Real Time Recurrent Learning with Complex-Valued Trace Units

by

Esraa Elelimy

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Recurrent Neural Networks (RNNs) are typically used to learn representations in partially observable environments. Unfortunately, training RNNs is known to be difficult, and the difficulty increases for agents who learn online and continually interact with the environment. Two common strategies to overcome this difficulty are to approximate gradient-based algorithms for learning the recurrent state or to find a recurrent architecture for which a computationally cheap gradient-based learning algorithm exists. Methods in the second category often limit representational capacity, just as using linear activations or diagonal weight matrices. In this work, we propose a novel recurrent architecture called Recurrent Trace Units (RTUs). RTUs expand representation capacity, but remain inexpensive to train. We derive RT2, a real-time recurrent learning algorithm for RTUs that is tractable, exact and has linear compute and memory complexities. We investigate performance on a diagnostic benchmark inspired by animal learning and across several partially observable control environments. We show the agents that use RT2 achieve overall better performance when faced with long-term prediction tasks, and reach their goals faster in control tasks.

Preface

No parts of this thesis have been published.

To my parents

Acknowledgements

First and foremost, I would like to thank my supervisor, Martha White, for her continuous help and feedback throughout the project. Over the past year, Martha's support has extended beyond my research, and I am forever grateful for the positive impact that working with Martha had on me. I couldn't have asked for a better supervisor. Second, I thank Adam White for all the insightful discussions on animal learning and for teaching us how to do good empirical experiments that answer clear scientific questions. Third, I thank Subhojeet Pramanik for the insightful discussions that shaped some of the ideas in this thesis and for the help in designing and running experiments. I am grateful to Martha Steenstrup for teaching me how to improve my scientific writing skills during the writing workshop; Martha's insights profoundly influenced how I communicate my ideas. I am also thankful to Michael Bowling for giving valuable feedback on this work and how it could be improved. Finally, I am grateful to all the wonderful friends and colleagues at RLAI and AMII for creating such an exciting research environment.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Animals perceive their surrounding environment through imperfect sensory observations. These observations reflect only partial information about the environment and determine the limits of the knowledge about the surrounding environment. For example, an animal can not perceive an object outside their field of view, which is a limitation of their vision.

Predicting and controlling the surrounding environment under partial observability is a salient feature of all natural intelligent agents. For example, imagine driving on a highway. You see a sign *Sharp Curve in* 10 *km*. You keep driving, and for the next 10 km, there is no information about that sharp curve. Still, your brain compensates for this partial observability by remembering important information about the world and signaling you to slow down in anticipation of the sharp curve. Moreover, empirical experiments on animal learning confirm that animals can make long-term predictions under partial observability. In classical conditioning, after a series of trials conditioning food to a cue signal, animals learn to predict food arrival based on the cue signal, usually a sound, which is observed many steps before food arrival (Pavlov, 1927).

Predicting and controlling the world is not innate to animals; they learn this ability by interacting with the world. The interaction with the world happens over the animal's lifetime, and through this continual interaction, animals learn some of the underlying regularities in the observations allowing them to accurately predict and control different aspects of the world around them.

Interacting with the world under partial observability is not limited to natural intelligent agents. A self-driving car, for example, also interacts with the world under partial observability; the sensors attached to the self-driving car limit the perceived information about its surrounding environment. Thus, to successfully deploy artificial agents in the real world, we need to equip them with the ability to predict and control aspects of the environment under partial observability.

In this work, we tackle the computational aspects of learning under partial observability. We study this problem within the Reinforcement Learning (RL) framework. RL agents learn from experience by interacting with their environment, similar to how animals learn. We also focus on the continual online learning setting where agents learn and are evaluated as they interact with the environment, which resembles the setting faced in the real world.

1.1 Problem Statement

When the state of the environment is partially observable, agents are responsible for constructing and maintaining their sense of state using the stream of observations. The constructed *agent state* summarizes all past environment-agent interactions that are useful to predict and control future interactions (Sutton, 2020b) and helps the agent mitigate partial observability.

Recurrent Neural Networks (RNNs) provide a possible solution for the agent state construction problem (Kapturowski et al., 2019; Li et al., 2015; Hausknecht and Stone, 2015; Espeholt et al., 2018; Gruslys et al., 2018). An RNN can learn to summarize and abstract a long trajectory of interactions in one vector, its recurrent state. This concise summary of all interactions helps the agent predict aspects of the environment and take better actions. Unfortunately, training RNNs is difficult (Pascanu, Mikolov, and Bengio, 2013), and the difficulty increases when we use RNNs in the online continual learning setting, which is the focus of this work.

Training RNNs typically uses Truncated-BackPropagation Through Time (T-BPTT), a gradient-based learning algorithm that unrolls the recurrent dynamics through time up to a specific time step, defined by the truncation length (Williams and Peng, 1990). T-BPTT requires saving the inputs to the RNN from all the previous time steps to perform one update to the RNN's learnable parameters. As a result, the computation and memory complexities of T-BPTT are functions of the truncation length. Learning with T-BPTT involves a trade-off between the network's ability to look further in time and its compute and memory requirements; to look further back in time, we need to increase the truncation length, increasing both the memory and the compute required per update.

An alternative to T-BPTT is Real-Time Recurrent Learning (RTRL), a gradient-based learning algorithm that uses the gradient's recurrent nature and carries forward the needed gradient information instead of unrolling the recurrent dynamics back in time (Williams and Zipser, 1989). In theory, the RTRL algorithm allows the network to look back arbitrarily many steps, and the computation and memory complexities per update are fixed, albeit expensive. While the motivation behind RTRL is appealing for online learning, its expensive computational and memory complexities render it intractable even for moderately sized networks.

We can divide the literature on overcoming the expensive training of RNNs into two categories. In the first category, we have methods for finding an approximate gradient-based algorithm for learning the recurrent dynamics. In the second category, we have attempts to find a recurrent architecture for which a computationally cheap gradient-based learning algorithm exists. Each category has advantages and flaws, which we touch upon next.

Methods in the first category typically start with a widely used RNN architecture such as Vanilla RNN, Gated Recurrent Units (GRUs) (Cho et al., 2014), and Long-Short Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) and attempt to approximate the gradient while maintaining a good performance. Examples of these methods include the NoBackTrack algorithm, which avoids propagating the gradient through time by maintaining a vector in the parameter space corresponding to a stochastic gradient estimation (Ollivier, Tallec, and Charpiat, 2015). The NoBackTrack algorithm requires that the RNN is sparse. Moreover, due to the randomness in estimating the gradient direction, it suffers from high variance. Improving on the NoBackTrack algorithm is the Unbiased Online Recurrent Optimization (UORO) algorithm (Tallec and Ollivier, 2017), which removed the need for sparsity. However, the high variance issue is still unresolved (Cooijmans and Martens, 2019). Another method in this category is Sparse N-Step Approximation (SnAp) (Menick et al., 2021), which uses a sparse approximation of the gradient to build a tractable RTRL algorithm. However, the tractability of the SnAp algorithm is tied to having a highly sparse gradient approximation, which leads to a biased gradient update.

To find an RNN for which exact gradient estimation is cheap, methods in the second category usually restrict the RNN architecture to reduce the number of learnable parameters and, as a result, the complexity of calculating the gradient. Restricting the RNN architecture leads to losing its full representational powers, which can result in poor performance. For example, Javed et al. (2023) showed that Columnar Networks, a diagonal RNN architecture, often performed poorly. To overcome the loss of representational power in Columnar Networks, they proposed combining them with a constructive approach that iteratively learns new columnar features. Recent work suggests overcoming the poor performance of diagonal RNNs by having a complex-valued recurrent state instead of restricting it to real values (Orvieto et al., 2023). However, this new recurrent architecture, namely Linear Recurrent Units (LRUs), was only applied in the offline learning setting with T-BPTT as the learning algorithm. Hence, it has the same tradeoff between looking further back in time and the computational complexity.

The self-attention mechanism (Vaswani et al., 2017) has been widely used for modeling long sequences, making it another potential solution for the agent state construction problem (Parisotto et al.). However, recent work suggests that the self-attention mechanism is unsuitable for modeling sequences with temporal correlation(Zeng et al., 2022), which is the case in RL. Additionally, we breifly explain in 2.3 why the sequence processing, as done in the self-attention mechanism, is unsuitable for our problem setting.

We can think of an online learner as mostly doing three things:

1. Defining a parametrized function that map the observations to an internal state. A naive

algorithm could store all observations and define an internal state as the history of all observations. On the other hand, a plausible algorithm learns to construct a concise representation of that history.

2. Making predictions (or taking actions) with the help of the function defined in the previous step.

3. Updating the functions' learnable parameters when given an error signal to minimize the overall error signals.

Algorithm 1 shows an example of an online learner for a prediction task. In this algorithm, we assumed that step (1) happens before learning starts since this step specifies how many layers and parameters we want in the neural network and the connections between the layers. The learner continually repeat step (2), making predictions, and step (3), updating the learnable parameters.[1]

Algorithm 1 Online perdiction

```
Input: a differentiable parametrization of the recurrent function $\mathbf{f} : \mathbb{R}^n \times \mathbb{R}^d \to \mathbb{R}^n$.
▷ $\mathbf{f}$ is a vector-valued function that maps the previous recurrent state and the current input to a new recurrent state.
Input: a differentiable parametrization mapping the recurrent state to a prediction $g : \mathbb{R}^n \to \mathbb{R}$.
                ▷ $g$ is a scalar-valued function that maps the recurrent state to a prediction.
Initialize: the set of parameters defining $\mathbf{f}(\cdot)$ and $g(\cdot)$.
Initialize: the initial recurrent state $\mathbf{h}_0 \doteq \mathbf{0}$.

for each input $\mathbf{x_t}$ do
    Forward Pass: update the recurrent state: $\mathbf{h}_t = \mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x_t})$
    and make a prediction: $\hat{y}_t = g(\mathbf{h}_t)$.
    Evaluate: recieve a loss: $\mathcal{L}_t(\hat{y}_t, y_t)$.        ▷ $y_t$ is the ground truth at time $t$.
    Backward Pass: update $\mathbf{f}(\cdot)$ and $g(\cdot)$ parameters to minimze the loss.
                ▷ The Backward Pass may not happen every time step.
```

The setting of continual online learning imposes a constraint on the learning algorithm that arguably does not exist when learning offline. The constraint is that the algorithm's computation and memory complexities per time step should be constant, i.e., independent of the current time step. An agent continually interacting with the environment makes a prediction or takes an action at each time step. Hence, if the learning algorithm's complexity depends on time, the memory and the computation resources required by the algorithm will keep growing indefinitely, which is impractical. This constraint restricts both the class of parametrized functions and the learning algorithm; a function that takes as input the whole history of interactions, for example, violates this

[1]To my knowledge no algorithm is yet able to construct the function on the fly so we assume that this phase is done prior to interacting with the environment.

constraint, and a learning algorithm that requires saving all previous inputs to perform gradient updates also violates this constraint.

1.2 Contributions

This work introduces a novel recurrent architecture, Recurrent Trace Units (RTUs). RTUs belong to the second category: it is a recurrent architecture for which calculating the exact gradient is cheap. We derive an RTRL learning algorithm for RTUs with linear computation and memory complexities while still calculating the exact gradient. We call the combination of RTUs and RTRL the RT2 algorithm: **R**eal-**T**ime **RT**Us. Unlike T-BPTT, RT2 calculates the exact gradient and can retain information from arbitrary long sequences without truncation. Additionally, RT2 is suitable for online learning even under constrained computational resources as it has linear complexity. Similar to LRUs, RTUs use a complex-valued diagonal recurrence. However, RTUs use different representations of complex numbers that facilitate RTRL updates. We show that RT2 outperforms other recurrent-based approaches in the online learning setting, first on a prediction benchmark inspired by animal learning and then on several partially observable classical control tasks.

Chapter 2

Background

In this chapter, we briefly describe Markov Decision Processes (MDPs), a formalization of the sequential decision-making problem with an underlying assumption of Markovian states. We then introduce the partially observable setting where the Markovian assumption is invalid and describe the agent state construction problem that arises in this setting. We then briefly touch on the self-attention mechanism and why it is unsuitable for our problem setting. Finally, we describe using Recurrent Neural Networks (RNNs) as a solution method for the agent state construction problem and discuss their limitations when the agent is learning online.

2.1 Markov Decision Processes

Markov Decision Processes (MDPs) are a classical framework for modeling the sequential decision-making problem. In the MDP formalization, an agent and an environment interact over discrete time steps $t = 0, 1, 2, \cdots$. At each time step t, the agent perceives a state $\mathbf{s}_t \in \mathcal{S}$ and takes an action $A_t \in \mathcal{A}(\mathbf{s}_t)$. Depending on the action taken, the agent finds itself in a new state $\mathbf{s}_{t+1} \in \mathcal{S}$ and it gets a reward signal from the environment $R_{t+1} \in \mathbb{R}$. The agent's goal is to maximize the received rewards. Towards this goal, the agent learns a policy $\pi : \mathcal{S} \to \mathcal{A}$ specifying the probability of selecting each action in the current state. The better the learned policy, the more rewards the agent collects. Formally, the agent aims to maximize the expected cumulative sum of the rewards, the return: $G_t \doteq \mathbb{E}_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots]$, where $\gamma \in [0, 1]$ is a discount factor determining how much the agent cares about immediate and future rewards (Sutton and Barto, 2018).

The MDP formalization assumes that the states have a Markov property. The Markovian assumption means that the perceived state at time t contains all the information about the environment and past agent-environment interactions. However, in many applications, the agent only perceives its sensory observations, which reflect only partial information about the environment.

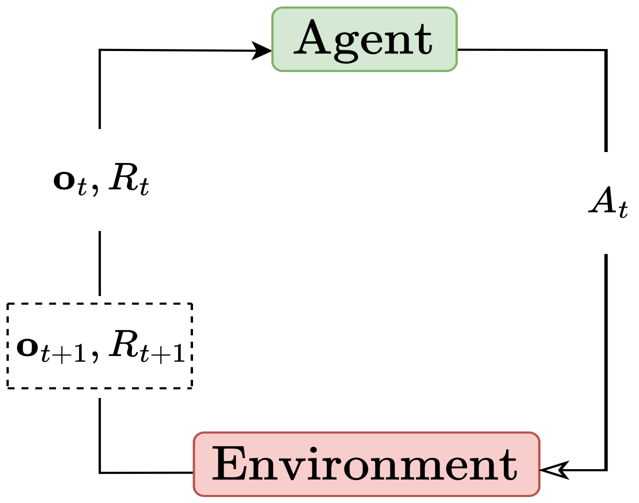


Figure 2.1: Agent-environment interaction step at time t. $\mathbf{o}_t$ is a sensory observation reflecting incomplete information about the environment.

