7.3}$$

In this equation, $\ddot{\theta}$ represents the angular acceleration of the handlebars, which indicates how quickly the handlebars are rotating. The term T represents the torque applied to the handlebars, which can influence the rotational motion. The symbols I_{dv}, $\dot{\omega}$, and $\dot{\sigma}$ have specific meanings in the context of the bicycle model, as previously explained. I_{dv} is the moment of inertia of the front wheel, $\dot{\omega}$ is the angular velocity of the bicycle, and $\dot{\sigma}$ is the angular velocity of the front tire. The term I_{dl} represents the moment of inertia of the handlebars and relates to how they resist changes in their rotational motion.

Because some second (and higher) order terms have been ignored, these equations do not provide an exact analytical description.

The observable values such as ω, $\dot{\omega}$,θ, $\dot{\theta}$,$\sin\psi_g$ and $\cos\psi_g$ are sent to the agent at each time-step where ψ_g is the angle of the bicycle to the goal, and the agent is expected to return d and the torque T which is applied to the handlebar.

Fig. 2 shows the bicycle from the above perspective. The front and rear tyres take different paths in a curve with different radii.

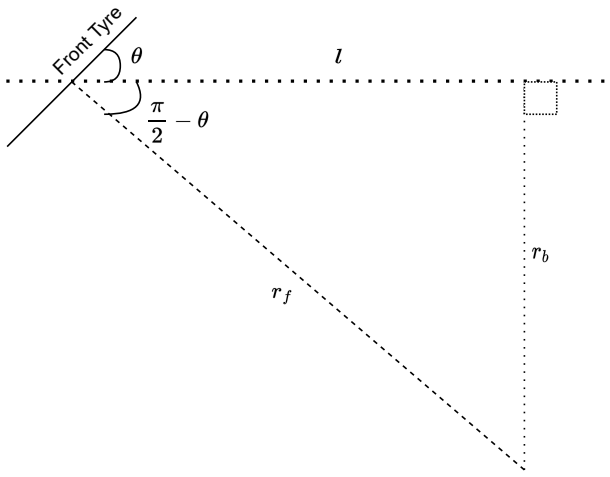


Figure 2: Bicycle's representation as seen from above (This image was inspired from Randløv and Alstrøm [1998]) paper.

The front tyre follows the longest path. The radius for the front tyre is:

$$r_f = \frac{l}{|\cos\frac{\pi}{2} - \theta|} = \frac{l}{|\sin\theta|}, \tag{7.4}$$

and for the back tyre:

$$r_b = l\left|\tan\frac{\pi}{2} - \theta\right| = \frac{l}{|\tan\theta|}. \tag{7.5}$$

For the CM, the radius can be calculated as:

$$r_{CM} = \left((l-c)^2 + \frac{l^2}{\tan\theta^2}\right)^{\frac{1}{2}}. \tag{7.6}$$

The equations of the position of the tyres for the front tyre:

$$\begin{pmatrix} x_f \\ y_f \end{pmatrix}_{(t+1)} = \begin{pmatrix} x_f \\ y_f \end{pmatrix}_{(t)} + vdt \begin{pmatrix} -\sin(\psi+\theta+sign(\psi+\theta)\arcsin\frac{vdt}{2r_f}) \\ \cos(\psi+\theta+sign(\psi+\theta)\arcsin\frac{vdt}{2r_f}) \end{pmatrix}. \tag{7.7}$$

The back tyre is calculated differently from the front tyre:

$$\begin{pmatrix} x_b \\ y_b \end{pmatrix}_{(t+1)} = \begin{pmatrix} x_b \\ y_b \end{pmatrix}_{(t)} + vdt \begin{pmatrix} -\sin(\psi+sign(\psi)\arcsin\frac{vdt}{2r_b}) \\ \cos(\psi+sign(\psi)\arcsin\frac{vdt}{2r_b}) \end{pmatrix}. \tag{7.8}$$

The moments of inertia values were calculated by:

$$I_{bicycleandcyclist} = \frac{13}{3}M_c h^2 + M_p\left(h + d_{CM}\right)^2, \tag{7.9}$$

$$I_{dc} = M_d r^2, \tag{7.10}$$

$$I_{dv} = \frac{3}{2}M_d r^2, \tag{7.11}$$

$$I_{dl} = \frac{1}{2}M_d r^2. \tag{7.12}$$

In the moments of inertia equation, the M_c is the mass of the bicycle, M_d is the mass of the tyre, and M_p is the mass of the cyclist, the r indicates the radius of the tyre. d_{CM} denotes the agent's vertical distance between the CM for the bicycle and the cyclist. l suggests the distance between the front tyre and the back tyre where they touch the ground.

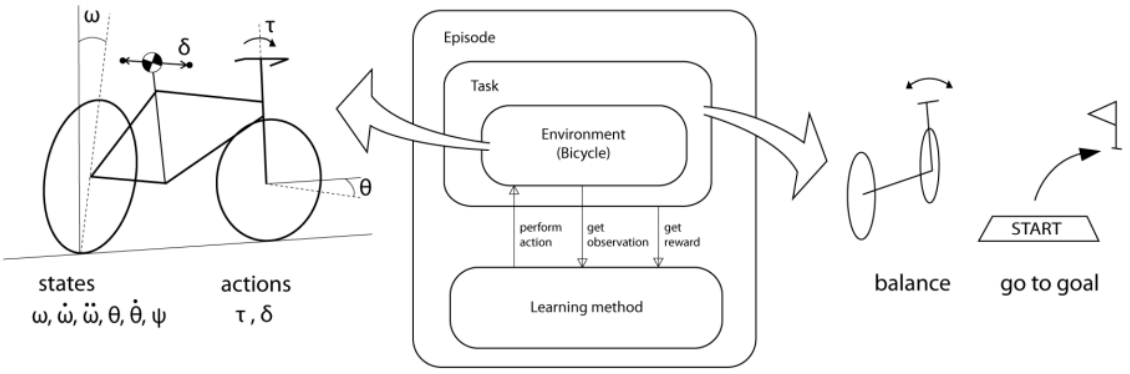


Figure 3: Bike physics as shown in the paper by Cam et al. [2013]