Figure 2.1 shows one step of the agent-environment interactions at time t in such cases. An observation at time t, $\mathbf{o}_t$, does not capture all the aspects of the environment and past agent-environment interactions. Hence, the observations lack the Markov property. We call the setting where the agent perceives $\mathbf{o}_t$ rather than $\mathbf{s}_t$ as the *partially observable setting.*

2.2 Agent State Construction

We can view the Markovian assumption as a limitation on the agent rather than the MDP framework (Sutton and Barto, 2018). The agent is responsible for maintaining its sense of state. Hence, when interacting with a partially observable environment, the agent should learn to use the stream of observations to construct its state, the *agent state*. The agent state summarizes information from the history of the agent-environment interactions that are useful for prediction and control (Sutton, 2020b).

We can think of the state at time t as a concatenation of the whole interaction history up to t:

$$\mathbf{s}_t \doteq \mathbf{o}_0, A_1, R_1, \mathbf{o}_1, A_2, R_2, \dots \mathbf{o}_t.$$

However, as we saw in section 1.1, storing the whole history is not plausible; we want the agent to have constant memory and computational requirements per time step and storing the whole history causes the memory and the computation to grow with time. Alternatively, the agent should learn a concise representation of the history and update that representation at every time step using the new observation. We refer to the agent's internal representation of the history at time t as its hidden state $\mathbf{h}_t$, $\mathbf{h}_t$ approximates $\mathbf{s}_t$. The agent constructs its current hidden state $\mathbf{h_t} \in \mathbb{R}^n$ from its previous hidden state $\mathbf{h}_{t-1} \in \mathbb{R}^n$ and the recent observation $\mathbf{o}_t \in \mathbb{R}^d$ using a state-update function $\mathbf{f} : \mathbb{R}^n \times \mathbb{R}^d \to \mathbb{R}^n$:

$$\mathbf{h}_t = \mathbf{f}(\mathbf{h}_{t-1}, \mathbf{o}_t).$$

Agent state construction is essential to any RL agent under partial observability; the state is the input to the value function, the policy, and both an input and an output of the environment model for model-based RL agents. Thus, the quality of the constructed states heavily affects the agent's performance.

2.3 The Self-Attention Mechanism

This section briefly describes the self-attention mechanism and why it is unsuitable for the continual online learning setting.

Vaswani et al.(2017) introduced the self-attention mechanism for machine translation, and since then, this mechanism resulted in many advances in the field of language modelling (Dai et al., 2019; Radford et al., 2019; Yang et al., 2019). Roughly speaking, the core idea behind self-attention is that for a sequence of data points, we can find the similarity between each pair in this sequence, and based on this similarity measure, we generate a suitable output. Figure 2.2 illustrates this idea. The self-attention mechanism has been the key to the recent success of transformer-based architecture (Vaswani et al., 2017).

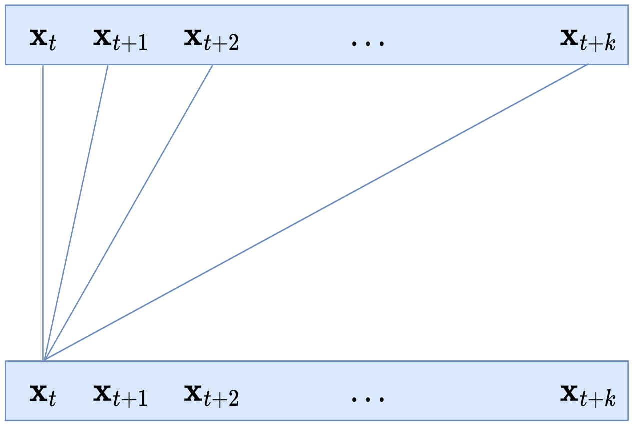


Figure 2.2: A rough schematic of the self-attention mechanism. The self-attention mechanism calculates the similarity between each input, such as $\mathbf{x}_t$, and all the other input elements in the sequence.

Two main issues limit the use of self-attention mechanisms in online learning. First, we need to have the whole sequence of observations before taking an action or updating the learnable parameters, which is impractical in continual learning. Second, calculating the similarity between each pair of points results in a computational complexity that is a function of k^2, where k is the sequence length. Moreover, calculating the similarity between all pairs ignores the temporal order of the data points, which limits the usefulness of self-attention when the data is temporally correlated (Zeng et al., 2022).

Recent work, such as the Gated Transformer-XL (GTrXL), tried to overcome the first issue by keeping a moving window of previous observations. As Figure 2.3 shows, keeping a moving window of past observations allows taking actions at each time step. However, we still have the computational complexity of operating on the saved history. We also lose all information from the history before the truncation length.

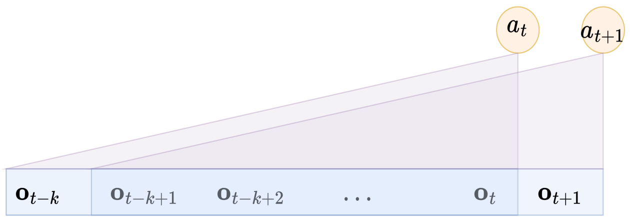


Figure 2.3: Using a moving window of past observations in GTrXL.

We end this section with a final remark on recent advances in language modeling. Recent advances in Large Language Models (LLMs) could give the impression that the problem of modeling long sequences is solved. However, all these advances were only limited to the offline learning setting, where training happens once using a large amount of offline data. Then, the model is frozen and deployed. This setting differs from how animals continually learn; success in this setting is rarely extendable to the continual online learning setting where observation occurs one at a time. Additionally, only a handful of institutions can benefit from and contribute to the recent advances in LLMs, as these models are expensive to train and use. In contrast, this thesis focuses on scalable, cheap algorithms for learning long temporal relations.

2.4 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) provide a solution method to learn the state-update function and the agent state.[1] This section describes two common ways to learn RNN's parameters online. We start with an overview of some mathematical notations and then describe online learning with RNNs.

[1]In the rest of the thesis, we use state, agent state, recurrent state, and hidden state interchangeably depending on the context. But in all cases, we mean the agent state. In the context of RNNs, we usually use recurrent state or hidden state to align with RNNs literature. While in the context of RL, we use agent state or simply state.

2.4.1 Mathematical Notation

Before we dive into learning with RNNs, we first describe the mathematical notations we use in the analysis and in the following chapters.[2]

Scalar-Vector Derivatives

Let $\mathbf{h} \in \mathbb{R}^n$ be a vector, we denote the element of $\mathbf{h}$ at index i as h_i. Let $l \in R$ be a scalar and f be a scalar-valued function. i.e, $f : \mathbb{R}^n \to \mathbb{R}$. Assume that $l = f(\mathbf{h})$, then the derivative of l w.r.t $\mathbf{h}$ is defined as:

$$\frac{\partial l}{\partial \mathbf{h}} \doteq \left[\frac{\partial l}{\partial h_1}, \frac{\partial l}{\partial h_2}, \ldots, \frac{\partial l}{\partial h_n}\right]^\top . \tag{2.1}$$

Vector-Vector Derivatives

Let $\mathbf{g}$ be a vector-valued function. i.e, $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^n$. Assume that $\mathbf{g}$ can be represented by a vector of scalar-valued functions where $g_i : \mathbb{R}^n \to \mathbb{R}$. Let $\mathbf{v} \in \mathbb{R}^n$ be a vector where $\mathbf{v} = \mathbf{g}(\mathbf{h})$, then each element in $\mathbf{v}$ is a function of all elements in $\mathbf{h}$. i.e, $v_i = g_i(\mathbf{h})$. The derivative of $\mathbf{v}$ w.r.t $\mathbf{h}$ is defined as:

$$\frac{\partial \mathbf{v}}{\partial \mathbf{h}} \doteq \begin{bmatrix} \frac{\partial v_1}{\partial h_1} & \frac{\partial v_2}{\partial h_1} & \cdots & \frac{\partial v_n}{\partial h_1} \\ \frac{\partial v_1}{\partial h_2} & \frac{\partial v_2}{\partial h_2} & \cdots & \frac{\partial v_2}{\partial h_n} \\ & & \cdot & \\ & & \cdot & \\ \frac{\partial v_1}{\partial h_n} & \frac{\partial v_2}{\partial h_n} & \cdots & \frac{\partial v_n}{\partial h_n} \end{bmatrix} . \tag{2.2}$$

Let $\mathbf{u}$ be a vector-valued function. i.e, $\mathbf{u} : \mathbb{R}^n \to \mathbb{R}^n$. Assume that $\mathbf{u}$ can be represented by a vector of element-wise scalar-valued functions where $u_i : \mathbb{R} \to \mathbb{R}$. Let $\mathbf{x} \in \mathbb{R}^n$ be a vector where $\mathbf{x} = \mathbf{u}(\mathbf{h})$, then each element in $\mathbf{x}$ is a function of the corresponding elements in $\mathbf{h}$. i.e, $x_i = u_i(h_i)$. The derivative of $\mathbf{x}$ w.r.t $\mathbf{h}$ is defined as:

$$\frac{\partial \mathbf{x}}{\partial \mathbf{h}} \doteq \left[\frac{\partial x_1}{\partial h_1}, \frac{\partial x_2}{\partial h_2}, \ldots, \frac{\partial x_n}{\partial h_n}\right]^\top . \tag{2.3}$$

[2]This section is practically a summary of the relevant parts of this complete review on Matrix Calculus: https://www.cs.cmu.edu/ mgormley/courses/10601/slides/10601-matrix-calculus.pdf

Scalar-Matrix Derivatives

Let $\mathbf{W} \in \mathbb{R}^{n \times n}$ be a matrix, we denote the element of $\mathbf{W}$ at row i and column j as $w_{i,j}$. Let $l \in R$ be a scalar and f be a function that maps $\mathbf{W}$ to l. i.e, $f : \mathbb{R}^{n \times n} \rightarrow \mathbb{R}$. Then the derivative of l w.r.t $\mathbf{W}$ is defined as:

$$\frac{\partial l}{\partial \mathbf{W}} \doteq \begin{bmatrix} \frac{\partial l}{\partial w_{1,1}} & \frac{\partial l}{\partial w_{1,2}} & \cdots & \frac{\partial l}{\partial w_{1,n}} \\ \frac{\partial l}{\partial w_{2,1}} & \frac{\partial l}{\partial w_{2,2}} & \cdots & \frac{\partial l}{\partial w_{2,n}} \\ & & . & \\ & & . & \\ \frac{\partial l}{\partial w_{n,1}} & \frac{\partial l}{\partial w_{n,2}} & \cdots & \frac{\partial l}{\partial w_{n,n}} \end{bmatrix} \tag{2.4}$$

Vector-Matrix Derivatives

Let $\mathbf{v} \in R^n$ be a vector and $\mathbf{f}$ be a function that maps $\mathbf{W}$ to $\mathbf{v}$. i.e, $\mathbf{f} : \mathbb{R}^{n \times n} \rightarrow \mathbb{R}^n$. Then the derivative of $\mathbf{v}$ w.r.t $\mathbf{W}$ is 3-dimentional matrix with shape $n \times n \times n$, a matrix at index i is defined as:

$$\frac{\partial v_i}{\partial \mathbf{W}} \doteq \begin{bmatrix} \frac{\partial v_i}{\partial w_{1,1}} & \frac{\partial v_i}{\partial w_{1,2}} & \cdots & \frac{\partial v_i}{\partial w_{1,n}} \\ \frac{\partial v_i}{\partial w_{2,1}} & \frac{\partial v_i}{\partial w_{2,2}} & \cdots & \frac{\partial v_i}{\partial w_{2,n}} \\ & & . & \\ & & . & \\ \frac{\partial v_i}{\partial w_{n,1}} & \frac{\partial v_i}{\partial w_{n,2}} & \cdots & \frac{\partial v_i}{\partial w_{n,n}} \end{bmatrix} \tag{2.5}$$

2.4.2 Learning Recurrent Neural Networks Online

For describing RNNs, we use $\mathbf{x}_t$ as the input to the recurrent function. $\mathbf{x}_t$ can simply be the observation at time t or a pre-processed observation. For example, pixel-based observations are usually processed with convolutional layers before passing them to the recurrent function (Kapturowski et al., 2019), in such cases, the input to the RNN is a pre-processed observation.

Consider an RNN with dynamics written as $\mathbf{h}_t = \mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x}_t, \boldsymbol{\psi})$, where $\mathbf{h}_t \in \mathbb{R}^n$ is the recurrent state, $\mathbf{x}_t \in \mathbb{R}^d$ is the input, $\boldsymbol{\psi}$ is a set of the network's learnable parameters, and the subscript t denotes the time step. The agent maps the recurrent state to an output $\hat{y}_t$ to make a prediction or take an action: $\hat{y}_t = g(\mathbf{h}_t, \boldsymbol{\phi})$, where $\boldsymbol{\phi}$ is another set of learnable parameters. Later, the agent receives a loss $\mathcal{L}_t \doteq \mathcal{L}(\hat{y}_t, y_t)$ indicating how far the output is from a target y_t. Finally, the agent updates $\boldsymbol{\psi}$ and $\boldsymbol{\phi}$ to minimize the prediction loss overall the interactions so far, i.e., minimize $\mathcal{L} = \frac{1}{t} \sum_{i=1}^{t} \mathcal{L}_i$, where t is the current time step.

Gradient-based learning algorithms minimize $\mathcal{L}$ by updating $\boldsymbol{\psi}$ and $\boldsymbol{\phi}$ to follow the opposite direction of the gradient $\nabla_{\boldsymbol{\psi}}\mathcal{L}$ and $\nabla_{\boldsymbol{\phi}}\mathcal{L}$, respectively. In this work, we focus on updating $\boldsymbol{\psi}$ since it includes the recurrent parameters used to update the agent state. There are two main gradient-based algorithms widely used to train RNNs: BackPropagation Through Time (BPTT) and Real-Time Recurrent Learning (RTRL).