Listing 7.2: Source code for the Randlov Bicycle Experiment, implemented in Python.

```
1  import tensorflow as tf
2  from tensorflow import keras
3  import sys
4  import math
5  import numpy as np
6  import matplotlib.pyplot as plt
7  import matplotlib.lines as mlines
8  import matplotlib.patches as mpatches
9  from datetime import datetime
10 from dateutil.relativedelta import relativedelta
11 import argparse
12 VALIDATION = False
13
14 tf.keras.backend.set_floatx("float32")
15
16 gpus = tf.config.list_physical_devices('GPU')
17 if gpus:
18     print(gpus)
19     for gpu in gpus:
20         tf.config.experimental.set_memory_growth(gpu, True)
21         logical_gpus = tf.config.list_logical_devices('GPU')
22 #PARSER
```

```
23 randomised_state = False
24 parser = argparse.ArgumentParser(description='Optional app description')
25 parser.add_argument('--trialname', type=str, default="trial_1")
26 parser.add_argument('--with_psi_restriction', type=int, default=1)
27 parser.add_argument('--use_tanh', type=int, default=1)
28 parser.add_argument('--goal', type=int, default=0)
29 parser.add_argument('--core', type=int, default=1)
30 parser.add_argument('--test', type=str, default='psiRemoved')
31 args = parser.parse_args()
32 testt = str(args.test)
33 mask = int(args.core)
34 trial_name = str(args.trialname)
35 use_tanh = bool(args.use_tanh)
36 goal = bool(args.goal)
37 with_psi_restriction = bool(args.with_psi_restriction)
38 #AFINITY
39 use_cpu_afinity = False
40 if use_cpu_afinity:
41     import win32api, win32con, win32process
42     def setaffinity(mask):
43         pid  = win32api.GetCurrentProcessId()
44         handle = win32api.OpenProcess(win32con.PROCESS_ALL_ACCESS, True,
              pid)
45         win32process.SetProcessAffinityMask(handle,mask)
46     setaffinity(mask)
47 #EXPERIMENT SETTINGS
48 max_iterations = 1000
49 b = 60
50 pi = tf.constant(math.pi)
51 print(pi)
52 action_is_theta = True
53 maximum_dis = 0.02  # 0.02
54 maximum_torque = 2.
55 batch_size = 100
56 action_space = 2 if action_is_theta else 1
57 num_hidden_units = [24, 24]
58 trajectory_length = 1000
59 try_to_wrap_around_gradients = True
60 randomised_goal_position = False
```

```
61
62 refresh_unroll_frequency = 100
63 unroll_pseudo_initial_states_to_truth = True
64 initialise_wrap_around_gradients = True
65 #VISUALISATION
66 colors = ['red', 'blue', 'green', 'orange', 'black', 'yellow', 'purple', '
       pink', 'olive', 'cyan']
67 plt.ion()
68 prinit = True
69 graphical = False
70 save = True
71 display = 5
72 print_time = 10
73 filename = str(args.trialname) + "_with_psi_restriction_" + str(
       with_psi_restriction) + "_randomised_state_" + str(
74      randomised_state) + "_goal_" + str(goal) + "_test_" + testt + "_tanh_"
           + str(use_tanh)
75
76 def diff(t_a, t_b):
77      t_diff = relativedelta(t_b, t_a)  # later/end time comes first!
78      return '{h}h {m}m {s}s'.format(h=t_diff.hours, m=t_diff.minutes, s=
           t_diff.seconds)
79
80 #BIKE PHYSICS
81 # Units in meters and kilograms
82 c = tf.constant(0.66)  # Horizontal distance between point where front
       wheel touches ground and centre of mass
83 d_cm =  tf.constant(0.30)  # Vertical distance between center of mass and
       cyclist
84 h =  tf.constant(0.94)  # Height of center of mass over the ground
85 l = tf.constant(1.11)  # Distance between front tire and back tire at the
       point where they touch the ground.
86 m_c =  tf.constant(15.0)  # mass of bicycle
87 m_d =  tf.constant(1.7)  # mass of tire
88 m_p =  tf.constant(60.0)  # mass of cyclist
89 r =  tf.constant(0.34)  # radius of tire
90 v =  tf.constant(10.0 / 3.6)  # velocity of the bicycle in m / s 2.7
91 goal_rsqrd =  tf.constant(1.0)
92 # Useful Precomputations
```

```
93  m = m_c + m_p
94  inertia_bc = (13. / 3) * m_c * h ** 2 + m_p * (h + d_cm) ** 2  # inertia of
        bicycle and cyclist
95  inertia_dv = (3. / 2) * (m_d * (r ** 2))  # Various inertia of tires
96  inertia_dl = .5 * (m_d * (r ** 2))  # Various inertia of tires
97  inertia_dc = m_d * (r ** 2)  # Various inertia of tires
98  sigma_dot = v / r
99  # Simulation constants
100 gravity =  tf.constant(9.82)
101 delta_time =  tf.constant(0.01)  # 0.01 # 0.054 m forward per delta time
102 # If omega exceeds +/- 12 degrees, the bicycle falls.
103 omega_range = tf.constant(np.array([[-np.pi * 12 / 180, np.pi * 12 / 180]]
        * batch_size))  # 12 degree in SI units.
104 theta_range = tf.constant(np.array([[-np.pi / 2, np.pi / 2]] * batch_size))
105 psi_range = tf.constant(np.array([[-np.pi, np.pi]] * batch_size))
106 rando = 0.0
107 yg = 60.
108 xg = 0.
109 goal_position = tf.constant(tf.random.uniform(minval=-50, maxval=50, shape
        =(batch_size, 2)) * (1 if randomised_goal_position else 0))
110 if not randomised_goal_position:
111     goal_position += tf.constant([[xg, yg]] * batch_size)
112 def safe_divide(tensor_numerator, tensor_denominator):
113     # attempt to avoid NaN bug in tf.where: https://github.com/tensorflow/
            tensorflow/issues/2540
114     safe_denominator = tf.where(tf.not_equal(tensor_denominator, tf.
            zeros_like(tensor_denominator)),
115                                 tensor_denominator,
116                                 tensor_denominator + 1)
117     return tensor_numerator / safe_denominator
118 def reset():
119     # Lagoudakis (2002) randomizes the initial state "arout the equilibrium
            position"
120     if randomised_state:
121         theta = np.random.normal(0, 1, size=(batch_size, 1)) * np.pi / 180
122         omega = np.random.normal(0, 1, size=(batch_size, 1)) * np.pi / 180
123         thetad = np.zeros((batch_size, 1))
124         omegad = np.zeros((batch_size, 1))
125         omegadd = np.zeros((batch_size, 1))
```

```
126             xb = np.random.uniform(-60, 60, (batch_size, 1))
127             yb = np.zeros((batch_size, 1))
128             xf = xb + (np.random.rand(batch_size, 1) * 1 - 0.5 * 1) / 2  #
                    halved it for psi
129             yf = np.sqrt(1 ** 2 - (xf - xb) ** 2) + yb
130             psi = np.arctan((xb - xf) / (yf - yb))
131             psig = psi - np.arctan(safe_divide((xb - xg), yg - yb))
132             init_state = tf.concat(
133                 [omega, omegad, omegadd, theta, thetad, xf, yf, xb, yb, psi,
                        psig, np.zeros((batch_size, 1))],
134                 axis=1)
135         else:
136             theta = thetad = omega = omegad = omegadd = xf = yf = xb = yb = np.
                    zeros((batch_size, 1))
137             yf = yf + 1
138             psi = np.arctan((xb - xf) / (yf - yb))
139             psig = psi - np.arctan(safe_divide((xb - xg), yg - yb))
140             init_state = tf.concat(
141                 [omega, omegad, omegadd, theta, thetad, xf, yf, xb, yb, psi,
                        psig, np.zeros((batch_size, 1))],
142                 axis=1)
143         # omega, omega_dot, omega_ddot, theta, theta_dot, x_f, y_f, x_b, y_b,
                psi,psig, timestep
144         return init_state
145 # STATE initialisation
146 state_dimension = 12  # omega, omega_dot, omega_ddot, theta, theta_dot, x_f
        , y_f, x_b, y_b, psi,psig, timestep
147 initial_state = reset()
148 #step(state, action, trajectories_terminated, batch_size)
149 def step(state, action, trajectories_terminated, p_batch_size,
        corruption_to_physics_model=None):
150     s = state
151     omega = s[:, 0]
152     omegad = s[:, 1]
153     theta = s[:, 3]
154     thetad = s[:, 4]  # theta - handle bar, omega - angle of bicycle to
            verticle psi = bikes angle to the yaxis
155     xf = s[:, 5]
156     yf = s[:, 6]
```

```
157     xb = s[:, 7]
158     yb = s[:, 8]
159     psi = s[:, 9]
160     timestep = s[:, -1]
161     last_pos = s[:, 5:6]
162     last_xf = xf
163     last_yf = yf
164     T = action[:, 0] * maximum_torque
165     T = tf.where(T > maximum_torque, tf.ones_like(T) * maximum_torque, T)
166     T = tf.where(T < -maximum_torque, tf.ones_like(T) * -maximum_torque, T)
167     d = action[:, 1] * maximum_dis
168     d = tf.where(d > maximum_dis, tf.ones_like(d) * maximum_dis, d)
169     d = tf.where(d < -maximum_dis, tf.ones_like(d) * -maximum_dis, d)
170     r_f = tf.where(theta == 0., tf.constant(1.e8), safe_divide(1, tf.abs(tf
            .sin(theta))))
171     r_b = tf.where(theta == 0., tf.constant(1.e8), safe_divide(1, tf.abs(tf
            .tan(theta))))
172     r_cm = tf.where(theta == 0., tf.constant(1.e8),
173                     tf.sqrt((1 - c) ** 2 + (safe_divide(tf.pow(1, 2), (tf.
                        pow(tf.tan(theta), 2))))))
174     phi = omega + tf.atan(d / h)
175     # Equations of motion.
176     # --------------------
177     # Second derivative of angular acceleration:
178     omegadd = 1 / inertia_bc * (m * h * gravity * tf.sin(phi)
179                                 - tf.cos(phi) * (inertia_dc * sigma_dot *
                                    thetad
180                                                  + tf.sign(theta) * (v **
                                                     2) * (
181                                                         m_d * r * (1.0 /
                                                            r_f + 1.0 / r_b
                                                            )
182                                                         + m * h / r_cm)))
183     thetadd = (T - inertia_dv * sigma_dot * omegad) / inertia_dl
184     # Integrate equations of motion using Euler's method.
185     # ---------------------------------------------------
186     # Must update omega based on PREVIOUS value of omegad.
187     df = delta_time
188     omegad += omegadd * df
```

```
189     omega += omegad * df
190     thetad += thetadd * df
191     theta += thetad * df
192     # Handlebars can't be turned more than 80 degrees.
193     oneslikeTheta= tf.ones_like(theta)
194     theta = tf.where(theta > 1.3963, oneslikeTheta * 1.3963, theta)
195     theta = tf.where(theta < -1.3963, oneslikeTheta * -1.3963, theta)
196     # Wheel ('tyre') contact positions.
197     # --------------------------------
198     # Front wheel contact position.
199     front_term = psi + theta + tf.sign(psi + theta) * tf.asin(v * df / (2.
            * r_f))
200     back_term = psi + tf.sign(psi) * tf.asin(v * df / (2. * r_b))
201     xf += v * df * -tf.sin(front_term)
202     yf += v * df * tf.cos(front_term)
203     xb += v * df * -tf.sin(back_term)
204     yb += v * df * tf.cos(back_term)
205     # Preventing numerical drift.
206     # ---------------------------
207     # Copying what Randlov did.
208     current_wheelbase = tf.sqrt((xf - xb) ** 2 + (yf - yb) ** 2)
209     relative_error = 1 / current_wheelbase - 1.0
210     xb = tf.where(tf.abs(current_wheelbase - 1) > 0.01, xb + (xb - xf) *
            relative_error, xb)
211     yb = tf.where(tf.abs(current_wheelbase - 1) > 0.01, yb + (yb - yf) *
            relative_error, yb)
212     # Update heading, psi.
213     # --------------------
214     delta_y = yf - yb
215     delta_goal_position = goal_position[:p_batch_size, :]
216     delta_yg = delta_goal_position[:, 1] - yb
217     psi = tf.where(tf.logical_and(xf == xb, delta_y < 0.0), pi,
218                    tf.where((delta_y > 0.0),
219                             tf.atan(safe_divide((xb - xf), delta_y)),
220                             tf.sign(xb - xf) * 0.5 * pi - tf.atan(
                                    safe_divide(delta_y, (xb - xf)))))
221     psig = tf.where(tf.logical_and(xf == xb, delta_yg < 0.0), psi - pi,
222                     tf.where((delta_y > 0.0),
223                              psi - tf.atan(safe_divide((xb -
```

```
                                    delta_goal_position[:, 0]), delta_yg)),
224                                 psi - tf.sign(xb - delta_goal_position[:, 0])
                                        * 0.5 * pi - tf.atan(
225                                     safe_divide(delta_yg, (xb -
                                            delta_goal_position[:, 0])))))
226         omega = tf.reshape(omega, (p_batch_size, 1))
227         omega = tf.where( tf.abs(omega) > pi / 2, tf.math.sign(omega) *pi / 2,
                omega)
228         omega_dot = tf.reshape(omegad, (p_batch_size, 1))
229         omega_ddot = tf.reshape(omegadd, (p_batch_size, 1))
230         theta = tf.reshape(theta, (p_batch_size, 1))
231         theta_dot = tf.reshape(thetad, (p_batch_size, 1))
232         psig = tf.reshape(psig, (p_batch_size, 1))
233         current_pos = tf.concat([tf.reshape(xf, [p_batch_size, 1]), tf.reshape(
                yf, [p_batch_size, 1])], axis=1)
234         pos_d = current_pos - last_pos
235         goal_displacement = delta_goal_position - current_pos
236         goal_dist = tf.sqrt(tf.reduce_sum(tf.pow(goal_displacement, 2)))
237         goal_displacement_normalised = safe_divide(goal_displacement, goal_dist
                )
238         x_d = xf - last_xf
239         y_d = yf - last_yf
240         goal_displacement_x = delta_goal_position[:, 0] - xf
241         goal_displacement_y = delta_goal_position[:, 1] - yf
242         goal_dist = tf.sqrt(tf.pow(goal_displacement_x, 2) + tf.pow(
                goal_displacement_y, 2))
243         goal_displacement_normalised_x = safe_divide(goal_displacement_x,
                goal_dist)  # constructing a unit vector here.
244         goal_displacement_normalised_y = safe_divide(goal_displacement_y,
                goal_dist)
245         if goal:
246             r_t = tf.reduce_sum(pos_d * goal_displacement_normalised, axis=1)
247             r_t = x_d * goal_displacement_normalised_x + y_d *
                    goal_displacement_normalised_y  # this is a dot product
248         else:
249             y_d = yf - last_yf
250             r_t = y_d
251         timestep += 1.
252         x_f = tf.reshape(xf, (p_batch_size, 1))
```

```
253         y_f = tf.reshape(yf, (p_batch_size, 1))
254         x_b = tf.reshape(xb, (p_batch_size, 1))
255         y_b = tf.reshape(yb, (p_batch_size, 1))
256         psi = tf.reshape(psi, (p_batch_size, 1))
257         r_t = tf.reshape(r_t, (p_batch_size, 1))
258         timestep = tf.reshape(timestep, (p_batch_size, 1))
259         trajectories_terminating = timestep >= trajectory_length
260         #trajectories_terminating = tf.logical_or(timestep >= trajectory_length
                , tf.abs(omega) > math.pi/9)
261         trajectories_terminating = tf.reshape(trajectories_terminating, [
                p_batch_size,])
262         new_state = tf.concat([omega, omega_dot, omega_ddot, theta, theta_dot,
                x_f, y_f, x_b, y_b, psi, psig, timestep],
263                               axis=1)
264         penalty_handle = flat_bottomed_barrier_function(tf.abs(theta), 1.3963 *
                0.9, 8)
265         penalty_angle = flat_bottomed_barrier_function(tf.abs(omega), pi / 15,
                8)
266         penalty_psi = 0
267         penalty_psig = 0
268         penalty_psi = flat_bottomed_barrier_function(tf.abs(psig), pi / 2, 8) *
                (1 if with_psi_restriction else 0)
269         psiRemoved = int(not(testt == "psiRemoved"))
270         angleRemoved = int(not(testt == "angleRemoved"))
271         handleRemoved = int(not(testt == "handleRemoved"))
272         recorded_tanh = -tf.tanh(penalty_psi * psiRemoved + penalty_angle*
                angleRemoved + penalty_handle*handleRemoved) + r_t
273         if use_tanh:
274             reward = -tf.tanh(penalty_psi * psiRemoved + penalty_angle*
                    angleRemoved + penalty_handle*handleRemoved) + r_t
275         else:
276             reward = -1 * (penalty_psi * psiRemoved + penalty_angle*
                    angleRemoved + penalty_handle*handleRemoved) + r_t
277         return [reward, recorded_tanh, new_state, trajectories_terminating]
278
279
280 def flat_bottomed_barrier_function(x, k_width, k_power):
281     return tf.pow(tf.maximum(x / (k_width * 0.5) - 1, 0), k_power)
282 def evaluate_final_state(state):
```

```
283     return tf.zeros_like(state[:, 0])
284 #MODEL DESIGN
285 learning_rate = 0.01
286 unroll_pseudo_initial_states_to_truth = True
287 initialise_wrap_around_gradients = True
288 class model(keras.Model):
289     def __init__(self):
290         super(model, self).__init__()
291         self.neural_layers = []
292         for hidden in num_hidden_units:
293             self.neural_layers.append(keras.layers.Dense(hidden, activation
                ="tanh",
294                                                          kernel_initializer
                                                             =keras.
                                                             initializers.
                                                             RandomNormal(
295                                                              stddev=0.001),
296                                                          bias_initializer=
                                                             keras.
                                                             initializers.
                                                             Zeros()))
297         self.neural_layers.append(keras.layers.Dense(action_space, name='
            output', activation="tanh",
298                                                      kernel_initializer=
                                                         keras.initializers.
                                                         RandomNormal(stddev
                                                         =0.001),
299                                                      bias_initializer=keras
                                                         .initializers.Zeros
                                                         ()))
300     @tf.function
301     def call(self, input):
302         x = input
303         for layer in self.neural_layers:
304             y = layer(x)
305             x = tf.concat([x, y], axis=1)
306         return y
307 keras_action_network = model()
308 #if VALIDATION == True:
```

```
309 #     keras_action_network.load_weights("./checkpoints/my_checkpoint")
310 @tf.function
311 def converter(state, passed_batch_size):
312     # omega, omega_dot, omega_ddot, theta, theta_dot, x_f, y_f, x_b, y_b,
        psi,psig, timestep
313     omega = state[:, 0]
314     omega_dot = state[:, 1]
315     theta = state[:, 3]
316     theta_dot = state[:, 4]  # theta - handle bar, omega - angle of bicycle
        to verticle
317     psi = state[:, 9]
318     psig = state[:, 10]
319     psi = tf.reshape(psi, (passed_batch_size, 1))
320     omega = tf.reshape(omega, (passed_batch_size, 1))
321     omega_dot = tf.reshape(omega_dot, (passed_batch_size, 1))
322     theta = tf.reshape(theta, (passed_batch_size, 1))
323     theta_dot = tf.reshape(theta_dot, (passed_batch_size, 1))
324     psig = tf.reshape(psig, (passed_batch_size, 1))
325     omega_visible = tf.tanh(omega * 10)
326     omega_dot = tf.tanh(omega_dot)
327     theta_dot = tf.tanh(theta_dot)
328     theta = tf.tanh(theta / (pi / 4))
329     if goal:
330         converted_state = tf.concat([omega_visible, omega_dot, theta,
            theta_dot, tf.sin(psig), tf.cos(psig)], axis=1)
331     else:
332         converted_state = tf.concat([omega_visible, omega_dot, theta,
            theta_dot, tf.sin(psi), tf.cos(psi)], axis=1)
333     return converted_state
334
335 action_list = []
336 @tf.function
337 def expand_trajectories(start_states):
338     total_rewards = tf.constant(0.0, shape=[batch_size])
339     recorded_total_rewards = tf.constant(0.0, shape= [batch_size])
340     actions = tf.zeros((batch_size, action_space))
341     action_list = []
342     trajectory=tf.expand_dims(start_states,axis=0)
343     trajectories_terminated = tf.cast(tf.zeros_like(start_states[:, 0]), tf
```

```
        .bool)
344     state = start_states
345     print( tf.TensorShape([None, state.get_shape()[0],state_dimension]))
346     print(np.shape(recorded_total_rewards))
347     [state,total_rewards,recorded_total_rewards,trajectory,
            trajectories_terminated]=\
348         tf.while_loop(while_loop_cond, while_loop_body, (state,
                total_rewards,recorded_total_rewards,trajectory,
                trajectories_terminated),
349                       shape_invariants=(state.get_shape(),
350                                         total_rewards.get_shape(),
351                                         recorded_total_rewards.get_shape(),
352                                         tf.TensorShape([None, state.
                                             get_shape()[0],state_dimension])
                                             ,
353                                         trajectories_terminated.get_shape()
                                             ))
354     # build main graph. This is a long graph with unrolled in time for
            trajectory_length steps. Each step includes one neural network
            followed by one physics-model
355     action_history = tf.stack(action_list, axis=0)
356     trajectory = tf.stack(trajectory, axis=0)
357     average_total_reward = tf.reduce_mean(total_rewards)
358     average_total_recorded_reward = tf.reduce_mean(recorded_total_rewards)
359     return [average_total_reward,average_total_recorded_reward, trajectory,
             action_history, trajectories_terminated]
360 opt = keras.optimizers.Adam(learning_rate)
361
362
363 @tf.function
364 def while_loop_cond(state,total_rewards,recorded_total_rewards,trajectory,
        trajectories_terminated):
365     return tf.logical_not(tf.reduce_all(trajectories_terminated))
366 @tf.function
367 def while_loop_body(state,total_rewards,recorded_total_rewards,trajectory,
        trajectories_terminated):
368     converted_state = converter(state, batch_size)
369     prevaction = keras_action_network(converted_state)
370     action = tf.reshape(prevaction, (batch_size, action_space))
```

```
371     [rewards, recorded_tanh, n_state, trajectories_terminating] = step(
            state, action, trajectories_terminated,
372                                                                 batch_size
                                                                        )
373     state = tf.where(tf.expand_dims(trajectories_terminated, 1), state,
            n_state)
374     action_list.append(action_list)
375
376     rewards = tf.reshape(rewards, (batch_size,))
377     recorded_tanh = tf.reshape(recorded_tanh, (batch_size,))
378     total_rewards += tf.where(trajectories_terminated, tf.zeros_like(
            rewards), rewards)
379     total_rewards += tf.where(tf.logical_and(trajectories_terminating, tf.
            logical_not(trajectories_terminated)),
380                               evaluate_final_state(state), tf.zeros_like(
                                      rewards))
381     recorded_total_rewards += tf.where(trajectories_terminated, tf.
            zeros_like(recorded_tanh), recorded_tanh)
382     recorded_total_rewards += tf.where(
383         tf.logical_and(trajectories_terminating, tf.logical_not(
                trajectories_terminated)),
384         evaluate_final_state(state), tf.zeros_like(recorded_tanh))
385
386     trajectories_terminated = tf.logical_or(trajectories_terminated,
            trajectories_terminating)
387     trajectory = tf.concat([trajectory, tf.expand_dims(state, axis=0)],
            axis=0)
388     return state,total_rewards,recorded_total_rewards,trajectory,
            trajectories_terminated
389
390 @tf.function
391 def dolearn(start_states):
392     with tf.GradientTape() as t:
393         t.watch(start_states)
394         [total_reward,recorded_total_reward, trajectory, action_hisotry,
                trajectories_terminated] = expand_trajectories(start_states)
395         cost_ = -tf.reduce_mean(total_reward)
396     grads = t.gradient(cost_, keras_action_network.trainable_weights)
397     return grads, trajectory, total_reward,recorded_total_reward,
```

```
        action_hisotry, trajectories_terminated
398
399 #GRAPHICS
400 def static_graphics():
401     fig, ((ax_omega, ax_theta), (ax_trajectory, ax_reward_history), (
            ax_actionT, ax_psi),
402             (ax_actiond, ax_timestep)) = plt.subplots(nrows=4, ncols=2,
                    figsize=(10, 10))
403     # display.clear_output(wait=True)
404     # fig=plt.figure(figsize=[12.4, 4.8])
405     fig.tight_layout(pad=5.0)
406     # ax_omega = omega
407     pad = 10
408     ax_omega.axis(
409         [0, trajectory_length, (-math.pi / 15) * 180 / math.pi - pad, (math
                .pi / 15) * 180 / math.pi + pad])
410     ax_omega.set(xlabel='timestep', ylabel='Bike Roll value in Degrees')
411     ax_omega.set_title('(Bike Roll).')
412     ax_omega.grid()
413     # ax2 = theta
414     ax_theta.axis([0, trajectory_length, -80 - pad, 80 + pad])
415     ax_theta.set(xlabel='timestep', ylabel='Bike handle value in Degrees')
416     ax_theta.set_title('(Bike Handle).')
417     ax_theta.grid()
418     # ax3 = Agent moving in the field
419     ax_trajectory.axis([-b, b, -b, b])
420     ax_trajectory.set(xlabel='x', ylabel='y')
421     ax_trajectory.set_title('Bike Trajectory.')
422     ax_trajectory.plot(goal_position[:, 0], goal_position[:, 1], color='
            green', marker='o')
423     ax_trajectory.grid()
424     # ax4 = reward over time.
425     ax_reward_history.axis([0, max_iterations, -1 - 0.5, 1 + 0.5])
426     ax_reward_history.set(xlabel='Iteration', ylabel='Reward')
427     ax_reward_history.set_title('Reward over Iteration')
428     for traj in range(batch_size):
429         trajectory_x_coord = [0]  # trajectory[:,traj,0]
430         ax_reward_history.plot(trajectory_x_coord)
431     ax_reward_history.grid()
```

```
432         # ax5 = action the agent takes
433         ax_actionT.axis([0, trajectory_length, -2 - 0.5, 2 + 0.5])
434         ax_actionT.legend(loc="upper right")
435         ax_actionT.set(xlabel='timestep', ylabel='torque')
436         ax_actionT.set_title('Trajectory torque')
437         ax_actionT.grid()
438         # ax6 = agent psi
439         ax_psi.axis([0, trajectory_length, -180 - pad, 180 + pad])
440         ax_psi.set(xlabel='timestep', ylabel='psi')
441         ax_psi.set_title('Bike direction Psi')
442         ax_psi.grid()
443         # ax7 = agent d
444         ax_actiond.axis([0, trajectory_length, -maximum_dis - 0.01, maximum_dis
                + 0.01])
445         ax_actiond.set(xlabel='timestep', ylabel='displacement')
446         ax_actiond.set_title('The centre of mass displacement')
447         ax_actiond.grid()
448         # ax8 timestep
449         ax_timestep.axis([0, max_iterations, 0 - pad, trajectory_length + pad])
450         ax_timestep.set_title('max balancing duration')
451         ax_timestep.set(xlabel='iteration', ylabel='time steps')
452         ax_timestep.grid()
453         plt.draw()
454         plt.pause(0.001)
455         return [fig, ax_omega, ax_theta, ax_trajectory, ax_reward_history,
                ax_actionT, ax_psi, ax_actiond, ax_timestep]
456     def dynamic_graphics(trajectory, stats, passed_actions,reward_history,
            timestep_history):
457         # omega, omega_dot, omega_ddot, theta, theta_dot, x_f, y_f, x_b, y_b,
                psi,psig, timestep
458         fig, ax_omega, ax_theta, ax_trajectory, ax_reward_history, ax_actionT,
                ax_psi, ax_actiond, ax_timestep = stats
459         T = passed_actions[:, :, 0] * maximum_torque
460         T = tf.where(T > maximum_torque, tf.ones_like(T) * maximum_torque, T)
461         T = tf.where(T < -maximum_torque, tf.ones_like(T) * -maximum_torque, T)
462         d = passed_actions[:, :, 1] * maximum_dis
463         d = tf.where(d > maximum_dis, tf.ones_like(d) * maximum_dis, d)
464         d = tf.where(d < -maximum_dis, tf.ones_like(d) * -maximum_dis, d)
465         omega = trajectory[:, :, 0]
```

```
466     omega_dot = trajectory[:, :, 1]
467     omega_ddot = trajectory[:, :, 2]
468     theta = trajectory[:, :, 3]
469     theta_dot = trajectory[:, :, 4]  # theta - handle bar, omega - angle of
            bicycle to verticle
470     x_f = trajectory[:, :, 5]
471     y_f = trajectory[:, :, 6]
472     x_b = trajectory[:, :, 7]
473     y_b = trajectory[:, :, 8]
474     psi = trajectory[:, :, 9]
475     time_steps = trajectory[:, :, -1]
476     col = 0
477     ax_omega.lines.clear()
478     ax_theta.lines.clear()
479     ax_trajectory.lines.clear()
480     ax_reward_history.lines.clear()
481     ax_actionT.lines.clear()
482     ax_psi.lines.clear()
483     ax_actiond.lines.clear()
484     ax_timestep.lines.clear()
485     for trajectory_number in range(batch_size):
486         if trajectory[0, trajectory_number, -1] == 0:
487             col = (col + 1) % len(colors)
488         conv_omega = omega[:, trajectory_number] * 180 / np.pi
489         conv_theta = theta[:, trajectory_number] * 180 / np.pi
490         conv_psi = psi[:, trajectory_number] * 180 / np.pi
491         ax_omega.plot(time_steps[:, trajectory_number], conv_omega, color=
                colors[col])
492         ax_theta.plot(time_steps[:, trajectory_number], conv_theta, color=
                colors[col])
493         x = x_f[:, trajectory_number]
494         y = y_f[:, trajectory_number]
495         ax_trajectory.plot(x, y, color=colors[col])
496         ax_actionT.plot(time_steps[1:, trajectory_number], T[:,
                trajectory_number], color=colors[col])
497         ax_actiond.plot(time_steps[1:, trajectory_number], d[:,
                trajectory_number], color=colors[col])
498         ax_psi.plot(time_steps[:, trajectory_number], conv_psi, color=
                colors[col])
```

```
499         ax_trajectory.plot(goal_position[0, 0], goal_position[0, 1], color='b',
                marker='o')
500         ax_reward_history.axis([0, max_iterations, min(reward_history) - 10,
                max(reward_history) + 10])
501         ax_reward_history.plot(reward_history, color='red')
502         ax_timestep.plot(timestep_history, color='red')
503         plt.draw()
504         plt.pause(0.001)
505     if graphical:
506         stat = static_graphics()
507
508     #TRAINING
509     timestep_history = []
510     action_history = []
511     reward_history = []
512     trajectory_history = []
513     initial_state_backup = initial_state
514     trajectories_terminated = tf.cast(tf.zeros_like(initial_state[:, 0]), tf.
            bool)
515     t_a = datetime.now()
516     t_b = datetime.now()
517     final_artificial_gradient = np.zeros_like(initial_state)
518     print(batch_size)
519     for iteration in range(max_iterations):
520         state1 = initial_state_backup
521         # trajectories_terminated = trajectories_terminated
522         gradients_list, trajectory, total_reward,rc_reward, actions,
                trajectories_terminated = dolearn(state1)
523         trajectory = trajectory.numpy()
524         trajectories_terminated = trajectories_terminated.numpy()
525         #dReward_dInputState = dReward_dInputState.numpy()
526         for _ in gradients_list:
527             if np.isnan(_).any():
528                 print("Nan Grads")
529         opt.apply_gradients(zip(gradients_list, keras_action_network.
                trainable_weights))
530         # total_reward = total_reward + (-1 *(trajectory_length - trajectory
                [-1,:,-1]))
531         average_total_reward_stepwise = np.max(total_reward.numpy())
```

```
532     rc_average_total_reward_stepwise = np.max(rc_reward.numpy())
533     reward_history.append(rc_average_total_reward_stepwise)
534
535     timestep_history.append(np.max(trajectory[:, :,-1]))
536     final_trajectory_steps = trajectory[-1, :, :]
537     #action_history.append(actions)
538     #trajectory_history.append(trajectory)
539     if iteration % print_time == 0:
540         if graphical:
541             dynamic_graphics(trajectory, stat, actions)
542         if prinit:
543             t_b = datetime.now()
544             dt = t_b - t_a
545             print("iteration: ", iteration, "//
                    Average_total_reward_step_wise: ",
                    rc_average_total_reward_stepwise,
546                   "Average Step: ", np.mean(trajectory[-1, :, -1]), "in
                        steps and ",
547                   np.mean(trajectory[-1, :, -1]) * delta_time, "in seconds"
                        , "time taken from last iter: ",
548                   diff(t_a, t_b))
549             t_a = t_b
550
551     if save:
552         if iteration % 10 == 0:
553             save_mat = np.concatenate([[reward_history], [timestep_history
                    ]], axis=0)
554             np.save("withoutClipping/" + trial_name + "_marker_" + str(
                    iteration) + "_results_" + filename + ".npy",save_mat)
555             #np.save("runs5/" + trial_name + "_marker_" + str(iteration) +
                    "_action_history_" + filename + ".npy",np.array(
                    action_history))
556             #np.save("runs5/" + trial_name + "_marker_" + str(iteration) +
                    "_trajectory_history_" + filename + ".npy",np.array(
                    trajectory_history))
557             np.save("withoutClipping/last_state_" + filename + ".npy",
                    trajectory)
558
559             action_history = []
```

```
560                 reward_history = []
561                 timestep_history = []
562                 trajectory_history = []
563             if iteration % 10 == 0:
564                 keras_action_network.save_weights("./checkpoints/my_checkpoint"
                        +trial_name)
565 if save:
566     np.save("withoutClipping/" + trial_name + "_marker_" + str(iteration) +
            "_results_" + filename + ".npy",save_mat)
567     #np.save("withoutClipping/" + trial_name + "_marker_" + str(iteration)
            + "_trajectory_history_" + filename + ".npy",np.array(
            trajectory_history))
568     np.save("withoutClipping/last_state_" + filename + ".npy", trajectory)
569     keras_action_network.save_weights("./checkpoints/my_checkpoint_"+
            trial_name)
```