2.4.3 BackPropagation Through Time

BPTT calculates the gradient, $\nabla_{\boldsymbol{\psi}}\mathcal{L}$, by unfolding the recurrent dynamics through time and incorporating the impact of the parameters on the loss from all observed time steps. Formally, we can write $\nabla_{\boldsymbol{\psi}}\mathcal{L}$ as:

$$\begin{aligned}\nabla_{\boldsymbol{\psi}}\mathcal{L} &= \frac{1}{t}\sum_{i=0}^{t-1}\nabla_{\boldsymbol{\psi}}\mathcal{L}_i \\ &= \frac{1}{t}\sum_{i=0}^{t-1}\frac{\partial\mathcal{L}_i}{\partial\mathbf{h}_i}\frac{\partial\mathbf{h}_i}{\partial\boldsymbol{\psi}}.\end{aligned} \tag{2.6}$$

When calculating $\frac{\partial\mathbf{h}_i}{\partial\boldsymbol{\psi}}$, we need to consider the effect of $\boldsymbol{\psi}$ from all the time steps. To illustrate this effect, consider unrolling the last 2 steps of the RNN dynamics:

$$\begin{aligned}\mathbf{h}_t &= \mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x}_t, \boldsymbol{\psi}) \\ \text{Re-write } \mathbf{h}_{t-1} \text{ as } \mathbf{f}(\mathbf{h}_{t-2}, \mathbf{x}_{t-1}, \boldsymbol{\psi}) \\ &= \mathbf{f}(\mathbf{f}(\mathbf{h}_{t-2}, \mathbf{x}_{t-1}, \boldsymbol{\psi}), \mathbf{x}_t, \boldsymbol{\psi}) \\ \text{Re-write } \mathbf{h}_{t-2} \text{ as } \mathbf{f}(\mathbf{h}_{t-3}, \mathbf{x}_{t-2}, \boldsymbol{\psi}) \\ &= \mathbf{f}(\mathbf{f}(\mathbf{f}(\mathbf{h}_{t-3}, \mathbf{x}_{t-2}, \boldsymbol{\psi}), \mathbf{x}_{t-1}, \boldsymbol{\psi}), \mathbf{x}_t, \boldsymbol{\psi}).\end{aligned} \tag{2.7}$$

Equation 2.7 shows that the network parameters $\boldsymbol{\psi}$ affect the construction of the recurrent state $\mathbf{h}_t$ through two pathways: a direct pathway, i.e., using $\boldsymbol{\psi}$ to evaluate $\mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x}_t, \boldsymbol{\psi})$, and an implicit pathway, i.e., $\boldsymbol{\psi}$ affected constructing all previous recurrent states, $\mathbf{h}_{t-1}, \ldots, \mathbf{h}_1$, and all those recurrent states affected $\mathbf{h}_t$ construction. Thus, to calculate $\frac{\partial\mathbf{h}_t}{\partial\boldsymbol{\psi}}$, we need to consider those two pathways:

$$\frac{\partial\mathbf{h}_t}{\partial\boldsymbol{\psi}} = \frac{\partial\mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x}_t, \boldsymbol{\psi})}{\partial\boldsymbol{\psi}} + \frac{\partial\mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x}_t, \boldsymbol{\psi})}{\partial\mathbf{h}_{t-1}}\frac{\partial\mathbf{h}_{t-1}}{\partial\boldsymbol{\psi}}. \tag{2.8}$$

Once again, we need to consider the two pathways when evaluating $\frac{\partial\mathbf{h}_{t-1}}{\partial\boldsymbol{\psi}}$ in 2.8. For simplicity, let

$\mathbf{J}_t \doteq \frac{\partial \mathbf{h}_t}{\partial \boldsymbol{\psi}}$, $\mathbf{B}_t = \frac{\partial \mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x}_t, \boldsymbol{\psi})}{\partial \boldsymbol{\psi}}$, $\mathbf{C}_t = \frac{\partial \mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x}_t, \boldsymbol{\psi})}{\partial \mathbf{h}_{t-1}}$, and re-write 2.8:

$$\begin{aligned}
\mathbf{J}_t &= \mathbf{B}_t + \mathbf{C}_t \mathbf{J}_{t-1} \\
&\text{Unrolling } \mathbf{J}_{t-1} \\
&= \mathbf{B}_t + \mathbf{C}_t \left(\mathbf{B}_{t-1} + \mathbf{C}_{t-1} \mathbf{J}_{t-2} \right) \\
&= \mathbf{B}_t + \mathbf{C}_t \mathbf{B}_{t-1} + \mathbf{C}_t \mathbf{C}_{t-1} \mathbf{J}_{t-2} \\
&\text{Unrolling } \mathbf{J}_{t-2} \\
&= \mathbf{B}_t + \mathbf{C}_t \mathbf{B}_{t-1} + \mathbf{C}_t \mathbf{C}_{t-1} \left(\mathbf{B}_{t-2} + \mathbf{C}_{t-2} \mathbf{J}_{t-3} \right) \\
&= \mathbf{B}_t + \mathbf{C}_t \mathbf{B}_{t-1} + \mathbf{C}_t \mathbf{C}_{t-1} \mathbf{B}_{t-2} + \mathbf{C}_t \mathbf{C}_{t-1} \mathbf{C}_{t-2} \mathbf{J}_{t-3} \\
&\text{Keep unrolling} \\
&= \mathbf{B}_t + \mathbf{C}_t \mathbf{B}_{t-1} + \mathbf{C}_t \mathbf{C}_{t-1} \mathbf{B}_{t-2} + \mathbf{C}_t \mathbf{C}_{t-1} \mathbf{C}_{t-2} \dots \\
&= \mathbf{B}_t + \sum_{k=1}^{t-1} \left(\prod_{i=k+1}^{t} \mathbf{C}_i \right) \mathbf{B}_k,
\end{aligned} \tag{2.9}$$

where $\mathbf{J}_0 = 0$, assuming we initialize $\mathbf{h}_0 = \mathbf{0}$. Using the results from 2.9, we can now write the expanded $\frac{\partial \mathbf{h}_t}{\partial \boldsymbol{\psi}}$:

$$\begin{aligned}
\frac{\partial \mathbf{h}_t}{\partial \boldsymbol{\psi}} &= \frac{\partial \mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x}_t, \boldsymbol{\psi})}{\partial \boldsymbol{\psi}} + \frac{\partial \mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x}_t, \boldsymbol{\psi})}{\partial \mathbf{h}_{t-1}} \frac{\partial \mathbf{h}_{t-1}}{\partial \boldsymbol{\psi}} \\
&= \frac{\partial \mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x}_t, \boldsymbol{\psi})}{\partial \boldsymbol{\psi}} + \sum_{k=1}^{t-1} \left(\prod_{i=k+1}^{t} \frac{\partial \mathbf{f}(\mathbf{h}_{i-1}, \mathbf{x}_i, \boldsymbol{\psi})}{\partial \mathbf{h}_{i-1}} \right) \frac{\partial \mathbf{f}(\mathbf{h}_{k-1}, \mathbf{x}_k, \boldsymbol{\psi})}{\partial \boldsymbol{\psi}}.
\end{aligned} \tag{2.10}$$

As we can see from 2.10, the agent needs to store all the previous inputs and hidden states to evaluate $\frac{\partial \mathbf{h}_t}{\partial \boldsymbol{\psi}}$ which is impractical; the computation and memory complexity will be increasing as t increases.

Truncated-BackPropagation Through Time

Williams and Peng (1990) introduced Truncated-BackPropagation Through Time (T-BPTT) which solves the issue of increasing memory and computational complexities of BPTT. In T-BPTT, we specify a truncation length T which controls the number of steps taken into consideration when calculating the gradient in 2.10. Hence, the computation and memory complexities for learning the parameters are fixed for all the time steps and depend on T.

T-BPTT restricts what the agent can remember, and T controls this restriction. The agent can only retain information from previous time steps up to T steps back as the gradient information from steps further than T is assumed to be zero. In problems where we have some domain knowledge, we can select a suitable T to solve the task, and T-BPTT is recommended. However, in many applications, we do not know beforehand which T is suitable. Thus, specifying T can be limiting and a cause for inadequate performance.

We now write the truncated version of 2.9 which takes into consideration the gradient from the last T steps only:

$$\mathbf{J}_t = \mathbf{B}_t + \sum_{k=t-T}^{t-1} \left(\prod_{i=k+1}^{t} \mathbf{C}_i \right) \mathbf{B}_k. \tag{2.11}$$

Using results from 2.11, we then write the approximated gradient of the loss w.r.t the learnable parameters:

$$\begin{aligned} \nabla_{\boldsymbol{\psi}} \mathcal{L} &= \sum_{i=t-T}^{t} \frac{\partial \mathcal{L}_i}{\partial \mathbf{h}_i} \frac{\partial \mathbf{h}_i}{\partial \boldsymbol{\psi}} \\ &= \sum_{i=t-T}^{t} \frac{\partial \mathcal{L}_i}{\partial \mathbf{h}_i} \left(\frac{\partial \mathbf{f}(\mathbf{h}_{i-1}, \mathbf{x}_i, \boldsymbol{\psi})}{\partial \boldsymbol{\psi}} + \sum_{k=i-T}^{i-1} \left(\prod_{j=k+1}^{i} \frac{\partial \mathbf{f}(\mathbf{h}_{j-1}, \mathbf{x}_j, \boldsymbol{\psi})}{\partial \mathbf{h}_{j-1}} \right) \frac{\partial f(\mathbf{h}_{k-1}, \mathbf{x}_k, \boldsymbol{\psi})}{\partial \boldsymbol{\psi}} \right). \end{aligned} \tag{2.12}$$

Algorithm 2 Online prediction with T-BPTT

```
Inputs: a differentiable parametrization of the recurrent function $\mathbf{f} : \mathbb{R}^n \times \mathbb{R}^d \to \mathbb{R}^n$
Inputs: a differentiable parametrization mapping the recurrent state to a prediction $g : \mathbb{R}^n \to \mathbb{R}$
Initialize: the set of parameters $\boldsymbol{\psi}$ and $\boldsymbol{\phi}$.
Initialize: $O(j)$, for all $j \in \{t-T, \dots, t\}$.                ▷ A list to save the inputs.

for each input $\mathbf{x_t}$ do
    Forward Pass: update the recurrent state, $\mathbf{h}_t = \mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x_t})$
    and make a prediction, $\hat{y}_t = g(\mathbf{h}_t)$.
    Evaluate: recieve a loss $\mathcal{L}_t(\hat{y}_t, y_t)$.
    Append to Memory: $O(t) \leftarrow \mathbf{x_t}$.
    Backward Pass: LEARN($O$)                ▷ This call may not happen every time step.

function LEARN($O$)                ▷ Taking last $T$ observations.
    for $i = t - T \to t$ do
        Calculate and save $\frac{\partial \mathcal{L}_i}{\partial \mathbf{h}_i}$, $\frac{\partial \mathbf{f}(\mathbf{h}_{i-1}, \mathbf{x}_i, \boldsymbol{\psi})}{\partial \boldsymbol{\psi}}$, and $\frac{\partial \mathbf{f}(\mathbf{h}_{i-1}, \mathbf{x}_i, \boldsymbol{\psi})}{\partial \mathbf{h}_{i-1}}$.
    for $i = t - T \to t$ do
        Calculate and save $\frac{\partial \mathbf{h}_i}{\partial \boldsymbol{\psi}}$                ▷ From 2.12.
    Calculate $\nabla_{\boldsymbol{\psi}} \mathcal{L} = \sum_{i=t-T}^{t} \frac{\partial \mathcal{L}_i}{\partial \mathbf{h}_i} \frac{\partial \mathbf{h}_i}{\partial \boldsymbol{\psi}}$.
    Update $\boldsymbol{\psi}$ to follow the opposite direction of $\nabla_{\boldsymbol{\psi}} \mathcal{L}$.
```

In algorithm 2, we show an example for using T-BPTT in online predction. At each time step t, the agent gets an input $\mathbf{x}_t$, updates its recurrent state $\mathbf{h}_t$ and makes a prediction $\hat{y}_t$. The agent then receives a loss $\mathcal{L}_t$, and finally calls the *LEARN* function. The *LEARN* function calculates the gradient of the loss w.r.t the learnable parameters $\boldsymbol{\psi}$ using 2.12. The agent then updates its parameters $\boldsymbol{\psi}$ to follow the opposite direction of that gradient. Both the memory and the computational complexities of this algorithm are propotional to the truncation length specified, T. In section 2.5, we will dive into the exact memory and computational complexities of T-BPTT.

2.4.4 Real-Time Recurrent Learning

Williams and Zipser (1989) introduced the Real-time Recurrent Learning algorithm (RTRL) as a learning algorithm for continual recurrent learning. RTRL employs the recurrent formulation of the gradient in 2.8; instead of unrolling $\frac{\partial \mathbf{h}_{t-1}}{\partial \boldsymbol{\psi}}$ further back in time, RTRL saves its calculated value from the previous time step and use it later when needed. It is worth emphasizing that after the agent updates its parameters, the gradient information saved from previous time steps would be stale, i.e., calculated w.r.t old parameters, however, under the assumption of small learning rates, RTRL is known to converge. The gradient formulation of RTRL can be written as:

$$\begin{aligned}\nabla_{\boldsymbol{\psi}}\mathcal{L} &= \sum_{i=0}^{t}\frac{\partial \mathcal{L}_i}{\partial \mathbf{h}_i}\frac{\partial \mathbf{h}_i}{\partial \boldsymbol{\psi}} \\ &= \sum_{i=0}^{t}\frac{\partial \mathcal{L}_i}{\partial \mathbf{h}_i}\left(\frac{\partial \mathbf{f}(\mathbf{h}_{i-1},\mathbf{x}_i,\boldsymbol{\psi})}{\partial \boldsymbol{\psi}} + \frac{\partial \mathbf{f}(\mathbf{h}_{i-1},\mathbf{x}_i,\boldsymbol{\psi})}{\partial \mathbf{h}_{i-1}}\frac{\partial \mathbf{h}_{i-1}}{\partial \boldsymbol{\psi}}\right)\end{aligned} \tag{2.13}$$

Algorithm 3 shows an online learner using RTRL to update its learnable parameters.

2.5 Complexity of Training RNNs Online

In this section, we describe the recurrent learning algorithms' memory and computational complexities.

2.5.1 Complexity analysis

Now, we move to analyzing the computational and memory complexity of RNNs. Widely used RNNs such as Gated Recurrent Units (GRUs) (Cho et al., 2014) and Long-Short Term Memory (LSTM)(Hochreiter and Schmidhuber, 1997) boil down to the following formulation:

Algorithm 3 Online prediction with RTRL

```
Inputs: a differentiable parametrization of the recurrent function $\mathbf{f} : \mathbb{R}^n \times \mathbb{R}^d \to \mathbb{R}^n$
Inputs: a differentiable parametrization mapping the recurrent state to a prediction $g : \mathbb{R}^n \to \mathbb{R}$
Initialize: the set of parameters $\boldsymbol{\psi}$ and $\boldsymbol{\phi}$.
Initialize: an empty matrix to store the gradient from previous time step, $\mathbf{J}_{-1}$.

for each input $\mathbf{x_t}$ do
    Forward Pass: update the recurrent state, $\mathbf{h}_t = \mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x_t})$
    and make a prediction, $\hat{y}_t = g(\mathbf{h}_t)$.
    Evaluate: recieve a loss $\mathcal{L}_t(\hat{y}_t, y_t)$.
    Backward Pass: LEARN($\mathbf{x_t}$,$\mathcal{L}_t$,$\mathbf{h}_{t-1}$).

function LEARN($\mathbf{x_t}$,$\mathcal{L}_t$,$\mathbf{h}_{t-1}$)
    Calculate $\frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t}$, $\frac{\partial \mathbf{f}(\mathbf{h}_{t-1},\mathbf{x}_t,\boldsymbol{\psi})}{\partial \boldsymbol{\psi}}$, and $\frac{\partial \mathbf{f}(\mathbf{h}_{t-1},\mathbf{x}_t,\boldsymbol{\psi})}{\partial \mathbf{h}_{t-1}}$
    Calculate $\mathbf{J}_t = \frac{\partial \mathbf{f}(\mathbf{h}_{t-1},\mathbf{x}_t,\boldsymbol{\psi})}{\partial \boldsymbol{\psi}} + \frac{\partial \mathbf{f}(\mathbf{h}_{t-1},\mathbf{x}_t,\boldsymbol{\psi})}{\partial \mathbf{h}_{t-1}} \mathbf{J}_{t-1}$        ▷ $J_t = \frac{\partial \mathbf{h}_t}{\partial \boldsymbol{\psi}}$
    Calculate $\nabla_{\boldsymbol{\psi}} \mathcal{L}_t = \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t} \mathbf{J}_t$
    Update $\boldsymbol{\psi}$ to follow the opposite direction of $\nabla_{\boldsymbol{\psi}} \mathcal{L}_t$
```

$$\mathbf{h}_t \doteq \mathbf{f}(\mathbf{W}_x \mathbf{x}_t + \mathbf{W}_h \mathbf{h}_{t-1}). \tag{2.14}$$

Where $\mathbf{h}_t \in \mathbb{R}^n$ is the recurrent state at time t, $\mathbf{h}_{t-1} \in \mathbb{R}^n$ is the recurrent state at the previous time step $t-1$, $\mathbf{x}_t \in \mathbb{R}^d$ is the input at time t, and both $\mathbf{W}_x \in \mathbb{R}^{n \times d}$ and $\mathbf{W}_h \in \mathbb{R}^{n \times n}$ are the learnable paramters.

In the following subsections, we analyze the computation and memory complexity for training RNNs. The computation complexity is measured in terms of the number of Floating-Point Operations (FLOPs). However, our estimates might differ from the actual number of FLOPs due to other factors such as the hardware and the actual implementation.

Forward Pass in RNNs

Table 2.1 shows the operations and the FLOPs required for doing a forward pass, i.e., passing an input through the recurrent function and evaluating its output. We can see that the number of FLOPs for doing a forward pass is $2nd + 2n^2 + 2n$, where n is the hidden state dimension and d is the input dimension. In a moderate-sized architecture $n \gg d$. Thus, we can write the computational complexity of doing a forward pass in RNN as $\mathcal{O}(n^2)$. The memory complexity for the forward pass is also $\mathcal{O}(n^2)$, which is the memory required to save the learnable parameters.

Operation	FLOPs
$\mathbf{W}_x\mathbf{x}_t$	$2nd$
$\mathbf{W}_h\mathbf{h}_{t-1}$	$2n^2$
$\mathbf{W}_x\mathbf{x}_t + \mathbf{W}_h\mathbf{h}_{t-1}$	n
$\mathbf{f}(\cdot)$	n

Table 2.1: The operations performed in a forward pass of an RNN and their computational requirements.

Backward Pass in RNNs

The computation and memory complexity of the backward pass, i.e., updating the learnable parameters, depend on the learning algorithm. We look at the computation and memory for both T-BPTT and RTRL learning algorithms introduced earlier in sections 2.4.3 and 2.4.4, respectively.

T-BPTT

T-BPTT, as shown in algorithm 2, unrolls the RNN dynamics for T steps, resulting in a complexity $T\times$ the complexity of doing a single forward pass. i.e., $\mathcal{O}(Tn^2)$, and also requires and an additional memory that is $\mathcal{O}(Td)$ to store previous inputs. Thus, an RNN that uses T-BPTT to learn its parameters requires a total computation of $\mathcal{O}((T+1)n^2)$ and total memory of $\mathcal{O}(n^2 + Td + nd)$ for doing (1 forward pass + 1 backward pass) each time step. Depending on the number of learnable parameters, the memory bottleneck for T-BPTT can either be the memory required to store the previous inputs or the memory required to store the learnable parameters. However, the computational bottleneck is always dependent on the truncation length. Figure 2.4 shows an example of the computational complexity of T-BPTT as a function of the truncation length T and the hidden state dimension.

RTRL

RTRL, as shown in algorithm 3 does not unroll the RNN back in time, altenatively, it stores the relevant gradient information, i.e, $\frac{\partial \mathbf{h}_{t-1}}{\partial \boldsymbol{\psi}}$, from the previous time step. From the definition of RNN 2.14:

$$\frac{\partial \mathbf{h}_{t-1}}{\partial \boldsymbol{\psi}} = \left\{ \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{W}_x}, \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{W}_h} \right\} \tag{2.15}$$

Each of the two components is a derivative of a vector w.r.t a matrix and according to 2.5 has the dimensions of $n \times d \times n$ and $n \times n \times n$, respectively. Thus, RTRL has a memory complexity of $\mathcal{O}(n^3)$, if we assume that the number of hidden units is greater than the input dimension.

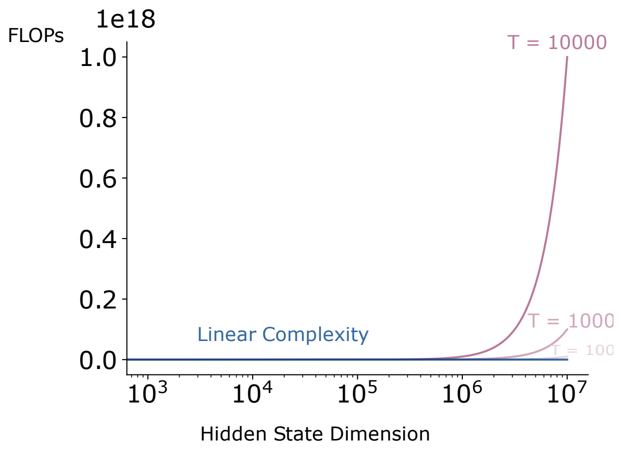
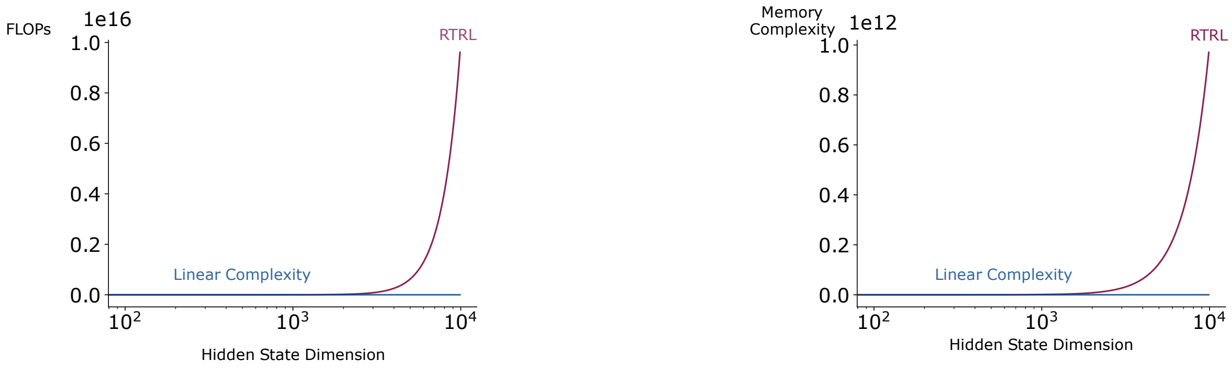


Figure 2.4: The computational complexity of T-BPTT as the truncation length T increases. The horizontal axis shows the number of features in the RNN and the vertical axis shows the number of FLOPs required per update.



(a) Computational complexity of RTRL.

(b) Memory complexity of RTRL.

Figure 2.5: The computational and memory complexities of RTRL against a linear complexity as a function of the hidden state dimension.

Calculating the gradient according to 2.13 requires calculating the product $\frac{\partial \mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x}_t, \boldsymbol{\psi})}{\partial \mathbf{h}_{t-1}} \frac{\partial \mathbf{h}_{t-1}}{\partial \boldsymbol{\psi}}$, where $\frac{\partial \mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x}_t, \boldsymbol{\psi})}{\partial \mathbf{h}_{t-1}} \in \mathbb{R}^{n \times n}$ is the derivative of a vector w.r.t a vector as in 2.2 and $\frac{\partial \mathbf{h}_{t-1}}{\partial \boldsymbol{\psi}} \in \{\mathbb{R}^{n \times d \times n}, \mathbb{R}^{n \times n \times n}\}$ is a derivative of a vector w.r.t a matrix as in 2.5. This product has a computational complexity of $\mathcal{O}(n^4)$. Figure 2.5 shows an example of the memory and computational complexities of RTRL as a function of the hidden state dimension compared to linear complexity, which would be ideal for an algorithm. We can see that even for a small hidden dimension, RTRL's memory and computational complexities grow exponentially.

The ability to learn from arbitrary long sequences without truncating the history and with a constant cost per update is appealing. However, the memory and computational complexities of

RTRL render it impractical. On the other hand, T-BPTT has a more plausible complexity, but it suffers from the truncation bias. In the next chapter, we introduce a novel architecture for which the exact RTRL updates have a linear complexity. We also discuss the differences between our proposed architecture and other RNN approximations from the literature.

Chapter 3

Recurrent Trace Units

This chapter motivates and introduces Recurrent Trace Units (RTUs), the main contribution of this thesis. We first discuss a diagonal formulation for the linear recurrence. Then, we introduce a couple of illustrative examples highlighting two fundamental properties of RNNs. Finally, we introduce RTUs and derive an exact RTRL update for RTUs with linear computational and memory complexities.

3.1 Linear Diagonal Recurrent Dynamics

A linear recurrent layer has an equivalent diagonal form with fewer learnable parameters and the same representational powers. The idea behind deriving this diagonal form is simple; instead of learning a dense matrix, a learner can learn the eigenvalues of that matrix. Formally, a linear dense recurrent layer has the form:

$$\mathbf{h}_t \doteq \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t. \tag{3.1}$$

The only difference between 3.1 and the RNN dynamics in 2.14 is dropping the non-linear function $\mathbf{f}$. For a matrix $\mathbf{W}_h$ with n linearly independent eigenvectors, we can re-write $\mathbf{W}_h$ in terms of its eigenvectors as:

$$\mathbf{W}_h = \mathbf{P}\,\mathbf{\Lambda}\,\mathbf{P}^{-1}, \tag{3.2}$$

where $\mathbf{P}$ contains the n linearly independent eigenvectors and Λ is a diagonal matrix with the diagonal elements being the corresponding eigenvalues. Substituting 3.2 in 3.1 and multiplying both sides by $\mathbf{P}^{-1}$, we get:

$$\begin{aligned} \mathbf{h}_t &= \ \mathbf{P\Lambda P}^{-1}\ \mathbf{h}_{t-1}\ +\ \mathbf{W}_x\ \mathbf{x}_t. \\ \mathbf{P}^{-1}\mathbf{h}_t &= \mathbf{\Lambda P}^{-1}\ \mathbf{h}_{t-1}\ +\ \mathbf{P}^{-1}\ \mathbf{W}_x\ \mathbf{x}_t. \end{aligned} \tag{3.3}$$

Finally, by defining $\overline{\mathbf{h}}_t \doteq \mathbf{P}^{-1}\ \mathbf{h}_t$ and $\overline{\mathbf{W}}_x\ \doteq\ \mathbf{P^{-1}}\ \mathbf{W}_x$, then substituting in 3.3, we get:

$$\overline{\mathbf{h}}_t = \mathbf{\Lambda}\overline{\mathbf{h}}_{t-1} + \overline{\mathbf{W}}_x\ \mathbf{x}_t. \tag{3.4}$$

Using the recurrent formulation from 3.4, the learner only needs to learn the n eigenvalues instead of the full $\mathbf{W}_h$ matrix.

Learning a diagonal RNN has been previously explored in the literature. Columnar Networks (Javed et al., 2023) and element-wise LSTM (eLSTM) (Irie, Gopalakrishnan, and Schmidhuber, 2023) both proposed using a diagonal form of the recurrent dynamics. The appeal of diagonal RNNs for online learning is due to having fewer learnable parameters in the recurrence. i.e., n instead of $n \times n$, which leads to a scalable RTRL algorithm for online learning. However, as noted by Javed et al. (2023), Columnar Networks had poor performance when used alone, and they moved to combine them with a constructive approach to achieve better performance.

Both Columnar Networks and eLSTM had real-valued diagonal elements in their recurrent formulation. However, from the derivation we did in 3.3 and 3.4, we notice that having only real-valued diagonals could be limiting; there is no guarantee that all eigenvalues are real. Orvieto et al. (2023) and Huang et al. (2023) re-explored the idea of diagonal RNNs; they showed that having complex-valued diagonal elements performed better compared to only real-valued elements.

3.2 Illustrative Examples

In this section, we discuss two follow-up questions on learning a diagonal RNN:

1. Do complex eigenvalues naturally emerge when learning under partial observability? This question should help in understanding whether or not it is necessary to have complex-valued diagonal elements.

2. If complex eigenvalues emerge, how do different representations for complex numbers affect learning? A complex number can be represented mathematically in three ways that are all equivalent and describe the same number. It is unclear if different complex representations have different learning properties.

To answer the first question, we use an MDP introduced by Sutton (2020a), we call it *Three States World*. As the name suggests, this MDP has only 3 states and no actions. The states are s_1, s_2, and s_3. If the agent is in either s_1 or s_2, it transitions to any of the three states with equal probabilities. However, if the agent is in s_3, it transitions to the state preceded by s_3. A sequence of observations would look like $1, 3, 1, 2, 2, 3, 2, \cdots$, and we ask the agent to predict the next observation (Sutton, 2020a). Figure 3.1 shows an illustration of this task. To succeed in this task, the agent needs to remember one cue that happened one step back and only use this memory in state 3.

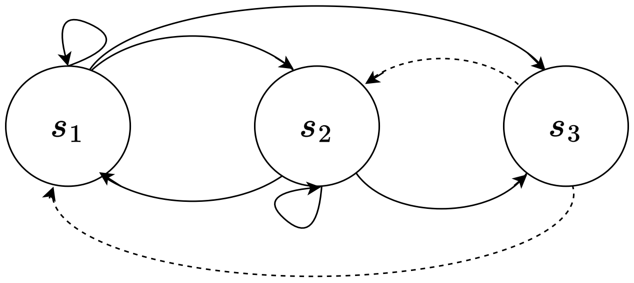


Figure 3.1: Illustration of the Three States World MDP. We used dashed lines for the transitions starting in s_3 to distinguish them.