Appendix 3: A Simulated 2D Navigational Agent

This appendix provides the source code for a simulated 2D navigational agent. The code is divided into three main sections:

1. **Brain of the Simulated 2D Navigational Agent**: This section contains the core logic that governs the agent's behaviour.

2. **Environment of the Simulated 2D Navigational Agent**: This section outlines the environmental parameters and variables.

3. **Experiment of the Simulated 2D Navigational Agent**: This section is devoted to the experimental setup where the agent operates.

Brain of the Simulated 2D Navigational Agent

Listing 7.3: Brain of the Simulated 2D Navigational Agent implemented in Python.

```
1 import numpy as np
2 import tensorflow as tf
3 import matplotlib.pyplot as plt
4 from matplotlib import cm
```

```
5  from mpl_toolkits.mplot3d import Axes3D
6  from tensorflow import keras
7  from tensorflow.keras import layers
8  from IPython import display
9  import argparse
10 import wandb
11 from matplotlib.lines import Line2D
12 import random
13 import matplotlib.lines as mlines
14 import matplotlib.patches as mpatches
15 #added new
16 class AntBrain(keras.Model):
17     def __init__(self, num_layers, action_network_num_outputs):
18         super(AntBrain, self).__init__()
19         self.layer1=layers.Dense(num_layers, activation='tanh')
20         #self.layer2=layers.Dense(num_layers, activation='tanh')
21         self.output_layer=layers.Dense(action_network_num_outputs,
               activation=None)
22
23     @tf.function
24     def call(self, input_vector):
25         x=input_vector
26         y = self.layer1(x)
27         #print(y)
28         #y2=self.layer2(y)
29         #x=tf.concat([x,y], axis=1)# This adds shortcut connections from
               the previous layer to the next layer
30         #y3=self.layer2(y2)
31         #x=tf.concat([x,y], axis=1)# More shortcut connections.
32         y4=self.output_layer(y)
33         # Using the shortcut connections above means I don't need to worry
34         # too much about how many hidden layers to add.  For example, if
               hidden
35         # layers 1 and 2 are not needed then they can simply be skipped
               over.
36         # Also it ensures there are shortcut connections from the input
               layer to the final layer, which
37         # potentially allows memories i
38         return y4
```