We trained an agent with a standard RNN to solve the Three State World. The RNN had 3 hidden states. We used T-BPTT to learn the network parameters with truncation length 2; that is the whole history needed to predict the next observation. We swept over multiple learning rates for the RNN in this setting and chose the learning rate that achieved the best performance. Then, we calculated the number of complex eigenvalues of the matrix $\mathbf{W}_h$ after each parameter update and recorded it. Since we have 3 hidden states, we know that the matrix $\mathbf{W}_h$ in 2.14 is $\in \mathbb{R}^{3\times 3}$ and could have at most 2 complex eigenvalues.

Figure 3.2 shows the percentage of correct predictions the agent achieves over time. As we can see, this agent can completely solve the problem and achieves 100% correct predictions. Figure 3.3 shows the number of complex eigenvalues we recorded during training, and we can see that complex eigenvalues appear frequently while learning. These results suggest that a standard RNN could have complex eigenvalues even in this simple task, which answers our first question. From this experiment, we conclude that to represent RNNs in the diagonal form, we need to have complex-valued diagonal elements.

We now move to the second question: How do we represent complex numbers, and how do different representations affect learning? A complex number can be represented in three ways: $a + bi$ (the real representation), $r\exp(i\theta)$ (the exponential representation), and $r(\cos(\theta) + i\sin(\theta))$ (the cosine representation). Mathematically, these three representations are equivalent, but do they affect learning differently? Orvieto et al. (2023) partially addressed this question empirically on a simple task. They showed that using the exponential representation resulted in a better-behaved

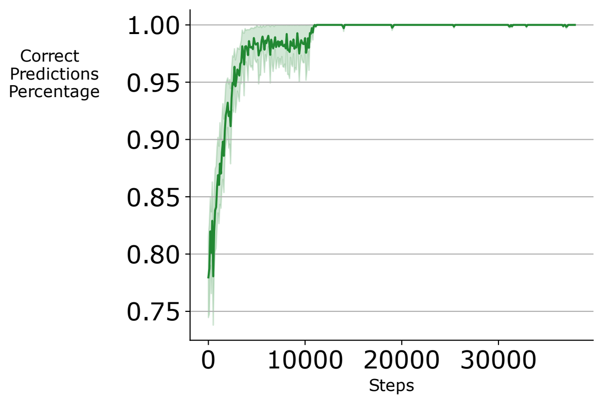


Figure 3.2: The percentage of correct predictions when training an RNN in the Three States World. The solid green line is the average over 30 runs and the shaded area is the standard error.

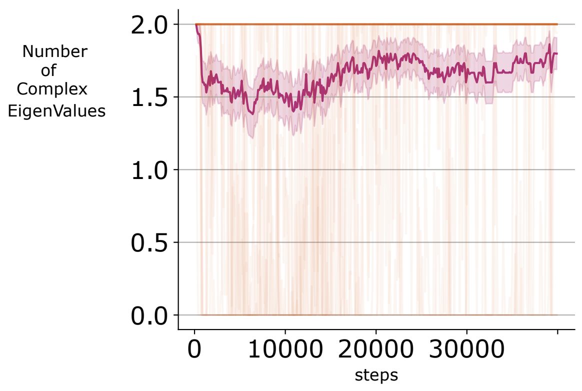


Figure 3.3: Number of complex eigenvalues when training an RNN in the Three States World. The red line represents the mean averaged over 30 runs, the shaded red area represents the standard error, and the rest of the lines represent individual runs.

loss function than the real parametrization.

We take a different approach to address the second question. A known issue when training RNNs is the vanishing/exploding gradient (Pascanu, Mikolov, and Bengio, 2013), which happens when the gradient components of the loss w.r.t the parameters either grow to large numbers and explode or shrink and vanish. In the following example, we reason about the vanishing/exploding gradient issue in a single recurrent unit using the three complex representations. For a recurrent unit with no inputs:

$$\begin{aligned} h_t &= \lambda h_{t-1} \\ &= \lambda^t h_0, \end{aligned} \tag{3.5}$$

where h_0 is the initial hidden state, consider a case where we want h_t to reach a specific state h^* after k time steps. The error at time k is:

$$\varepsilon_k = (h_k - h^*)^2. \tag{3.6}$$

Assuming λ is complex-valued, we now consider the gradient from each of the three representations.

Real Representation $a + bi$

Substituting λ in 3.5 with the real representation, we get:

$$\begin{aligned} h_t &= (a + bi)h_{t-1} \\ &= (a + bi)^t h_0 \\ &= h_0 \sum_{k=0}^{t} \binom{t}{k} a^{t-k} (bi)^k \\ &= h_0 \sum_{k=0}^{t} \binom{t}{k} a^{t-k} b^k i^k. \end{aligned} \tag{3.7}$$

Then it follows that the gradient w.r.t the learnable parameters i.e, a and b is:

$$\begin{aligned} \frac{\partial h_t}{\partial a} &= h_0 \sum_{k=0}^{t} \binom{t}{k} (t-k) a^{t-k-1} b^k i^k \\ \frac{\partial h_t}{\partial b} &= h_0 \sum_{k=0}^{t} \binom{t}{k} k a^{t-k} b^{k-1} i^k. \end{aligned} \tag{3.8}$$

Based on 3.8, to prevent the gradient from vanishing/exploding, we need to restrict both $|a|$ and $|b|$ to be $\in (0, 1]$. This restricts both the magnitude and the phase of the complex number to be $\in (0, 1]$ and $\in (0, \frac{\pi}{2}]$, respectively.

Exponential Representation $r\exp(i\theta)$

Substituting λ in 3.5 with the exponential representation, we get:

$$\begin{aligned} h_t &= r\exp(i\theta)h_{t-1} \\ &= (r\exp(i\theta))^t h_0 \\ &= (r)^t \exp(it\theta)h_0. \end{aligned} \tag{3.9}$$

Then the gradient w.r.t the learnable parameters i.e., r and θ is:

$$\begin{aligned} \frac{\partial h_t}{\partial r} &= tr^{t-1}\exp(it\theta)h_0 \\ \frac{\partial h_t}{\partial \theta} &= r^t \exp(it\theta) * ith_0. \end{aligned} \tag{3.10}$$

Based on 3.10, to prevent the gradient from vanishing/exploding, we need to restrict $r \in (0, 1]$.

Cosine Representation $r(\cos(\theta) + i\sin(\theta))$

Substituting λ in 3.5 with the cosine representation, we get:

$$\begin{aligned} h_t &= r(\cos(\theta) + i\sin(\theta))h_{t-1} \\ &= (r(\cos(\theta) + i\sin(\theta)))^t h_0 \\ &= (r)^t (\cos(t\theta) + i\sin(t\theta))h_0. \end{aligned} \tag{3.11}$$

Then the gradient w.r.t the learnable parameters i.e., r and θ is:

$$\begin{aligned} \frac{\partial h_t}{\partial r} &= tr^{t-1}(\cos(t\theta) + i\sin(t\theta))h_0 \\ \frac{\partial h_t}{\partial \theta} &= r^t (it\cos(t\theta) - t\sin t\theta)h_0. \end{aligned} \tag{3.12}$$

Based on 3.12, to prevent the gradient from vanishing/exploding, we need to restrict $r \in (0, 1]$.

From the above analysis, we conclude that the real representation is the most restrictive since it restricts both the magnitude and the phase of the complex number. These results align with the empirical experiments presented by Orvieto et al.(2023), where they showed that the exponential representation outperformed the real representation on a simple task. However, it is still unclear whether there is an advantage to using exponential representation over the cosine representation and vice versa.

3.3 Recurrent Trace Units

We have established in the previous section that having complex-valued diagonal elements is fundamental when using the diagonal recurrent formulation. In this section, we discuss a few issues that arise when using complex-valued diagonal elements. Then, we introduce Recurrent Trace Units (RTUs), a new complex-valued diagonal RNN that addresses these issues.

We use Linear Recurrent Units (LRUs, Orvieto et al., 2023) as an example of complex-valued diagonal RNNs since they have shown promising results. Despite their good performance on offline tasks, LRUs are still unsuitable for continual online learning for a few reasons. Firstly, LRUs learn their parameters using T-BPTT, which presents some issues when agents continually learn online; as discussed in 2.4.3, T-BPTT has an inherent trade-off between computational complexity and retaining history. Secondly, as the name suggests, LRUs have linear recurrent dynamics, restricting the class of function they could represent (El-Naggar, Madhyastha, and Weyde, 2023). Finally, since the recurrent states are now complex-valued, and most of the desired outputs are real numbers when feeding the recurrent state to the following layers of the network, LRUs consider only the real part of the recurrent state, which means that we might lose some of the representations encoded by the complex part.

We illustrate the implications of the last point, considering only the real part of the recurrent state, with an example of a single complex-valued recurrent unit. For simplicity, assume h_t is a complex-valued recurrent unit with no inputs:

$$h_t = \lambda h_{t-1}, \tag{3.13}$$

where $\lambda = r(\cos(\theta) + i\sin(\theta))$, and both r and θ are learnable parameters. Then, to generate an output y_t, we consider only the real part of h_t:

$$y_t = wRe\{h_t\}, \tag{3.14}$$

where w is a learnable parameter. This formalization leads to a gradient of the form:

$$\begin{aligned}\frac{\partial y_t}{\partial r} &= w\cos(\theta)h_{t-1} + wr\cos(\theta)\frac{\partial h_{t-1}}{\partial r}\\ \frac{\partial y_t}{\partial \theta} &= -wr\sin(\theta)h_{t-1} + wr\cos(\theta)\frac{\partial h_{t-1}}{\partial \theta}.\end{aligned} \tag{3.15}$$

To show the missing gradient information in 3.15, we first consider a fundamental property of complex numbers: multiplying by a complex number z is equivalent to a rotation by the matrix $\begin{bmatrix} Re\{z\} & -Img\{z\} \\ Img\{z\} & Re\{z\} \end{bmatrix}$ and a scale by $\sqrt{Re\{z\}^2 + Img\{z\}^2}$. We use this property to re-write 3.13

and 3.14 as:

$$\begin{aligned} h_t^{c_1} &= r\cos(\theta)h_{t-1}^{c_1} - r\sin(\theta)h_{t-1}^{c_2} \\ h_t^{c_2} &= r\cos(\theta)h_{t-1}^{c2} + r\sin(\theta)h_{t-1}^{c_1} \\ y_t &= w_1 h_t^{c_1} + w_2 h_t^{c_2}. \end{aligned} \tag{3.16}$$

We now write the gradient using this new formulation:

$$\begin{aligned} \frac{\partial y_t}{\partial r} &= w_1\cos(\theta)h_{t-1}^{c_1} + w_1 r\cos(\theta)\frac{\partial h_{t-1}^{c_1}}{\partial r} - w_1\sin(\theta)h_{t-1}^{c_2} - w_1 r\sin(\theta)\frac{\partial h_{t-1}^{c_2}}{\partial r} + \\ &\quad w_2\cos(\theta)h_{t-1}^{c_2} + w_2 r\cos(\theta)\frac{\partial h_{t-1}^{c_2}}{\partial r} + w_2\sin(\theta)h_{t-1}^{c_1} + w_2 r\sin(\theta)\frac{\partial h_{t-1}^{c_1}}{\partial r} \\ \frac{\partial y_t}{\partial \theta} &= -w_1 r\sin(\theta)h_{t-1}^{c_1} + w_1 r\cos(\theta)\frac{\partial h_{t-1}^{c_1}}{\partial \theta} - w_1 r\cos(\theta)h_{t-1}^{c_2} - w_1 r\sin(\theta)\frac{\partial h_{t-1}^{c_2}}{\partial \theta} - \\ &\quad w_2 r\sin(\theta)h_{t-1}^{c_2} + w_2 r\cos(\theta)\frac{\partial h_{t-1}^{c_2}}{\partial \theta} + w_2\cos(\theta)h_{t-1}^{c_1} + w_2 r\sin(\theta)\frac{\partial h_{t-1}^{c_1}}{\partial \theta}. \end{aligned} \tag{3.17}$$

Comparing 3.17 and 3.15, we can see that using only the real part of the recurrent state as in 3.14 leads to a loss of information in the gradient which could affect learning.

We now introduce Recurrent Trace Units (RTUs) addressing previous issues with complex-valued diagonal RNNs. First, it is safe to assume that the matrix $\mathbf{\Lambda}$ in 3.4 has only complex eigenvalues; a complex eigenvalue can be easily turned into a real one by setting the imaginary component to 0. Since all eigenvalues are now learnable parameters, the learner can choose which ones to convert to real and which remain complex. Secondly, we use the rotational representations of complex numbers as it allows for generating real-valued outputs without losing gradient information. To use this representation, we need a complex representation of the form: $\text{Real} + \imath\text{Imaginary}$, which restricts us to either the real or the cosine representation. As we previously discussed in 3.2, the real representation is the most restrictive, so it is natural to use the cosine representation here. We can now write the matrix $\mathbf{\Lambda}$ as blocks of rotation matrices instead of explicit complex numbers:

$$\mathbf{\Lambda} = \begin{bmatrix} \mathbf{c}_1 & & \\ & \cdots & \\ & & \mathbf{c}_n \end{bmatrix}, \tag{3.18}$$

where

$$\begin{aligned} \mathbf{c}_k &= \sqrt{a_k^2 + b_k^2}\begin{bmatrix} a_k & -b_k \\ b_k & a_k \end{bmatrix} \\ &= \boldsymbol{\nu}_k \begin{bmatrix} \cos(\theta_k) & -\sin(\theta_k) \\ \sin(\theta_k) & \cos(\theta_k) \end{bmatrix}. \end{aligned} \tag{3.19}$$

We re-write 3.4 as:

$$\begin{aligned}\mathbf{h}_t^{c1} &= \boldsymbol{\nu}\cos(\boldsymbol{\theta}) \odot \mathbf{h}_{t-1}^{c1} - \boldsymbol{\nu}\sin(\boldsymbol{\theta}) \odot \mathbf{h}_{t-1}^{c2} + \mathbf{W}_x^{c1}\mathbf{x}_t,\\ \mathbf{h}_t^{c2} &= \boldsymbol{\nu}\cos(\boldsymbol{\theta}) \odot \mathbf{h}_{t-1}^{c2} + \boldsymbol{\nu}\sin(\boldsymbol{\theta}) \odot \mathbf{h}_{t-1}^{c1} + \mathbf{W}_x^{c2}\ \mathbf{x}_t.\end{aligned} \tag{3.20}$$

Then, we non-linearly combine the new recurrent states into one state:

$$\mathbf{h}_t = [\mathbf{f}(\mathbf{h}_t^{c1}); \mathbf{f}(\mathbf{h}_t^{c2})]. \tag{3.21}$$

Besides having complex-valued diagonals, there were two additional contributors to the improved performance of LRUs. The first was learning logarithmic representations of the learnable parameters rather than learning them directly. i.e., instead of learning ν and θ, the network learns $\nu^{\log} \doteq \log(\nu)$ and $\theta^{\log} \doteq \log(\theta)$. We then exponentiate these parameters before using them to get our $\nu = \exp(\nu^{\log})$ and $\theta = \exp(\theta^{\log})$. The second was normalizing the mapped input representations. Orvieto et al.(2023) empirically showed that these modifications improve LRUs' performance. We similarly noted that these modifications improve the performance when applied to our recurrent formulation.

We now write the final formulation of RTUs:

$$\begin{aligned}\mathbf{h}_t^{c1} &= \mathbf{g}(\boldsymbol{\nu},\boldsymbol{\theta}) \odot \mathbf{h}_{t-1}^{c1} - \boldsymbol{\phi}(\boldsymbol{\nu},\boldsymbol{\theta}) \odot \mathbf{h}_{t-1}^{c2} + \boldsymbol{\gamma} \odot \mathbf{W}_x^{c1}\mathbf{x}_t.\\ \mathbf{h}_t^{c2} &= \mathbf{g}(\boldsymbol{\nu},\boldsymbol{\theta}) \odot \mathbf{h}_{t-1}^{c2} + \boldsymbol{\phi}(\boldsymbol{\nu},\boldsymbol{\theta}) \odot \mathbf{h}_{t-1}^{c1} + \boldsymbol{\gamma} \odot \mathbf{W}_x^{c2}\mathbf{x}_t.\\ \mathbf{h}_t &= [\mathbf{f}(\mathbf{h}_t^{c1}); \mathbf{f}(\mathbf{h}_t^{c2})].\end{aligned} \tag{3.22}$$

where

$$\begin{aligned}g(\nu_j, \theta_j) &= \exp(\nu_j^{\log}) \cos(\exp(\theta_j^{\log})),\\ \phi(\nu_j, \theta_j) &= \exp(\nu_j^{\log}) \sin(\exp(\theta_j^{\log})),\end{aligned} \tag{3.23}$$

and γ is a normalization factor, $\gamma_j = (1 - (\exp(-\nu_j))^2)^{\frac{1}{2}}$ These modifications do not affect our previous analysis; $\boldsymbol{\gamma}$ can be absorbed by $\mathbf{W}$, and we exponentiate the parameters before using them in the formulation. Finally, the name RTUs comes from the fact that each element in the recurrent state in 3.22 is a decaying trace of two elements from the previous time step.

3.4 Real-Time Recurrent Learning for RTUs

In this section, we derive Real-Time Recurrent Learning rules for RTUs. We call the combination of RTRL and RTUs, *the RT2 algorithm.*

The set of learnable parameters for RTUs is $\boldsymbol{\psi} \doteq \{\boldsymbol{\nu}^{\log}, \boldsymbol{\theta}^{\log}, \mathbf{W}_x^{c1}, \mathbf{W}_x^{c2}\}$. At each time step t, the learner receives a loss $\mathcal{L}_t(\hat{y}_t, y_t; \boldsymbol{\psi})$ where y_t is the network output at time t, then the gradient of the loss w.r.t the parameters is:

$$\frac{\partial \mathcal{L}_t}{\partial \boldsymbol{\psi}} = \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t^{c_1}} \frac{\partial \mathbf{h}_t^{c_1}}{\partial \boldsymbol{\psi}} + \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t^{c_2}} \frac{\partial \mathbf{h}_t^{c_2}}{\partial \boldsymbol{\psi}}, \tag{3.24}$$

where $\frac{\partial \mathbf{h}_t^{c1}}{\partial \boldsymbol{\psi}} = \left\{ \frac{\partial \mathbf{h}_t^{c1}}{\partial \boldsymbol{\nu}^{\log}}, \frac{\partial \mathbf{h}_t^{c1}}{\partial \boldsymbol{\theta}^{\log}}, \frac{\partial \mathbf{h}_t^{c1}}{\partial \boldsymbol{W}_x^{c1}}, \frac{\partial \mathbf{h}_t^{c1}}{\partial \boldsymbol{W}_x^{c1}} \right\}$ and $\frac{\partial \mathbf{h}_t^{c2}}{\partial \boldsymbol{\psi}} = \left\{ \frac{\partial \mathbf{h}_t^{c2}}{\partial \boldsymbol{\nu}^{\log}}, \frac{\partial \mathbf{h}_t^{c2}}{\partial \boldsymbol{\theta}^{\log}}, \frac{\partial \mathbf{h}_t^{c2}}{\partial \boldsymbol{W}_x^{c2}}, \frac{\partial \mathbf{h}_t^{c2}}{\partial \boldsymbol{W}_x^{c2}} \right\}$.

From equation 3.22, we can derive the following gradients:

$$\begin{aligned} \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t^{c_1}} &= \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_t^{c_1}} \\ \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t^{c_2}} &= \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_t^{c_2}} \end{aligned} \tag{3.25}$$

$$\begin{aligned} \frac{\partial \mathbf{h}_t^{c_1}}{\partial \boldsymbol{\nu}^{\log}} &= \frac{\partial \mathbf{g}(\boldsymbol{\nu},\boldsymbol{\theta})}{\partial \boldsymbol{\nu}^{\log}} \odot \mathbf{h}_{t-1}^{c_1} + \mathbf{g}(\boldsymbol{\nu},\boldsymbol{\theta}) \frac{\partial \mathbf{h}_{t-1}^{c_1}}{\partial \boldsymbol{\nu}^{\log}} - \frac{\partial \boldsymbol{\phi}(\boldsymbol{\nu},\boldsymbol{\theta})}{\partial \boldsymbol{\nu}^{\log}} \odot \mathbf{h}_{t-1}^{c_2} - \boldsymbol{\phi}(\boldsymbol{\nu},\boldsymbol{\theta}) \frac{\partial \mathbf{h}_{t-1}^{c_1}}{\partial \boldsymbol{\nu}^{\log}} \\ \frac{\partial \mathbf{h}_t^{c_2}}{\partial \boldsymbol{\nu}^{\log}} &= \frac{\partial \mathbf{g}(\boldsymbol{\nu},\boldsymbol{\theta})}{\partial \boldsymbol{\nu}^{\log}} \odot \mathbf{h}_{t-1}^{c_2} + \mathbf{g}(\boldsymbol{\nu},\boldsymbol{\theta}) \frac{\partial \mathbf{h}_{t-1}^{c_2}}{\partial \boldsymbol{\nu}^{\log}} + \frac{\partial \boldsymbol{\phi}(\boldsymbol{\nu},\boldsymbol{\theta})}{\partial \boldsymbol{\nu}^{\log}} \odot \mathbf{h}_{t-1}^{c_1} + \boldsymbol{\phi}(\boldsymbol{\nu},\boldsymbol{\theta}) \frac{\partial \mathbf{h}_{t-1}^{c_1}}{\partial \boldsymbol{\nu}^{\log}} \end{aligned} \tag{3.26}$$

$$\begin{aligned} \frac{\partial \mathbf{h}_t^{c_1}}{\partial \boldsymbol{\theta}^{\log}} &= \frac{\partial \mathbf{g}(\boldsymbol{\nu},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^{\log}} \odot \mathbf{h}_{t-1}^{c_1} + \mathbf{g}(\boldsymbol{\nu},\boldsymbol{\theta}) \frac{\partial \mathbf{h}_{t-1}^{c_1}}{\partial \boldsymbol{\theta}^{\log}} - \frac{\partial \boldsymbol{\phi}(\boldsymbol{\nu},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^{\log}} \odot \mathbf{h}_{t-1}^{c_2} - \boldsymbol{\phi}(\boldsymbol{\nu},\boldsymbol{\theta}) \frac{\partial \mathbf{h}_{t-1}^{c_1}}{\partial \boldsymbol{\theta}^{\log}} \\ \frac{\partial \mathbf{h}_t^{c_2}}{\partial \boldsymbol{\theta}^{\log}} &= \frac{\partial \mathbf{g}(\boldsymbol{\nu},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^{\log}} \odot \mathbf{h}_{t-1}^{c_2} + \mathbf{g}(\boldsymbol{\nu},\boldsymbol{\theta}) \frac{\partial \mathbf{h}_{t-1}^{c_2}}{\partial \boldsymbol{\theta}^{\log}} + \frac{\partial \boldsymbol{\phi}(\boldsymbol{\nu},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^{\log}} \odot \mathbf{h}_{t-1}^{c_1} + \boldsymbol{\phi}(\boldsymbol{\nu},\boldsymbol{\theta}) \frac{\partial \mathbf{h}_{t-1}^{c_1}}{\partial \boldsymbol{\theta}^{\log}} \end{aligned} \tag{3.27}$$

where

$$\begin{aligned} \frac{\partial \mathbf{g}(\boldsymbol{\nu},\boldsymbol{\theta})}{\partial \boldsymbol{\nu}^{\log}} &= \mathbf{g}(\boldsymbol{\nu},\boldsymbol{\theta}) \\ \frac{\partial \mathbf{g}(\boldsymbol{\nu},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^{\log}} &= -\boldsymbol{\phi}(\boldsymbol{\nu},\boldsymbol{\theta}) \exp(\boldsymbol{\theta}^{\log}) \\ \frac{\partial \boldsymbol{\phi}(\boldsymbol{\nu},\boldsymbol{\theta})}{\partial \boldsymbol{\nu}^{\log}} &= \boldsymbol{\phi}(\boldsymbol{\nu},\boldsymbol{\theta}) \\ \frac{\partial \boldsymbol{\phi}(\boldsymbol{\nu},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^{\log}} &= \mathbf{g}(\boldsymbol{\nu},\boldsymbol{\theta}) \exp(\boldsymbol{\theta}^{\log}). \end{aligned} \tag{3.28}$$

To efficiently compute the gradient w.r.t $\mathbf{W}_x^{c_1}$ and $\mathbf{W}_x^{c_2}$, we look at the influence of each when considering a single element from each recurrent state, $\mathbf{h}_t^{c_1}$ and $\mathbf{h}_t^{c_2}$:

$$
\begin{aligned}
h_{t,i}^{c_1} &= g(\nu_i,\theta_i)h_{t-1,i}^{c_1} - \phi(\nu_i,\theta_i)h_{t-1,i}^{c_2} + \gamma_i \sum_{j=0}^{d} w_{x,(i,j)}^{c_1} x_{t,j} \\
h_{t,i}^{c_2} &= g(\nu_i,\theta_i)h_{t-1,i}^{c_2} + \phi(\nu_i,\theta_i)h_{t-1,i}^{c_1} + \gamma_i \sum_{j=0}^{d} w_{x,(i,j)}^{c_2} x_{t,j}.
\end{aligned}
\tag{3.29}
$$

We then get:

$$
\begin{aligned}
\frac{\partial h_{t,i}^{c_1}}{\partial W_{x,(i,j)}^{c_1}} &= g(\nu_i,\theta_i)\frac{\partial h_{t-1,i}^{c_1}}{\partial W_{x,(i,j)}^{c_1}} - \phi(\nu_i,\theta_i)\frac{\partial h_{t-1,i}^{c_2}}{\partial W_{x,(i,j)}^{c_1}} + \gamma_i x_{t,j} \\
\frac{\partial h_{t,i}^{c_2}}{\partial W_{x,(i,j)}^{c_1}} &= g(\nu_i,\theta_i)\frac{\partial h_{t-1,i}^{c_2}}{\partial W_{x,(i,j)}^{c_1}} + \phi(\nu_i,\theta_i)\frac{\partial h_{t-1,i}^{c_1}}{\partial W_{x,(i,j)}^{c_1}} \\
\frac{\partial h_{t,i}^{c_1}}{\partial W_{x,(i,j)}^{c_2}} &= g(\nu_i,\theta_i)\frac{\partial h_{t-1,i}^{c_1}}{\partial W_{x,(i,j)}^{c_2}} - \phi(\nu_i,\theta_i)\frac{\partial h_{t-1,i}^{c_2}}{\partial W_{x,(i,j)}^{c_2}} \\
\frac{\partial h_{t,i}^{c_2}}{\partial W_{x,(i,j)}^{c_2}} &= g(\nu_i,\theta_i)\frac{\partial h_{t-1,i}^{c_2}}{\partial W_{x,(i,j)}^{c_2}} + \phi(\nu_i,\theta_i)\frac{\partial h_{t-1,i}^{c_1}}{\partial W_{x,(i,j)}^{c_2}} + \gamma_i x_{t,j}.
\end{aligned}
\tag{3.30}
$$

We see that each $h_{t,i}^{c_1}$ gets affected by weights from only one row of $\mathbf{W}_x^{c_1}$, thus, $\frac{\partial \mathbf{h}_t^{c_1}}{\mathbf{W}_x^{c_1}}$ can be written as a matrix of the same dimension as $\mathbf{W}_x^{c_1}$. The same is true for $\frac{\partial \mathbf{h}_t^{c_2}}{\mathbf{W}_x^{c_2}}, \frac{\partial \mathbf{h}_t^{c_1}}{\mathbf{W}_x^{c_2}}$, and $\frac{\partial \mathbf{h}_t^{c_2}}{\mathbf{W}_x^{c_1}}$.

3.4.1 Complexity Analysis of RT2

We now move to calculate the computation and memory complexity of RTUs when learning using the RTRL rules introduced in the previous section.

For an input $\mathbf{x_t} \in \mathbb{R}^d$ and hidden states $\mathbf{h}_t = [\mathbf{f}(\mathbf{h}_t^{c_1}); \mathbf{f}(\mathbf{h}_t^{c_2})] \in \mathbb{R}^{2n}$,we have $\mathbf{g}(\boldsymbol{\nu},\boldsymbol{\theta}), \boldsymbol{\phi}(\boldsymbol{\nu},\boldsymbol{\theta}), \boldsymbol{\gamma} \in \mathbb{R}^n$ and $\mathbf{W}_x^{c1}, \mathbf{W}_x^{c2} \in \mathbb{R}^{d\times n}$. An agent using the RT2 algorithm needs to store the gradient information,$\frac{\partial \mathbf{h}_{t-1}^{c_1}}{\partial \boldsymbol{\psi}}$ and $\frac{\partial \mathbf{h}_{t-1}^{c_2}}{\partial \boldsymbol{\psi}}$, from one step to the next. We denote the set of saved gradient information as:

$$
\begin{aligned}
\nabla_{\boldsymbol{\nu}^{t-1}} &\doteq \left\{ \frac{\partial \mathbf{h}_{t-1}^{c_1}}{\partial \boldsymbol{\nu}^{\log}}, \frac{\partial \mathbf{h}_{t-1}^{c_2}}{\partial \boldsymbol{\nu}^{\log}} \right\} \\
\nabla_{\boldsymbol{\theta}^{t-1}} &\doteq \left\{ \frac{\partial \mathbf{h}_{t-1}^{c_1}}{\partial \boldsymbol{\theta}^{\log}}, \frac{\partial \mathbf{h}_{t-1}^{c_2}}{\partial \boldsymbol{\theta}^{\log}} \right\} \\
\nabla_{\boldsymbol{W}_x^{t-1}} &\doteq \left\{ \frac{\partial \mathbf{h}_{t-1}^{c_1}}{\partial \mathbf{W}_x^{c_1}}, \frac{\partial \mathbf{h}_{t-1}^{c_2}}{\partial \boldsymbol{W}_x^{c_1}}, \frac{\partial \mathbf{h}_{t-1}^{c_1}}{\partial \mathbf{W}_x^{c_2}}, \frac{\partial \mathbf{h}_{t-1}^{c_2}}{\partial \boldsymbol{W}_x^{c_2}} \right\}.
\end{aligned}
\tag{3.31}
$$

The saved gradient information has the following dimensions:

$$\begin{aligned}
&\nabla_{\boldsymbol{\nu}^{t-1}} \in \mathbb{R}^{2n} \\
&\nabla_{\boldsymbol{\theta}^{t-1}} \in \mathbb{R}^{2n} \\
&\nabla_{\boldsymbol{W}_x^{t-1}} \in \mathbb{R}^{4(d\times n)}.
\end{aligned} \tag{3.32}$$

Then, it follows that memory complexity for RT2 is $\mathcal{O}(n + nd)$. i.e., linear in the number of parameters.