Environment of the Simulated 2D Navigational Agent

Listing 7.4: Environment of the Simulated 2D Navigational Agent implemented in Python.

```
1  import sys
2
3  import numpy as np
4  import tensorflow as tf
5  import matplotlib.pyplot as plt
6  from matplotlib import cm
7  from mpl_toolkits.mplot3d import Axes3D
8  from tensorflow import keras
9  from tensorflow.keras import layers
10 from IPython import display
11 import argparse
12 import wandb
13 from matplotlib.lines import Line2D
14 import random
15 import matplotlib.lines as mlines
16 import matplotlib.patches as mpatches
17 class Environment:
18     def __init__(self, batch_size, randomise_food_heights,
           randomise_food_location, state_dimension, use_sensor,
           num_memory_nodes,type_mem):
19         self.randomise_food_location = randomise_food_location
20         self.type_mem = type_mem
21         self.use_sensor = use_sensor
22         self.batch_size = batch_size
23         self.randomise_food_heights = randomise_food_heights
24         self.num_memory_nodes = num_memory_nodes
25         self.food_location = tf.constant( (np.random.rand(batch_size,3)
               -0.5) *(np.array(
26             [8,8,randomise_food_heights]) if randomise_food_location else
                   np.array([0,0,0])), tf.float32)
27         self.food_location_validation = tf.constant((np.random.rand(
               batch_size, 3) - 0.5) * (
28             np.array([8, 8, randomise_food_heights]) if
                   randomise_food_location else np.array([0, 0, 0])), tf.
                   float32)
29         self.initial_state = tf.concat(
```

```
30              [tf.constant((np.random.rand(batch_size, 2) - 0.5) * (0 if
                    randomise_food_location else 8), tf.float32),
31               tf.zeros([batch_size, state_dimension - 2], tf.float32)], axis
                    =1)
32          self.initial_state_validation = tf.concat(
33              [tf.constant((np.random.rand(batch_size, 2) - 0.5) * (0 if
                    randomise_food_location else 8), tf.float32),
34               tf.zeros([batch_size, state_dimension - 2], tf.float32)], axis
                    =1)
35
36      def food_density(self,xy_position, food_location):
37          # Define a gaussian bump:
38          bump_width = 8
39          result = tf.exp(tf.reduce_sum(-(xy_position - food_location[:, :2])
                ** 2, axis=1) / bump_width)
40          result = result + food_location[:, 2]
41          return result
42
43      def plot_food_pile(self,axes,food_location):
44          X = np.linspace(-5, 5)
45          Y = np.linspace(-5, 5)
46          X, Y = np.meshgrid(X, Y)
47          xy_grid = np.stack([X, Y], axis=2).reshape((-1, 2))
48          Z = tf.reshape(self.food_density(xy_grid, food_location[0:1, :]),
                [50, 50])
49          axes.plot_surface(X, Y, Z, rstride=3, cstride=3, linewidth=1,
                antialiased=True, cmap=cm.viridis, alpha=0.3)
50          cset = axes.contourf(X, Y, Z, zdir='z', offset=0, colors=['#808080'
                , '#A0A0A0', '#C0C0C0'], alpha=0.4)
51          # limits ticks and view angle
52          axes.set_zlim(-0.5, 1.2)
53          axes.set_zticks(np.linspace(0, 1, 5))
54          axes.view_init(27, -21)
55
56      def sensor_calculation(self,pos, food_location):
57          sensor_result = self.food_density(pos, food_location)
58          sensor_result = tf.reshape(sensor_result, [self.batch_size, 1])  #
                reshape it to a rank-2 tensor
59          return sensor_result
```

```
60      @tf.function
61      def run_one_step_of_physics_model(self,state, action, food_location,
            state_sensor_pointer,state_memory_pointer,state_memory_pointer_h,
            state_memory_pointer_c):
62          '''
63          State
64          '''
65          pos_xy = state[:, 0:2]
66          if self.use_sensor:
67              sensor = tf.reshape(state[:, state_sensor_pointer], (self.
                    batch_size, int(self.use_sensor)))
68          old_memory_state = state[:, state_memory_pointer:]
69          if self.type_mem == "Full_LSTM":
70              h = state[:, state_memory_pointer_h:state_memory_pointer_c]
71              c = state[:, state_memory_pointer_c:]
72          '''
73          Action
74          '''
75          vel_xy = action[:, 0:2]
76          # Note this assumes there is no tanh on the final layer!
77          vel_xy_magnitude = tf.sqrt(tf.reduce_sum(tf.square(vel_xy), axis=1)
                )
78          vel_xy_magnitude = tf.expand_dims(vel_xy_magnitude, 1)
79          vel_xy_normalised = vel_xy / (vel_xy_magnitude + 1e-6)
80          vel_xy = vel_xy_normalised * tf.tanh(vel_xy_magnitude)
81          next_pos_xy = pos_xy + vel_xy * 0.2
82          next_state_list = [next_pos_xy]
83          # Note for sensor functionality we have the sensor_calculation
                function above to call upon.
84          # use next_state_list.append(...) to add new chunks of the tensor
                you are building up
85          if self.use_sensor:
86              next_state_list.append(self.sensor_calculation(next_pos_xy,
                    food_location))
87          if self.num_memory_nodes > 0:
88              action_memory = action[:, 2:]
89
90              if self.type_mem == "Minimal_GRU":
91                  # A simplified version of the GRU
```

```
92                  input_gate , new_value = tf.split(value=action_memory ,
93                                                    num_or_size_splits=2, axis
                                                        =1)
94                  # output_gate=-input_gate
95                  # forget_bias_tensor = 1.0
96                  input_gate = tf.sigmoid(input_gate)
97                  # hidden_c = ((hidden_c * tf.sigmoid(forget_gate+
                        forget_bias_tensor)) +
98                  #           (tf.sigmoid(input_gate) * tf.tanh(new_value)))
99                  action_memory = (tf.tanh(new_value)*input_gate +
                        old_memory_state * (1 - input_gate))
100
101             elif self.type_mem == "Full_GRU":
102                 #matrix_x , matrix_inner = tf.split(value=action_memory ,
                        num_or_size_splits=2, axis=1)
103                 #x_z, x_r, x_h = tf.split(matrix_x ,3 ,axis=1)
104                 #recurrent_z, recurrent_r, recurrent_h = tf.split(
                        matrix_inner, 3, axis=1)
105                 #z = tf.sigmoid(x_z + recurrent_z)
106                 #r = tf.sigmoid(x_r + recurrent_r)
107                 #hh = tf.tanh(x_h + r * recurrent_h)
108                 #action_memory = z *tf.tanh(old_memory_state)+ hh * (1 - z)
109                 x_z,x_r,x_h = tf.split(value=action_memory ,
                        num_or_size_splits=3, axis=1)
110                 z = tf.sigmoid(x_z)
111                 r = tf.sigmoid(x_r)
112                 hh = tf.tanh(x_h*r)
113                 action_memory = (1-z) *old_memory_state + z*hh
114             elif self.type_mem == "Full_LSTM":
115                 #state_vector  actionmemory,c -> action memory is shown to
                        neural network
116                 input_gate ,forget_gate ,output_gate ,new_value = tf.split(
                        action_memory ,num_or_size_splits=4,axis=1)
117                 i = tf.sigmoid(input_gate)
118                 f = tf.sigmoid(forget_gate)
119                 c = f * c + i * tf.tanh(new_value)
120                 o = tf.sigmoid(output_gate)
121                 h = o * tf.tanh(c)
122                 action_memory = tf.concat([h,c],axis=1)
```

```
123             elif self.type_mem == "CARU":
124                 x , a, b = tf.split(action_memory,num_or_size_splits=3,axis
                        =1)
125                 n = tf.tanh(a)
126                 l = tf.sigmoid(x)*tf.sigmoid(b)
127                 action_memory = (1-l)*old_memory_state + l *n
128             else:
129                 action_memory = tf.tanh(action_memory)
130             next_state_list.append(action_memory)
131         # END of code block in which to insert lines for CHALLENGE 2
132         next_state = tf.concat(next_state_list,
133                                axis=1)  # appends the rank-2 tensors in
                                    next_state_list side-by-side into one
                                    rank-2 tensor
134
135         rewards = self.food_density(next_state[:, 0:2], food_location)
136         return [rewards, next_state]
137
138     def add_arrow_to_line2D( self,axes, line, arrow_locs=[0.2, 0.4, 0.6,
            0.8], arrowstyle='-|>', arrowsize=1, transform=None):
139         if not isinstance(line, mlines.Line2D):
140             raise ValueError("expected a matplotlib.lines.Line2D object")
141         x, y = line.get_xdata(), line.get_ydata()
142         arrow_kw = {
143             "arrowstyle": arrowstyle,
144             "mutation_scale": 10 * arrowsize,
145         }
146         color = line.get_color()
147         use_multicolor_lines = isinstance(color, np.ndarray)
148         if use_multicolor_lines:
149             raise NotImplementedError("multicolor lines not supported")
150         else:
151             arrow_kw['color'] = color
152         linewidth = line.get_linewidth()
153         if isinstance(linewidth, np.ndarray):
154             raise NotImplementedError("multiwidth lines not supported")
155         else:
156             arrow_kw['linewidth'] = linewidth
157         if transform is None:
```

```
158             transform = axes.transData
159         arrows = []
160         for loc in arrow_locs:
161             s = np.cumsum(np.sqrt(np.diff(x) ** 2 + np.diff(y) ** 2))
162             n = np.searchsorted(s, s[-1] * loc)
163             arrow_tail = (x[n], y[n])
164             arrow_head = (np.mean(x[n:n + 2]), np.mean(y[n:n + 2]))
165             p = mpatches.FancyArrowPatch(
166                 arrow_tail, arrow_head, transform=transform,
167                 **arrow_kw)
168             axes.add_patch(p)
169             arrows.append(p)
170         return arrows
171
172     def show_trajectories(self,trajectories, initial_state, food_location,
        iteration_number, reward, fig0):
173         bs = self.batch_size
174         reward = np.mean(reward)
175         if trajectories.shape[1] > 10:
176             bs = 10
177         if fig0 != None:
178             plt.close(fig0)
179         display.clear_output(wait=True)
180         fig = plt.figure(figsize=[5, 5])
181         axes_2d = fig.add_subplot(1, 1, 1)
182         axes_2d.axis([-6, 6, -6, 6])
183         colcycle = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd',
            '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22',
184                     '#17becf']
185         for traj in range(bs):
186             axes_2d.scatter(food_location[traj, 0], food_location[traj, 1],
                marker="x",
187                             c=colcycle[traj % len(colcycle)])
188             axes_2d.scatter(initial_state[traj, 0], initial_state[traj, 1],
                marker="o",
189                             c=colcycle[traj % len(colcycle)])
190         for traj in range(bs):
191             trajectory_x = trajectories[:, traj, 0]
192             trajectory_y = trajectories[:, traj, 1]
```

```
193             lines, = axes_2d.plot(trajectory_x, trajectory_y, '-', label='
                    Traj1', c=colcycle[traj % len(colcycle)])
194 
195             self.add_arrow_to_line2D(axes_2d, lines, arrow_locs=np.array
                    ([0.5
196                                                                         ]),
197                                      arrowstyle='->', arrowsize=1.2)
198         axes_2d.grid(True)
199         if self.randomise_food_location == 0:
200             axes_2d.set_title('Top-Down view. Food pile fixed at (0,0).')
201         else:
202             axes_2d.set_title('Top-Down view.  Agent starting location
                    fixed at (0,0).')
203         '''
204         if not(randomise_food_location):
205             # since there is only one food location, we can do a 3d plot
                    too
206             axes_3d=fig.add_subplot(1,2, 2,projection='3d')
207             plot_food_pile(axes_3d)
208             for traj in range(bs):
209                 trajectory_x = trajectories[:, traj, 0]
210                 trajectory_y = trajectories[:, traj, 1]
211                 tZ=food_density(trajectories[:, traj, 0:2],food_location[
                        traj:traj+1, :])
212                 axes_3d.plot(trajectory_x, trajectory_x,tZ , c=colcycle[
                        traj%len(colcycle)])
213 
214             axes_3d.set_title('3d view')
215         '''
216         if fig0 != None:
217             display.display(plt.gcf())
218         return fig
219     def rerandomise_start(self):
220 
221         self.food_location = tf.constant( (np.random.rand(self.batch_size
                ,3)-0.5) *(np.array([8,8,self.randomise_food_heights]) if self.
                randomise_food_location else np.array([0,0,0])), tf.float32)
```