For the computational complexity, a forward pass according to 3.22 has a computational complexity of $\mathcal{O}(n + nd)$. Additionally, after doing the forward pass, the learner needs to update the saved gradient information according to equations 3.26 through 3.30 which has a computational complexity of $\mathcal{O}(n + nd)$. To summarize, using Real-Time Recurrent Learning with RTUs (RT2) has linear computational and memory complexities.

Chapter 4

Animal Learning

This chapter presents multiple experiments with RT2 and GRUs on online multi-step prediction problems inspired by animal learning. All the experiments are on *continuing problems*, i.e., the agent-environment interactions go on continually. At each time step, we evaluate the agents on the prediction error at that step. Then the agents update their learnable parameters to minimize the prediction error.

All agents have one recurrent layer, either an RTU or a GRU, and one linear layer. At each time step t, the agent passes the observation $\mathbf{o}_t$ to the recurrent layer, which outputs the recurrent state, the agent state. The recurrent state is then passed to the linear layer generating the prediction.

For each agent, we swept over the learning rate α used to update the network parameters, $\alpha \in \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}\}$, and averaged the performance for each learning rate over 5 independent runs. We then selected the best-performing learning rate for each agent and ran 30 independent runs using it. For all the experiments, we ran the agents for 2 million steps, and the performance was the mean squared prediction error averaged over the 2 million steps.

4.1 Animal Learning Benchmark

Trace conditioning is the type of experiment where animals predict when a specific stimulus will occur, typically food, based on the occurrence of another stimulus, usually a tone. There is no prior connection between the two stimuli. However, after enough repetitions of pairing them together, i.e., playing the tone and then serving the food, the animal learns to predict food arrival when it hears the tone (Pavlov, 1927).

Rafiee et al.,(2020) introduced a multi-step prediction benchmark inspired by experiments in animal learning. The first benchmarking task simulates the trace conditioning experiments; two signals appear in a sequence, the Conditional Stimulus (CS) followed by the Unconditional Stimulus (US). The CS is the trigger signal, similar to the tone, and the US is the signal of interest which appears several time steps after the CS, similar to the food. Figure 4.1 shows an example of the CS and the US in a sequence, the interval between the occurrence of the CS and the US is called the Inter-Stimulus Interval (ISI). The longer the ISI, the harder it is for the agent to predict when the US will occur. The interval between two CS signals is called Inter-Trial Interval (ITI) since a CS signal marks the beginning of a new conditioning trial.

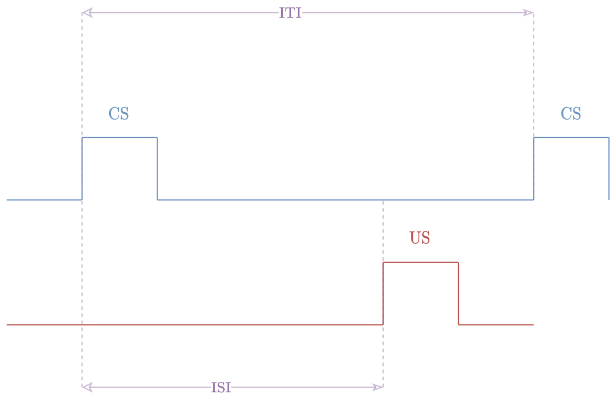


Figure 4.1: A trial in the trace conditioning task.

The trace conditioning task can be made harder by introducing distractor signals: signals that are not correlated to either the CS or the US. When introducing distractor signals, the agent needs to filter out these signals and focus only on the CS signals, then it needs to remember when the CS happened to predict the US. Figure 4.2 shows the observations when distractor signals are part of the observation stream.

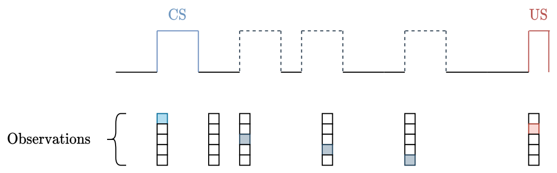


Figure 4.2: Examples of observations in trace conditioning.

We designed the experiments to match the hardest setting for the trace conditioning task as introduced by Rafiee et al.,(2020). In all the experiments, we specified the activation length of the CS signals to 4 and the US to 2 time steps. We also added 10 distractors, each distractor has a different activation probability modeled as a Poisson distribution, and all the distractors have an activation length of 4 time steps. At the beginning of each trial, we select ITI uniformly from $[80, 120]$ and ISI uniformly from $[20, 40]$.

The agent's objective is to predict when the US signal will occur which we model as a prediction of the discounted sum of the future US signal, the return G_t:

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k US_{t+k+1}, \tag{4.1}$$

where γ is a discount factor determining the prediction horizon, and set equal to $1 - 1/\mathbb{E}[\text{ISI}]$.

4.2 Experimental Analysis

In this section, we empirically examine the properties of RT2, and how it compares to other existing methods, specifically GRUs with T-BPTT. We chose GRU as our baseline here since it was shown to have superior performance in partially observable RL problems (Morad et al., 2023). We did an ablation study to investigate the performance of RT2 and GRUs along several axes of comparison.

4.2.1 Learning Under Computational Constraints

The first experiment attempts to answer the question: *how well do different agents exploit the available computation?* The hypothesis behind this question is: under limited computational resources, agents that depend on T-BPTT to update their parameters have a trade-off between the truncation length T and the number of parameters in their recurrent architectures since the computational complexity of T-BPTT is $\mathcal{O}(Tn^2)$, where n is hidden state dimension in the recurrent architecture. However, agents using RT2 do not have this trade-off and can use all the available computation to have more parameters and more features in their recurrent state, leading to better performance.

To verify our first hypothesis, we specified the same computational budget for the two agents to be around 15000 FLOPs. The first agent used RT2 and we selected the number of parameters to fit the computational budget. The second agent used GRUs, and we tested several configurations of T and n such that, the overall computations fit the computational constraints. Figure 4.3 shows the results of this experiment. As we move along the horizontal axis, the number of parameters for GRU decreases as T increases to fit the computational constraints. However, the RT2 agent does not depend on T, so it only has one configuration. Additionally, we show an example of the predictions generated by the two agents over the last 600 timesteps in figure 4.4. The RT2 agent exploited all the available computations to have more features and achieved a lower error than the GRU agent. [1]

[1] It is quite hard to give the exact same computation to the two agents. When this is the case, we err on the side of giving GRU more computation than RT2.

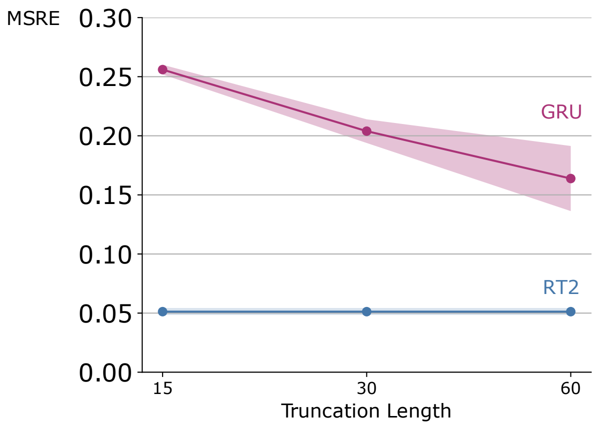


Figure 4.3: Learning under computational constraints. For RT2, the network has 500 hidden units. For GRU, we start with 13 hidden units for $T = 15$, then decrease it to 8 units when $T = 30$, and finally decrease it to 5 units when $T = 60$. These numbers were selected to satisfy the computational constraints. In this task, $\mathbb{E}[\text{ISI}] = 30$ which means that GRU agent with $T \geq 30$ has a sufficiently large history to learn the prediction.

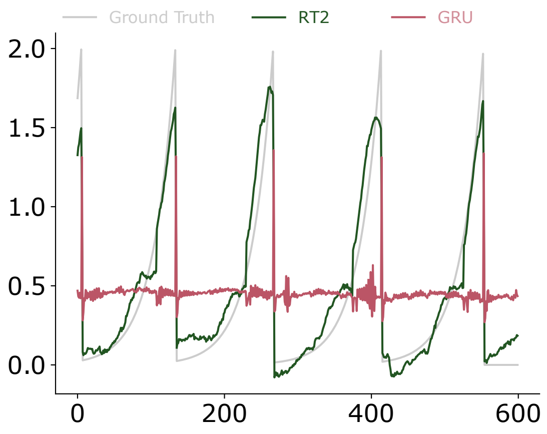
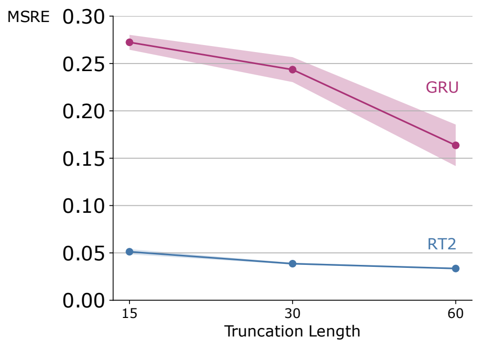


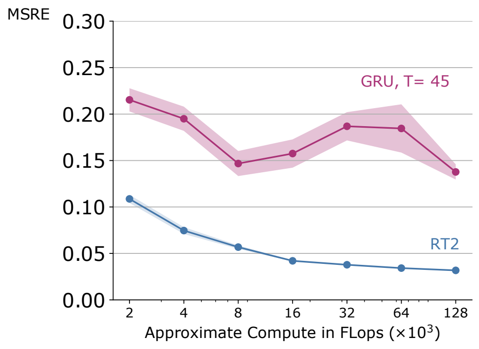
Figure 4.4: Example of predictions generated by RT2 and GRU agents from the last 600 timesteps of experiment 1. For GRU, these predictions were generated by the agent with $T = 60$.

4.2.2 Scaling With Computation

A natural follow-up question to our first hypothesis is: *what if more computational resources are available?* Agents using T-BPTT have two choices when more computational resources are available. The first choice is to use the additional resources to increase the truncation length, and the second is to use the additional resources to have more features in the recurrent state and more parameters. On the other hand, agents using RT2 have one option: use additional resources to make the network bigger and have more features.



(a) Scaling with computation (Part 1). For RT2, the network starts with 550 hidden units, then 1110 units, and finally 2220 units. For GRU, the network has 13 hidden units. At each point of comparison, both GRU and RT2 have the same computational budget.



(b) Scaling with computation (Part 2). For RT2, the network has hidden units $\in \{80, 160, 300, 600, 1150, 2300, 4600\}$. For GRU, the network has hidden units $\in \{2, 3, 5, 8, 13, 20, 30\}$.

Figure 4.5: Scaling with computation

We designed our second experiment to study T-BPTT with increasing T and a fixed number of parameters. In this experiment, the computation increases for GRU as T increases. We compensated for the computational increase by adding more features to the RT2 agent such that, each two corresponding points from GRU and RT2 have the same overall computation. Figure 4.5a shows the results of this experiment. Increasing the truncation length did help improve the GRU agent. However, the RT2 agent is still achieving a lower error.

Our third experiment studies the case where a T-BPTT agent uses the computation to increase the number of features. We fixed the truncation length to 45 which is greater than the ISI interval. With a 45 truncation length, the GRU agent has the whole context needed to make a prediction. Then, we increased the computational resources of both GRU and RT2 agents by increasing the number of learnable parameters. We used the estimated number of FLOPs for T-BPTT as $2n^2 + 2nd + 2n$ and for RT2 as $n + nd$.[2] Figure 4.5b shows the results of this experiment. As we increase the computation available, the RT2 agent's performance consistently improves. However, the performance improvement for the GRU agent is inconsistent which also aligns with our hypothesis on the trade-off between the truncation length and the number of parameters: using most of the computational resources for more parameters in the GRU agent does not guarantee improved performance.

4.2.3 Scaling With Parameters

In the final experiment, we study the performance of RT2 and GRU when given the same number of parameters and allow the GRU agent to use more computation. We fixed the truncation length to 45 as before and used the same number of parameters for both agents. Figure 4.6 shows the results of this experiment. For RT2, we see the same consistent performance improvement as we increase the number of parameters. For GRU, the performance improvement is also quite consistent though degrades slightly towards the end. In this experiment, the RT2 agent outperforms the GRU agent even though the GRU uses more computation.

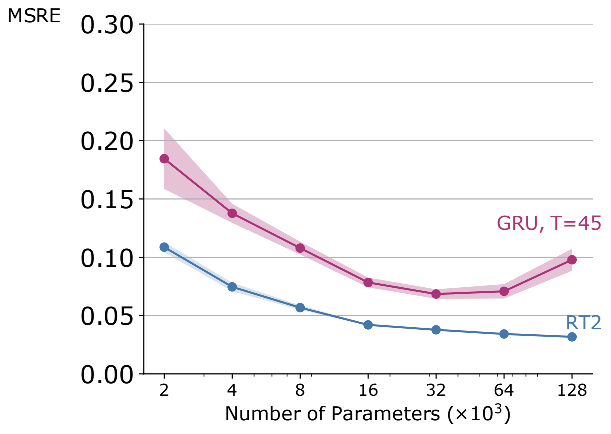


Figure 4.6: Scaling by increasing the number of parameters. For RT2, the number of hidden units is $\in \{80, 160, 300, 600, 1150, 2300, 4600\}$. For GRU, the number of hidden units is $\in \{20, 30, 48, 67, 97, 140, 200\}$.

All of the above experiments suggest that using real-time recurrent learning combined with a scalable architecture, RT2, achieves a noticeable gain in performance on online continuing tasks. Additionally, all RT2 agents have a noticeably lower variance than GRU agents. However, we leave

[2]This just an estimation of the FLOPs, and while it probably differs from the actual number of FLOPs, it gives us an indication on the effect of increasing the compute.

examining the variance properties for future work.