Experiment of the Simulated 2D Navigational Agent

Listing 7.5: Experimental setup for the Simulated 2D Navigational Agent implemented in Python.

```
import sys

import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
from tensorflow import keras
from tensorflow.keras import layers
from IPython import display
import argparse
import progressbar
recordwandb = True
if recordwandb:
    import wandb
from matplotlib.lines import Line2D
from GaussianAntEnvironment import Environment
from Brain import AntBrain
import random
import matplotlib.lines as mlines
import matplotlib.patches as mpatches
#experiment
parser = argparse.ArgumentParser()
parser.add_argument('--sensor_is_directable', type=int, default=0)     #use
    for network to control the sensors
parser.add_argument('--action_network_sees_xy_position', type=int, default
    =0) #add x,y cordinates to the input
parser.add_argument('--sees_highest_points', type=int, default=0)   #add
    hgihest coordinate to the input
parser.add_argument('--sees_direction_to_highest', type=int, default=0) #
    compass method
parser.add_argument('--sees_timestep', type = int, default=0)   #add
    timestep to the input
parser.add_argument('--sensor_distance', type=int, default=0)   #sensor
    will have a distance from the agent
parser.add_argument('--slow_start', type=int, default=0)    #add random
```

```
    uniform with fixed value at the intialisation of network
31
32 parser.add_argument('--n_sensors', type= int, default=1)    #number of
    sensors used 1 is suggested
33 parser.add_argument('--num_food', type=int, default=1) #number of goals in
     the same map
34 parser.add_argument('--randomised_height', type = int, default = 1) #
    randomise the height of the food for each batch
35 parser.add_argument('--num_memory_nodes', type = int, default= 20)   #
    number of memory nodes used to input
36 parser.add_argument('--randomised_food_location', type= int, default=1) #1:
     agent starts at (0,0) 0: agents scattered around 0,0
37 parser.add_argument('--memory_mod', type= str, default="identity") #
    Minimal_GRU, CARU, Full_LSTM
38 parser.add_argument('--randomise_start_trajectory', type= int, default=1) #
    randomise food or the starting pos at the start of each iteration.
39 parser.add_argument('--batch_size', type=int, default=100)
40 parser.add_argument('--num_layers', type = int, default=20)
41 parser.add_argument('--trajectory_length', type=int,default=30)
42 parser.add_argument('--max_iteration', type=int,default=100000)
43 parser.add_argument('--learning_rate', type=float,default=0.001)
44 parser.add_argument('--render', type=bool, default=False)
45 parser.add_argument('--printit', type=bool, default=False)
46 parser.add_argument('--record', type=bool, default=True)
47 parser.add_argument('--algo_name', type=str, default="BPTT")
48 parser.add_argument('--trial', type=int, default=1)
49
50 args = parser.parse_args()
51 num_layers = int(args.num_layers)
52 max_iteration = int(args.max_iteration)
53 learning_rate = float(args.learning_rate)
54 randomise_start_trajectory = bool(args.randomise_start_trajectory)
55 memory_mod = str(args.memory_mod)
56 num_memory_nodes = int(args.num_memory_nodes)
57 randomised_food_location = bool(args.randomised_food_location)
58 sees_timestep = bool(args.sees_timestep)
59 trajectory_length= int(args.trajectory_length)
60 randomised_height = bool(args.randomised_height)
61 num_food = int(args.num_food)
```

```
62 batch_size = int(args.batch_size)
63 sensor_is_directable = bool(args.sensor_is_directable)
64 action_network_sees_xy_position = bool(args.action_network_sees_xy_position
       )
65 sees_highest_points = bool(args.sees_highest_points)
66 algo_name = str(args.algo_name)
67 sees_direction_to_highest = bool(args.sees_direction_to_highest)
68 slow_start = bool(args.slow_start)
69 sensor_distance = int(args.sensor_distance)
70 number_of_sensors = int(args.n_sensors)
71 render = bool(args.render)
72 printit =  bool(args.printit)
73 record = bool(args.record)
74 trial = int(args.trial)
75 state_dimension = 2 + (1 if number_of_sensors >0 else 0)
76 simp_gru = 0
77 full_gru= 0
78 full_lstm = 0
79 CARU = 0
80 original_n_memories = num_memory_nodes
81 if memory_mod == "Minimal_GRU":
82     simp_gru = 1
83     num_memory_nodes =  num_memory_nodes * 2  # vx, vy
84 elif memory_mod == "Full_GRU":
85     full_gru =1
86     num_memory_nodes =  num_memory_nodes * 3
87 elif memory_mod == "Full_LSTM":
88     full_lstm = 1
89     num_memory_nodes = num_memory_nodes *4
90 elif memory_mod == "CARU":
91     CARU = 1
92     num_memory_nodes =  num_memory_nodes *3
93 else:
94     num_memory_nodes =  num_memory_nodes
95 action_network_num_outputs = 2 + (2 if sensor_is_directable else 0) +
       num_memory_nodes #x, y, sx, sy, num_mem
96 action_network_num_inputs = (2 if action_network_sees_xy_position else 0)+
       number_of_sensors+num_memory_nodes  # x, y
97 #pointers#
```

```
98  #statedimention 0x,1y,2sensor,memory,3timestep
99  #action 0velx,1vely,2memory
100 use_sensor = False
101 use_memory = False
102 if number_of_sensors >=1:
103     use_sensor = True
104 if num_memory_nodes >=1:
105     use_memory = True
106 state_memory_pointer = 1 + (1 if use_sensor else 0) + (1 if use_memory else
        0)
107 state_memory_pointer_h = state_memory_pointer
108 state_memory_pointer_c = state_memory_pointer+ original_n_memories
109 state_sensor_pointer = 1+ (1 if use_sensor else 0)
110 action_memory_pointer = 2
111 state_dimension +=  original_n_memories + (original_n_memories if
    memory_mod == "Full_LSTM"  else 0) # x,y, timestep, hidden vector memory
112
113 print("action_network_num_inputs",action_network_num_inputs )
114 print("action_network_num_outputs",action_network_num_outputs)
115 print("state_memory_pointer",state_memory_pointer)
116 print("state_memory_pointer_h", state_memory_pointer_h)
117 print("state_memory_pointer_c",state_memory_pointer_c)
118 print("state_sensor_pointer",state_sensor_pointer)
119 print("action_memory_pointer",action_memory_pointer)
120 print("state_dimension",state_dimension)
121
122 filename = "Trial_"+ str(trial)+"_Sensors_"+str(number_of_sensors)+"
    _Memories_"+ str(num_memory_nodes)+"_TypeMemory_"+memory_mod+"
    _randomisedfood_"+str(randomised_food_location)+"_randomisedheight_"+str
    (randomised_height)+"_"+"BPTTrunANT"+"_"
123 if recordwandb:
124     wandb.init(name=algo_name, project="memory_modification_experiment",
          group=algo_name,
125                 config={
126                     "randomised_heights": randomised_height,
127                     "memory_modification_type" : memory_mod ,
128                     "sensor_distance" : 0,
129                     "dimenstional_information" : state_dimension,
130                     "number_of_sensors" : number_of_sensors,
```

```
131                     "num_memory_nodes" : num_memory_nodes,
132                     "sensor_is_directable" : sensor_is_directable,
133                     "action_network_sees_xy_position" :
                            action_network_sees_xy_position,
134                     "randomised_food_location" : randomised_food_location,
135                     "sees_highest_points" : sees_highest_points,
136                     "sees_direction_to_highest" : sees_direction_to_highest,
137                     "neural_layers" :[12,12],
138                     "num_food" : num_food,
139                     "batch_size" : batch_size,
140                     "trajectory_length" : trajectory_length,
141                     "max_iteration" : max_iteration,
142                     "slow_start" : slow_start,
143                     "learning_rate" : learning_rate,
144                     "randomise_start_of_iterations":
                            randomise_start_trajectory,
145                     "shortcuts": False
146                 }
147                 )
148
149 fig = plt.figure()
150 axes = fig.add_subplot(1,1, 1,projection='3d')
151
152
153
154 environment = Environment(batch_size,randomised_height,
        randomised_food_location,state_dimension,number_of_sensors,
        num_memory_nodes,memory_mod)
155 keras_ant_brain = AntBrain(num_layers,action_network_num_outputs)
156
157 def convert_state(state):
158     pos_xy = state[:, 0:2]
159     sensor = tf.reshape(state[:,state_sensor_pointer], (batch_size,
            number_of_sensors))
160     old_memory_state = state[:, state_memory_pointer:]
161     h = state[:,state_memory_pointer_h:state_memory_pointer_c]
162     #input_gate, forget_gate, output_gate, h = tf.split(h,
            num_or_size_splits=4, axis=1)
163     c = state[:,state_memory_pointer_c:]
```

```
164     converted_state = tf.concat([sensor,h], axis=1)
165     return converted_state
166 @tf.function
167 def expand_full_trajectory(start_states, food_location):
168     total_rewards=tf.constant(0.0, dtype=tf.float32, shape=[batch_size])
169     state=start_states # this is shape [batch_size, state_dimension]
170     trajectory_list=[start_states]
171     # build main graph.  This is a long graph with unrolled in time for
            trajectory_length steps.  Each step includes one neural network
            followed by one physics-model
172     for time_step in range(trajectory_length):
173         action = keras_ant_brain(convert_state(state))
174         [rewards,state]=environment.run_one_step_of_physics_model(state,
                action,food_location,state_sensor_pointer,state_memory_pointer,
                state_memory_pointer_h,state_memory_pointer_c)
175         total_rewards+=rewards # This is shape [batch_size]
176         trajectory_list.append(state)
177
178     trajectories=tf.stack(trajectory_list) # This will be shape [batch_size
            , trajectory_length+1, state_dimension]
179     average_total_reward=tf.reduce_mean(total_rewards) # this is a scalar
180     return [average_total_reward,trajectories]
181
182 [average_total_reward,trajectories] = expand_full_trajectory( environment.
        initial_state, environment.food_location)
183 fig=None
184 fig=environment.show_trajectories(trajectories, environment.initial_state,
        environment.food_location,0, average_total_reward.numpy(), fig)
185 reward_history=[] # Keep a log for plotting training history
186 reward_history_validation=[] # Keep a log for plotting training history
187 reward_history_iters=[] # Keep a log for plotting training history
188 reward_history_iters_validation=[] # Keep a log for plotting training
        history
189
190 optimizer = keras.optimizers.Adam(learning_rate)
191
192 @tf.function
193 def run_exp(initial_state, food_location):
194     with tf.GradientTape() as tape:
```

```
195             # Run the forward pass of the layer.
196             # The operations that the layer applies
197             # to its inputs are going to be recorded
198             # on the GradientTape.
199             [average_total_reward,trajectories] = expand_full_trajectory(
                    initial_state, food_location)
200             loss=-average_total_reward
201         grads = tape.gradient(loss,keras_ant_brain.trainable_weights)  # The "
            back-propagation through time" calculation is the computation of
            this gradient
202         return loss, average_total_reward, trajectories, grads
203
204 if printit:
205     widgets = [progressbar.Percentage(), ' ', progressbar.Bar('=', '[', ']'
        ),' ', progressbar.Timer(),' ',progressbar.FormatLabel('')]
206     bar = progressbar.ProgressBar(maxval=max_iteration,widgets=widgets)
207     bar.start()
208 for i in range(1,max_iteration):
209     iteration=len(reward_history)
210     if randomise_start_trajectory:
211         environment.rerandomise_start()
212     loss, average_total_reward , trajectories,grads = run_exp(environment.
        initial_state, environment.food_location)
213     # Use the gradient tape to automatically retrieve
214     # the gradients of the trainable variables with respect to the loss.
215     # Run one step of gradient descent by updating
216     # the value of the variables to minimize the loss.
217     optimizer.apply_gradients(zip(grads, keras_ant_brain.trainable_weights)
        )
218     if np.any(np.isnan(trajectories.numpy())):
219         print("trajectory",trajectories)
220         raise Exception("Trajectories is Nan")
221     average_total_reward=average_total_reward.numpy()
222     reward_history.append(average_total_reward)
223     reward_history_iters.append(iteration)
224     if recordwandb:
225         wandb.log({"global_step": i,"Training mean Trajectory Reward":
                average_total_reward})
226
```

```
227     if i%100==0:
228         [average_total_reward_validation, validation_trajectory] =
                expand_full_trajectory(environment.initial_state_validation,
                environment.food_location_validation)
229         reward_history_validation.append(average_total_reward_validation)
230         reward_history_iters_validation.append(iteration)
231         if printit:
232             p = "global_step: " + str(i) + " eval/mean_reward: " + str(
                    average_total_reward_validation.numpy())
233             widgets[-1] = progressbar.FormatLabel(p.format(i))
234             bar.update(i)
235             #print()
236         if recordwandb:
237             wandb.log({"global_step": i, "eval/mean_reward":
                    average_total_reward_validation})
238     if render:
239         if (len(reward_history)%40)==0:
240             fig=environment.show_trajectories(trajectories, environment.
                    initial_state, environment.food_location, len(reward_history
                    ), average_total_reward, fig)
241
242
243 if record:
244     fig = environment.show_trajectories(trajectories, environment.
            initial_state, environment.food_location, len(reward_history),
            average_total_reward, fig)
245     plt.savefig(filename+"wandb_upload_train.jpg")
246     if recordwandb:
247         im = plt.imread(filename+"wandb_upload_train.jpg")
248         wandb.log({"train": [wandb.Image(im, caption="Last iteration
                results")]})
249     np.save(filename+"trajectories.npy",validation_trajectory)
250     fig = environment.show_trajectories(validation_trajectory, environment.
            initial_state_validation, environment.food_location_validation, len(
            reward_history_validation), average_total_reward_validation, fig)
251     plt.savefig(filename+"wandb_upload_valid.jpg")
252     plt.savefig(filename+"save1.pdf")
253     if recordwandb:
254         im = plt.imread(filename+"wandb_upload_valid.jpg")
```

```
255             wandb.log({"validation": [wandb.Image(im, caption="Last iteration
                    results")]})
256         fig, ax = plt.subplots()
257         ax.set(xlabel='Training Iteration', ylabel='Reward', title='Reward
                History')
258         ax.grid(True)
259         ax.plot(reward_history_iters,reward_history,label="train")
260         plt.savefig(filename + "train_reward")
261         ax.plot(reward_history_iters_validation,reward_history_validation,label
                ="validation")
262         fig.legend()
263         plt.savefig(filename + "valid_reward")
264         np.save(filename+"reward_hisotry_iters",np.array(reward_history_iters))
265         np.save(filename+"reward_hisotry",np.array(reward_history))
266         np.save(filename+"reward_hisotry_iters_valid",np.array(
                reward_history_iters_validation))
267         np.save(filename+"reward_hisotry_iters_valid",np.array(
                reward_history_validation))
268         if recordwandb:
269             wandb.log({"plot":plt})
270
271
272
273 bar.finish()
```

References

Christoph Adami. The Elements of Intelligence. Artificial Life, 29(3):293–307, 08 2023. ISSN 1064-5462. doi: 10.1162/artl_a_00410. URL https://doi.org/10.1162/artl_a_00410.

Christos Alexopoulos. A note on state-space decomposition methods for analyzing stochastic flow networks. IEEE Transactions on Reliability, 44(2):354–357, 1995.

Dzmitry Bahdanau, Philemon Brakel, Kelvin Xu, Anirudh Goyal, Ryan Lowe, Joelle Pineau, Aaron C. Courville, and Yoshua Bengio. An actor-critic algorithm for sequence prediction. CoRR, abs/1607.07086, 2016. URL http://arxiv.org/abs/1607.07086.

Etienne Barnard. Temporal-difference methods and markov models. IEEE Transactions on Systems, Man, and Cybernetics, 23(2):357–365, 1993.

Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. Deepmind lab. CoRR, abs/1612.03801, 2016. URL http://arxiv.org/abs/1612.03801.

Randall D Beer. The dynamics of active categorical perception in an evolved model agent. Adaptive behavior, 11(4):209–243, 2003.

Marcus K Benna and Stefano Fusi. Computational principles of synaptic memory consolidation. Nature neuroscience, 19(12):1697–1706, 2016.

Michael Bowling and Manuela Veloso. Multiagent learning using a variable learning rate. Artificial Intelligence, 136(2):215–250, 2002.

Valentino Braitenberg. Vehicles: Experiments in Synthetic Psychology, volume 95. Duke University Press, 1986. doi: 10.2307/2185146.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. CoRR, abs/1606.01540, 2016. URL http://arxiv.org/abs/1606.01540.

Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 38(2):156–172, 2008.

Bruce Cam, Christopher L. Dembia, and Johnny Israeli. Reinforcement learning for bicycle control. In Report, pages 1–5, 2013. URL https://api.semanticscholar.org/CorpusID:352181.

Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In Proceedings of the 23rd International Conference on Machine Learning, ICML '06, page 161–168, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933832. doi: 10.1145/1143844.1143865. URL https://doi.org/10.1145/1143844.1143865.

Jim Martin Catacora Ocana, Francesco Riccio, Roberto Capobianco, and Daniele Nardi. Cooperative multi-agent deep reinforcement learning in soccer domains. In Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, page 1865–1867, Richland, SC, 2019. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 9781450363099.

Lawrence Cayton. Fast nearest neighbor retrieval for bregman divergences. In Proceedings of the 25th international conference on Machine learning, pages 112–119, 2008.

Ka-Hou Chan, Wei Ke, and Sio-Kei Im. Caru: A content-adaptive recurrent unit for the transition of hidden state in nlp. In Haiqin Yang, Kitsuchart Pasupa, Andrew Chi-Sing Leung, James T. Kwok, Jonathan H. Chan, and Irwin King, editors, Neural Information Processing, pages 693–703, Cham, 2020a. Springer International Publishing. ISBN 978-3-030-63830-6.

Ka-Hou Chan, Wei Ke, and Sio-Kei Im. Caru: A content-adaptive recurrent unit for the transition of hidden state in nlp. In International Conference on Neural Information Processing, pages 693–703. Springer, 2020b.

Melissa Chapman, Lily Xu, Marcus Lapeyrolerie, and Carl Boettiger. Bridging adaptive management and reinforcement learning for more robust decisions. Philosophical Transactions of the Royal Society B, 378(1881):20220195, 2023.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation, 2014. URL http://arxiv.org/abs/1406.1078.

JCH Christopher. Watkins and peter dayan. Q-Learning. Machine Learning, 8(3):279–292, 1992.

Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. CoRR, abs/1412.3555, 2014. URL http://arxiv.org/abs/1412.3555.

Caroline Claus and Craig Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. AAAI/IAAI, 1998(746-752):2, 1998.

John Conway et al. The game of life. Scientific American, 223(4):4, 1970.

Neil E Cotter and Peter R Conwell. Fixed-weight networks can learn. In 1990 IJCNN International Joint Conference on Neural Networks, pages 553–559. IEEE, 1990.

George E Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. IEEE Transactions on audio, speech, and language processing, 20(1):30–42, 2011.

Jay Robert B Del Rosario, Jefferson G Sanidad, Allimzon M Lim, Pierre Stanley L Uy, Allan Jeffrey C Bacar, Mark Anthony D Cai, and Alec Zandrae A Dubouzet. Modelling and characterization of a maze-solving mobile robot using wall follower algorithm. In Applied Mechanics and Materials, volume 446, pages 1245–1249. Trans Tech Publ, 2014.

Sien Dieltiens, Frederik Debrouwere, Marc Juwet, and Eric Demeester. Practical application of the whipple and carvallo stability model on modern bicycles with pedal assistance. Applied Sciences, 10(16):5672, Aug 2020. ISSN 2076-3417. doi: 10.3390/app10165672. URL http://dx.doi.org/10.3390/app10165672.

Michael Fairbank and Eduardo Alonso. A comparison of learning speed and ability to cope without exploration between dhp and td (0). In The 2012 International Joint Conference on Neural Networks (IJCNN), pages 1–8. IEEE, 2012.

Michael Fairbank, Eduardo Alonso, and Danil Prokhorov. An equivalence between adaptive dynamic programming with a critic and backpropagation through time. IEEE Transactions on Neural Networks and Learning Systems, 24(12):2088–2100, 2013.

Michael Fairbank, Shuhui Li, Xingang Fu, Eduardo Alonso, and Donald Wunsch. An adaptive recurrent neural-network controller using a stabilization matrix and predictive inputs to solve a tracking problem under disturbances. Neural Networks, 49:74–86, 2014a.

Michael Fairbank, Danil Prokhorov, and Eduardo Alonso. Clipping in neurocontrol by adaptive dynamic programming. IEEE transactions on neural networks and learning systems, 25(10): 1909–1920, 2014b.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In International conference on machine learning, pages 1126–1135. PMLR, 2017.

Sarah Finney, Natalia Gardiol, Leslie Pack Kaelbling, and Tim Oates. The thing that we tried didn't work very well : Deictic representation in reinforcement learning. CoRR, abs/1301.0567, 2013. URL http://arxiv.org/abs/1301.0567.

Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. CoRR, abs/1605.06676, 2016. URL http://arxiv.org/abs/1605.06676.

C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - A differentiable physics engine for large scale rigid body simulation, 2021. URL https://arxiv.org/abs/2106.13281.

Xingang Fu, Shuhui Li, Michael Fairbank, Donald C Wunsch, and Eduardo Alonso. Training recurrent neural networks with the levenberg–marquardt algorithm for optimal control of a grid-connected converter. IEEE transactions on neural networks and learning systems, 26(9): 1900–1912, 2014.

Yuchuan Fu, Changle Li, F. Richard Yu, Tom H. Luan, and Yao Zhang. A selective federated reinforcement learning strategy for autonomous driving. IEEE Transactions on Intelligent Transportation Systems, 24(2):1655–1668, 2023. doi: 10.1109/TITS.2022.3219644.

Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In International Conference on Machine Learning, pages 1587–1596. PMLR, 2018.

Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In 2013 IEEE international conference on acoustics, speech and signal processing, pages 6645–6649. Ieee, 2013.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. CoRR, abs/1410.5401, 2014. URL http://arxiv.org/abs/1410.5401.

Amy Greenwald, Keith Hall, and Roberto Serrano. Correlated q-learning. In ICML, volume 3, pages 242–249, 2003.

Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. Reinforcement learning with deep energy-based policies. In International Conference on Machine Learning, pages 1352–1361. PMLR, 2017.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In International Conference on Machine Learning, pages 1861–1870. PMLR, 2018.

Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. CoRR, abs/1507.06527, 2015. URL http://arxiv.org/abs/1507.06527.

Conor F. Hayes, Roxana Radulescu, Eugenio Bargiacchi, Johan Källström, Matthew Macfarlane, Mathieu Reymond, Timothy Verstraeten, Luisa M. Zintgraf, Richard Dazeley, Fredrik Heintz, Enda Howley, Athirai A. Irissappane, Patrick Mannion, Ann Nowé, Gabriel de Oliveira Ramos, Marcello Restelli, Peter Vamplew, and Diederik M. Roijers. A practical guide to multi-objective reinforcement learning and planning. CoRR, abs/2103.09568, 2021. URL https://arxiv.org/abs/2103.09568.

Johannes Heinrich, Marc Lanctot, and David Silver. Fictitious self-play in extensive-form games. In International conference on machine learning, pages 805–813. PMLR, 2015.

Matteo Hessel, Hubert Soyer, Lasse Espeholt, Wojciech Czarnecki, Simon Schmitt, and Hado van Hasselt. Multi-task deep reinforcement learning with popart. CoRR, abs/1809.04474, 2018. URL http://arxiv.org/abs/1809.04474.

Geoffrey E Hinton. Boltzmann machine. Scholarpedia, 2(5):1668, 2007.

Sepp Hochreiter and Jürgen Schmidhuber. Lstm can solve hard long time lag problems. In M.C. Mozer, M. Jordan, and T. Petsche, editors, Advances in Neural Information Processing Systems, volume 9. MIT Press, 1996. URL https://proceedings.neurips.cc/paper_files/paper/1996/file/a4d2f0d23dcc84ce983ff9157f8b7f88-Paper.pdf.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. Neural computation, 9(8): 1735–1780, 1997.

Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In International Conference on Artificial Neural Networks, pages 87–94. Springer, 2001.

John H Holland, John Henry Holland, et al. Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. University of Michigan press, 1975.

Bo Hu and Yuhai Tu. Behaviors and strategies of bacterial navigation in chemical and nonchemical gradients. PLoS computational biology, 10(6):e1003672, 2014.

Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z. Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. CoRR, abs/1611.05397, 2016a. URL http://arxiv.org/abs/1611.05397.

Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z. Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. CoRR, abs/1611.05397, 2016b. URL http://arxiv.org/abs/1611.05397.

Michael I. Jordan. Chapter 25 - serial order: A parallel distributed processing approach. In John W. Donahoe and Vivian Packard Dorsel, editors, Neural-Network Models of Cognition, volume 121 of Advances in Psychology, pages 471–495. North-Holland, 1997. doi: https://

doi.org/10.1016/S0166-4115(97)80111-2. URL https://www.sciencedirect.com/science/article/pii/S0166411597801112.

Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. CoRR, cs.AI/9605103, 1996. URL https://arxiv.org/abs/cs/9605103.

Christos Kaplanis, Murray Shanahan, and Claudia Clopath. Continual reinforcement learning with complex synapses. CoRR, abs/1802.07239, 2018. URL http://arxiv.org/abs/1802.07239.

Max Kleiman-Weiner, Mark K. Ho, Joseph L. Austerweil, Michael L. Littman, and Joshua B. Tenenbaum. Coordinate to cooperate or compete: Abstract goals and joint intentions in social interaction. Cognitive Science, 2016. URL https://api.semanticscholar.org/CorpusID:7561270.

Vijay Konda and John Tsitsiklis. Actor-critic algorithms. Advances in neural information processing systems, 12, 1999.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems, 25:1097–1105, 2012.

Harold William Kuhn and Albert William Tucker. Contributions to the Theory of Games, volume 2. Princeton University Press, 1953.

Marc Lanctot, Vinícius Flores Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Pérolat, David Silver, and Thore Graepel. A unified game-theoretic approach to multiagent reinforcement learning. CoRR, abs/1711.00832, 2017. URL http://arxiv.org/abs/1711.00832.

Terran Lane, Martin Ridens, and Scott Stevens. Reinforcement learning in nonstationary environment navigation tasks. In Conference of the Canadian Society for Computational Studies of Intelligence, pages 429–440. Springer, 2007.

Wei Li, Fan Zhou, Kaushik Roy Chowdhury, and Waleed Meleis. QTCP: Adaptive congestion control with reinforcement learning. IEEE Transactions on Network Science and Engineering, 6(3):445–458, 2018.

Timothy P Lillicrap and Adam Santoro. Backpropagation through time and the brain. Current Opinion in Neurobiology, 55:82–89, 2019. ISSN 0959-4388. doi: https://doi.org/10.1016/j.conb.2019.01.011. URL https://www.sciencedirect.com/science/article/pii/S0959438818302009. Machine Learning, Big Data, and Neuroscience.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971, 2015.

Long-Ji Lin. Reinforcement learning for robots using neural networks. Carnegie Mellon University, USA, 1992. UMI Order No. GAX93-22750.

Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In Machine learning proceedings 1994, pages 157–163. Elsevier, 1994.

Michael L Littman. Friend-or-foe q-learning in general-sum games. In ICML, volume 1, pages 322–328, 2001.

Michael L Littman, Anthony R Cassandra, and Leslie Pack Kaelbling. Learning policies for partially observable environments: Scaling up. In Machine Learning Proceedings 1995, pages 362–370. Elsevier, 1995.

James T Lo and Devasis Bassu. Adaptive vs. accommodative neural networks for adaptive system identification. In IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No. 01CH37222), volume 2, pages 1279–1284. IEEE, 2001.

Hamid R Maei, Csaba Szepesvári, Shalabh Bhatnagar, and Richard S Sutton. Toward off-policy learning control with function approximation. In Proceedings of the 27th International Conference on Machine Learning (ICML-10), pages 719–726, 2010.

Hamid Reza Maei, Csaba Szepesvari, Shalabh Bhatnagar, Doina Precup, David Silver, and Richard S Sutton. Convergent temporal-difference learning with arbitrary smooth function approximation. In NIPS, pages 1204–1212, 2009.

Lingheng Meng, Rob Gorbet, and Dana Kulić. Partial Observability during DRL for Robot Control. arXiv e-prints, art. arXiv:2209.04999, September 2022. doi: 10.48550/arXiv.2209.04999.

Hyman P. Minsky, Arthur M. Okun, and Clark Warburton. Comments on friedman's and schwartz' money and the business cycles. The Review of Economics and Statistics, 45(1):64–78, 1963. ISSN 00346535, 15309142. URL http://www.jstor.org/stable/1927149.

Volodymyr Mnih. Machine learning for aerial image labeling. University of Toronto (Canada), 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. CoRR, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. nature, 518(7540):529–533, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In International conference on machine learning, pages 1928–1937. PMLR, 2016.

Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisỳ, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. Science, 356(6337):508–513, 2017.

John J Murray, Chadwick J Cox, George G Lendaris, and Richard Saeks. Adaptive dynamic programming. IEEE transactions on systems, man, and cybernetics, Part C (Applications and Reviews), 32(2):140–153, 2002.

Thanh Thi Nguyen. A multi-objective deep reinforcement learning framework. CoRR, abs/1803.02965, 2018. URL http://arxiv.org/abs/1803.02965.

Alex Nichol and John Schulman. Reptile: a scalable metalearning algorithm. arXiv preprint arXiv:1803.02999, 2(3):4, 2018.

Frank Nielsen. Hierarchical clustering. In Introduction to HPC with MPI for Data Science, pages 195–211. Springer, 2016.

Markus Olsson, Simon Malm, and Kasper Witt. Evaluating the effects of hyperparameter optimization in vizdoom, 2022. URL https://urn.kb.se/resolve?urn=urn:nbn:se:his:diva-21533.

Pedro A. Ortega, Jane X. Wang, Mark Rowland, Tim Genewein, Zeb Kurth-Nelson, Razvan Pascanu, Nicolas Heess, Joel Veness, Alexander Pritzel, Pablo Sprechmann, Siddhant M. Jayakumar, Tom McGrath, Kevin J. Miller, Mohammad Gheshlaghi Azar, Ian Osband, Neil C. Rabinowitz, András György, Silvia Chiappa, Simon Osindero, Yee Whye Teh, Hado van Hasselt, Nando de Freitas, Matthew M. Botvinick, and Shane Legg. Meta-learning of sequential strategies. CoRR, abs/1905.03030, 2019. URL http://arxiv.org/abs/1905.03030.

Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. Actor-Mimic: Deep Multitask and Transfer Reinforcement Learning. arXiv e-prints, art. arXiv:1511.06342, November 2015. doi: 10.48550/arXiv.1511.06342.

Patrick M Pilarski, Michael R Dawson, Thomas Degris, Farbod Fahimi, Jason P Carey, and Richard S Sutton. Online human training of a myoelectric prosthesis controller via actor-critic reinforcement learning. In 2011 IEEE international conference on rehabilitation robotics, pages 1–7. IEEE, 2011.

Jordan B Pollack and Alan D Blair. Why did td-gammon work? Advances in Neural Information Processing Systems, 9(9):10–16, 1997.

Vitchyr Pong, Shixiang Gu, Murtaza Dalal, and Sergey Levine. Temporal difference models: Model-free deep RL for model-based control. CoRR, abs/1802.09081, 2018. URL http://arxiv.org/abs/1802.09081.

Warren B Powell. Approximate Dynamic Programming: Solving the curses of dimensionality, volume 703. John Wiley & Sons, 2007.

Danil V Prokhorov and Donald C Wunsch. Adaptive critic designs. IEEE transactions on Neural Networks, 8(5):997–1007, 1997a.

Danil V Prokhorov, LA Feldkarnp, and I Yu Tyukin. Adaptive behavior with fixed weights in rnn: an overview. In Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN'02 (Cat. No. 02CH37290), volume 3, pages 2018–2022. IEEE, 2002.

D.V. Prokhorov and D.C. Wunsch. Adaptive critic designs. IEEE Transactions on Neural Networks, 8(5):997–1007, 1997b. doi: 10.1109/72.623201.

Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable baselines3. https://github.com/DLR-RM/stable-baselines3, 2019.

Jette Randløv and Preben Alstrøm. Learning to drive a bicycle using reinforcement learning and shaping. In ICML, volume 98, pages 463–471, 1998.

James B Rawlings. Tutorial overview of model predictive control. IEEE control systems magazine, 20(3):38–52, 2000.

Stuart J Russell and Peter Norvig. Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited,, 2016.

Brian Sallans and Geoffrey E Hinton. Reinforcement learning with factored states and actions. The Journal of Machine Learning Research, 5:1063–1088, 2004.

Arthur L Samuel. Machine learning. The Technology Review, 62(1):42–45, 1959.

Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In International conference on machine learning, pages 1842–1850. PMLR, 2016.

Nikolay Savinov, Alexey Dosovitskiy, and Vladlen Koltun. Semi-parametric topological memory for navigation. CoRR, abs/1803.00653, 2018. URL http://arxiv.org/abs/1803.00653.

Tom Schaul and Jürgen Schmidhuber. Metalearning. Scholarpedia, 5(6):4650, 2010.

Jürgen Schmidhuber. An on-line algorithm for dynamic reinforcement learning and planning in reactive environments. In 1990 IJCNN international joint conference on neural networks, pages 253–258. IEEE, 1990.

John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. CoRR, abs/1502.05477, 2015. URL http://arxiv.org/abs/1502.05477.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. CoRR, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

Pierre Sermanet, Koray Kavukcuoglu, Soumith Chintala, and Yann LeCun. Pedestrian detection with unsupervised multi-stage feature learning. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 3626–3633, 2013.

Varun Shah. Reinforcement learning for autonomous software agents: Recent advances and applications. Revista Espanola de Documentacion Cientifica, 14(1):56–71, 2020.

Sahil Sharma and Balaraman Ravindran. Online multi-task learning using active sampling. CoRR, abs/1702.06053, 2017. URL http://arxiv.org/abs/1702.06053.

Yoav Shoham, Rob Powers, and Trond Grenager. If multi-agent learning is the answer, what is the question? Artificial intelligence, 171(7):365–377, 2007.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. nature, 529(7587): 484–489, 2016.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. CoRR, abs/1712.01815, 2017. URL http://arxiv.org/abs/1712.01815.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science, 362(6419):1140–1144, 2018.

Bradly C. Stadie, Ge Yang, Rein Houthooft, Xi Chen, Yan Duan, Yuhuai Wu, Pieter Abbeel, and Ilya Sutskever. Some considerations on learning to explore via meta-reinforcement learning. CoRR, abs/1803.01118, 2018. URL http://arxiv.org/abs/1803.01118.

Malcolm Strens. A bayesian framework for reinforcement learning. In ICML, volume 2000, pages 943–950, 2000.

Wen Sun, Nan Jiang, Akshay Krishnamurthy, Alekh Agarwal, and John Langford. Model-based rl in contextual decision processes: Pac bounds and exponential improvements over model-free approaches. In Conference on learning theory, pages 2898–2933. PMLR, 2019.

Richard S Sutton. Learning to predict by the methods of temporal differences. Machine learning, 3(1):9–44, 1988.

Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. MIT press, 2018.

Richard S Sutton, Andrew G Barto, et al. Introduction to reinforcement learning, volume 135. MIT press Cambridge, 1998.

Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In Proceedings of the tenth international conference on machine learning, pages 330–337, 1993.

Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, Timothy P. Lillicrap, and Martin A. Riedmiller. Deepmind control suite. CoRR, abs/1801.00690, 2018. URL http://arxiv.org/abs/1801.00690.

Hui Yie Teh, Andreas W Kempa-Liehr, and Kevin I-Kai Wang. Sensor data quality: A systematic review. Journal of Big Data, 7(1):1–49, 2020.

Gerald Tesauro. Temporal difference learning and td-gammon. Communications of the ACM, 38(3):58–68, 1995.

Sebastian Thrun and Lorien Pratt. Learning to learn: Introduction and overview. In Learning to learn, pages 3–17. Springer, 1998.

John N Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. IEEE transactions on automatic control, 42(5):674–690, 1997.

Ryan Urbanowicz and Jason Moore. Learning classifier systems: A complete introduction, review, and roadmap. Journal of Artificial Evolution and Applications, 2009, 01 2009. doi: 10.1155/2009/736398.

Ragav Venkatesan and Baoxin Li. Convolutional neural networks in visual computing: a concise guide. CRC Press, 2017.

Fei-Yue Wang, Huaguang Zhang, and Derong Liu. Adaptive dynamic programming: An introduction. IEEE computational intelligence magazine, 4(2):39–47, 2009.

Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Rémi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. CoRR, abs/1611.01224, 2016. URL http://arxiv.org/abs/1611.01224.

Christopher JCH Watkins and Peter Dayan. Q-learning. Machine learning, 8(3-4):279–292, 1992.

Paul J Werbos. Backpropagation through time: what it does and how to do it. Proceedings of the IEEE, 78(10):1550–1560, 1990.

Paul J Werbos. Brain-like intelligent control: from neural nets to larger-scale systems. In Proceedings of the 36th IEEE Conference on Decision and Control, volume 4, pages 3902–3904. IEEE, 1997.

Paul J. Werbos. Approximate Dynamic Programming (ADP), pages 76–82. Springer International Publishing, Cham, 2021. ISBN 978-3-030-44184-5. doi: 10.1007/978-3-030-44184-5_100096. URL https://doi.org/10.1007/978-3-030-44184-5_100096.

Nina Wiedemann, Valentin Wüest, Antonio Loquercio, Matthias Müller, Dario Floreano, and Davide Scaramuzza. Training Efficient Controllers via Analytic Policy Gradient. arXiv e-prints, art. arXiv:2209.13052, September 2022. doi: 10.48550/arXiv.2209.13052.

Daan Wierstra, Alexander Foerster, Jan Peters, and Juergen Schmidhuber. Solving deep memory pomdps with recurrent policy gradients. In International conference on artificial neural networks, pages 697–706. Springer, 2007.

Daan Wierstra, Alexander Förster, Jan Peters, and Jürgen Schmidhuber. Recurrent policy gradients. Logic Journal of the IGPL, 18(5):620–634, 2010.

Wikipedia. Euler tour technique — Wikipedia, the free encyclopedia, 2023. URL https://en.wikipedia.org/wiki/Euler_tour_technique#/media/File:Stirling_permutation_Euler_tour.svg. [Online; accessed 18-July-2023].

R.J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning, 8:229–256, 05 1992. doi: 10.1007/BF00992696.

Stewart W. Wilson. Classifier Fitness Based on Accuracy. Evolutionary Computation, 3(2): 149–175, 06 1995. ISSN 1063-6560. doi: 10.1162/evco.1995.3.2.149. URL https://doi.org/10.1162/evco.1995.3.2.149.

Stewart W Wilson and David E Goldberg. A critical review of classifier systems. In Proceedings of the third international conference on Genetic algorithms, pages 244–255, 1989.

Cathy Wu, Aravind Rajeswaran, Yan Duan, Vikash Kumar, Alexandre M. Bayen, Sham M. Kakade, Igor Mordatch, and Pieter Abbeel. Variance reduction for policy gradient with action-dependent factorized baselines. CoRR, abs/1803.07246, 2018. URL http://arxiv.org/abs/1803.07246.

Nikolai Yakovenko, Liangliang Cao, Colin Raffel, and James Fan. Poker-cnn: A pattern learning strategy for making draws and bets in poker games using convolutional networks. Proceedings of the AAAI Conference on Artificial Intelligence, 30, 02 2016. doi: 10.1609/aaai.v30i1.10013.

Zhaoyang Yang, Kathryn E Merrick, Hussein A Abbass, and Lianwen Jin. Multi-task deep reinforcement learning for continuous action control. In IJCAI, volume 17, pages 3301–3307, 2017.

A Steven Younger, Peter R Conwell, and Neil E Cotter. Fixed-weight on-line learning. IEEE Transactions on Neural Networks, 10(2):272–283, 1999.

A Steven Younger, Sepp Hochreiter, and Peter R Conwell. Meta-learning with backpropagation. In IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No. 01CH37222), volume 3, page 10. IEEE, 2001.

Zhaoxiang Zang, Dehua Li, and Junying Wang. Learning classifier systems with memory condition to solve non-markov problems. Soft Computing, 19(6):1679–1699, 2015.

Andreas Zell. Simulation neuronaler netze, volume 1. Addison-Wesley Bonn, 1994.

Jingwei Zhang, Jost Tobias Springenberg, Joschka Boedecker, and Wolfram Burgard. Deep reinforcement learning with successor features for navigation across similar environments. In

2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 2371–2378. IEEE, 2017.

Guo-Bing Zhou, Jianxin Wu, Chen-Lin Zhang, and Zhi-Hua Zhou. Minimal gated unit for recurrent neural networks. International Journal of Automation and Computing, 13(3):226–234, 2016.