4.2.4 Comparison to Diagonal RNNs

To complete the ablation study in this chapter, we performed an additional experiment where we compared the performance of RT2 to two diagonal RNNs. The first is LRUs as introduced in Orvieto et al., 2023, which has complex-valued diagonal elements. The second is a vanilla block diagonal RNN, which has a recurrent formulation similar to RTU but ignores the relation between the learnable parameters. i.e., replaces $\mathbf{c}_k$ in 3.18 with

$$\mathbf{c}_k = \begin{bmatrix} a_k & b_k \\ c_k & d_k \end{bmatrix}. \tag{4.2}$$

This simple modification means that instead of only learning ν and θ, we are learning 4 independent parameters. However, the memory and computational complexities are the same as RT2. Since LRUs require the recurrent dynamics to be linear, we slightly modified the network used in this experiment by adding a feedforward layer with a non-linearity before the recurrent layer. Hence, allowing the LRU agent to have a non-linear function approximator.

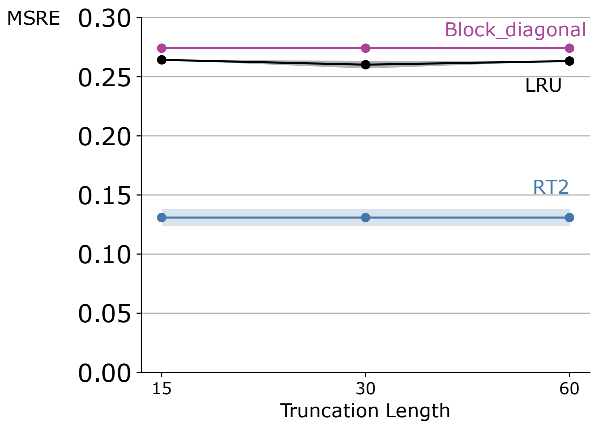


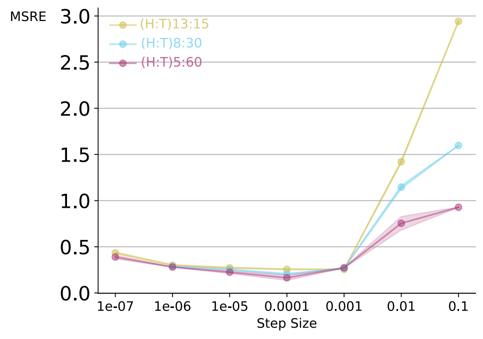
Figure 4.7: Comparison to diagonal RNNs. For RT2 and block diagonal RNN, the network has 330 hidden units. For LRU, the network starts with 22 hidden units for truncation length $T = 15$, then decreases to 9 units when $T = 30$, and finally decreases to 8 units when $T = 60$. Besides the recurrent layer, all agents have an additional feedforward layer with 10 hidden units. These results are averaged over 5 independent runs.

Figure 4.7 shows the result of the last experiment. We notice that RT2 achieved a better performance than both LRU and the block diagonal RNN. We can attribute the performance improvement over LRU to both the non-linear recurrent dynamics of RTU and using the full gradient information as discussed in 3.3. The main difference between RTUs and the block diagonal RNN is that RTU leverages the relation between the learnable parameters as a result of using the rotational representation of complex numbers. The results suggest that leveraging this relation might be helpful for efficient learning.

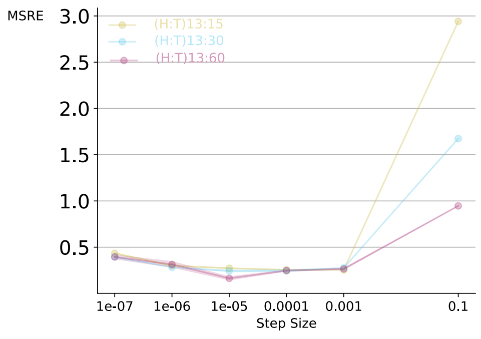
4.2.5 Learning Rate Sensitivity

Finally, we show the learning rate sensitivity plots for all GRU agents used in the above experiments in figure 4.8, the learning rate sensitivity plots for all RT2 agents from experiments 1 to 4 in figure 4.9, and the sensitivity plots for the LRU, the RT2, and the block diagonal agents in the last experiment in 4.10.

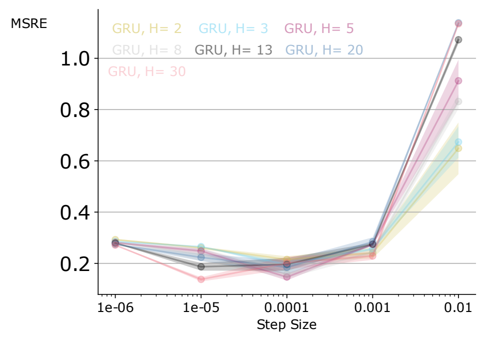
For RT2 and the block diagonal RNN, we observed that high learning rate values usually lead to divergence. This observation is consistent with the convergence conditions for RTRL (Williams and Zipser, 1989) and presents a future question on how to mitigate this issue.



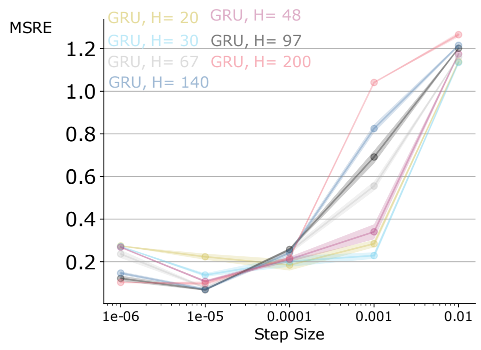
(a) GRUs used in learning under computational constraints experiment. The *(H: T)* in the label refers to the (hidden dimension: truncation length) for the GRU.



(b) GRUs used in scaling with computation experiment (Part 1). The *(H: T)* in the label refers to the (hidden dimension: truncation length) for the GRU.

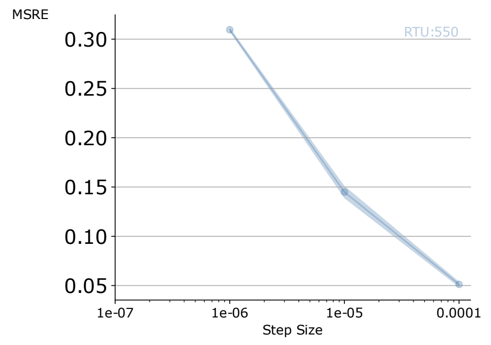


(c) GRUs used in scaling with computation experiment (Part 2) with truncation length $T = 45$. The H in the label refers to the hidden dimension.

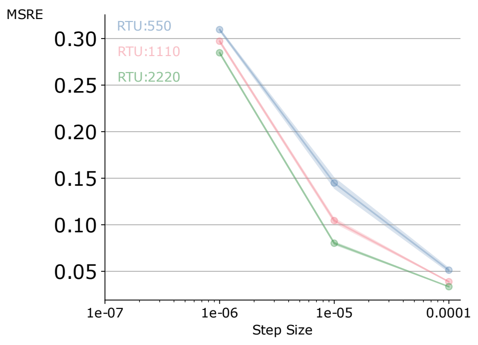


(d) GRUs used in scaling with parameters experiment with truncation length $T = 45$. The H in the label refers to the hidden dimension.

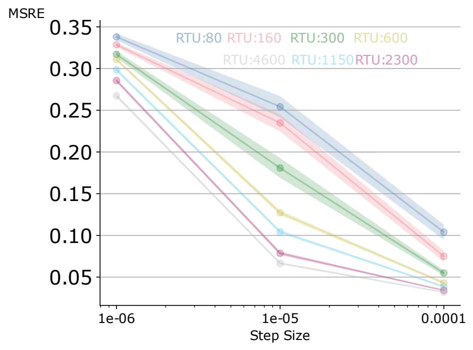
Figure 4.8: Learning rate sensitivity curves for GRU architectures used in the experiments. All solid lines are the average of 5 independent runs, and the shaded regions are the standard error.



(a) RTU architectures used in learning under computational constraints experiment.



(b) RTU architectures used in scaling with computation (Part 1) experiment.



(c) RTU architectures used in scaling with computation (Part 2) and scaling with parameters experiments.

Figure 4.9: Learning rate sensitivity curves for RTU architectures used in the experiments. Each curve corresponds to a different number of hidden units indicated in the corresponding label. All solid lines are the average of 5 independent runs, and the shaded regions are the standard error.

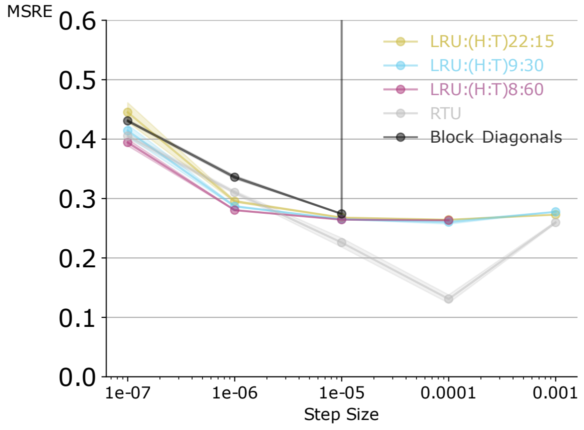


Figure 4.10: Learning rate sensitivity for both the RT2 and the LRU agents for the last experiment. The (H: T) in the labels refers to the number of hidden units and the truncation length. All solid lines are the average of 5 independent runs, and the shaded regions are the standard error.

Chapter 5

Partial Observability in Classical Control

In this chapter, we extend the experiments to the control setting, where the agent can take actions and control aspects of the environment to achieve a specific goal. We briefly describe policy gradient methods where the agent learns a parameterized policy and uses it to select actions. We then introduce the two classical control environments that we use in our experiments and the modifications we made to them to introduce partial observability. Finally, we look at the performance of two agents learning in those two environments, an RT2 agent and a GRU agent.

5.1 Policy Gradient Methods

As the agent interacts with the environment, it learns how to behave in each state to maximize the cumulative reward signal. This learned behavior is the agent's policy. The policy is a stochastic map from the states to actions $\pi : \mathcal{S} \rightarrow \mathcal{A}$, and it tells the agent the probability of taking each action in the current state. Formally, the agent learns a parametrized policy, $\pi(a|s, \mathbf{W}_p) = \Pr\{A_t = a|S_t = s, \mathbf{W}_{p_t = \mathbf{W}_p}\}$, that outputs the probability of taking action a at time t, given that the current state of the environment is s, and the current policy parameters are $\mathbf{W}_p$(Sutton and Barto, 2018).

The agent learns the policy parameters by maximizing an objective function $J(\mathbf{W}_p)$. For episodic problems, i.e., where the agent-environmental interactions are divided into sequences of episodes, the objective function is defined as the value of the start state of the episode(Sutton and Barto, 2018):

$$J(\mathbf{W}_p) \doteq v_{\pi_{\mathbf{W}_p}}(s_0).$$

The agent can then estimate the gradient of the objective function w.r.t the policy parameters and then update the parameters in the direction that maximizes this objective function. Methods that use this approach are called *policy gradient methods*. In some policy gradient methods, the agent also learns a parametrized state-value function $\hat{v}(s, \mathbf{W}_v)$, which tells the agent the expected return starting from state s and following the policy π. In such cases, we refer to the component learning the state-value function as the critic, and the component learning the policy as the actor. These policy gradient methods are called *actor-critic methods*.

Proximal Policy Optimization (PPO) (Schulman et al., 2017) is an online policy gradient method, i.e., the agent learns the policy parameters online while interacting with the environment. In PPO, the agent starts by collecting a trajectory of observations, actions, and rewards. From this trajectory, we can calculate the truncated λ-return (Sutton and Barto, 2018):

$$G^{\lambda}_{t:k} \doteq (1-\lambda) \sum_{n=1}^{k-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{k-t-1} G_{t:k}. \tag{5.1}$$

Roughly speaking, the λ-return is a weighted sum of the n-step returns, $G_{t:t+n}$, and λ is the parameter controlling this weighted sum. We can write the λ-return more efficiently to calculate it online from our estimated value function(Sutton and Barto, 2018):

$$G^{\lambda}_{t:k} \doteq \hat{v}(\mathbf{s}_t, \mathbf{W}_{v,t-1}) + \sum_{i=t}^{t+k-1} (\gamma\lambda)^{i-t} R_{t+1} + \gamma \hat{v}(\mathbf{s}_{i+1}, \mathbf{W}_{v,t}) - \hat{v}(\mathbf{s}_i, \mathbf{W}_{v,i-1}).$$

The PPO algorithm uses a variation of the λ-return, Generalized Advantage Estimate (GAE), which subtracts the value function from the λ-return to reduce the variance of the return estimate(Schulman et al., 2015):

$$\hat{A}^{(\gamma,\lambda)}_t \doteq \sum_{i=t}^{t+k-1} (\gamma\lambda)^{i-t} R_{t+1} + \gamma \hat{v}(\mathbf{s}_{i+1}, \mathbf{W}_{v,t}) - \hat{v}(\mathbf{s}_i, \mathbf{W}_{v,i-1}). \tag{5.2}$$

Using the GAE, the objective function of PPO is written as:

$$L_t(\mathbf{W}_p, \mathbf{W}_v) \doteq \mathbb{E}_t \left[L_{\text{actor},t} - c_1 L_{\text{critic},t} + c_2 S(\pi(\mathbf{s}_t, \mathbf{W}_p)) \right], \tag{5.3}$$

where $L_{\text{actor},t}$ is the policy loss and defined as:

$$L_{\text{actor},t}(\mathbf{W}_p, \mathbf{W}_v) \doteq \mathbb{E}_t \left[\min(r_t(\mathbf{W}_p)\hat{A}_t, \text{clip}_\epsilon(r_t(\mathbf{W}_p))\hat{A}_t) \right], \tag{5.4}$$

$$r_t(\mathbf{W}_p) = \frac{\pi(a_t|s_t, \mathbf{W}_{p_{\text{new}}})}{\pi(a_t|s_t, \mathbf{W}_{p_{\text{old}}})},$$

$$\text{clip}_\epsilon(r_t(\mathbf{W}_p)) = \text{clip}(r_t(\mathbf{W}_p), 1-\epsilon, 1+\epsilon)$$

$L_{\text{critic},t}$ is the value loss and defined as:

$$L_{\text{value},t}(\mathbf{W}_v) \doteq \max((\hat{v}(\mathbf{s}_t, \mathbf{W}_{v,t}) - G^{\lambda}_{t:k})^2, (\text{clip}_{\epsilon}(\hat{v}) - G^{\lambda}_{t:k})^2), \tag{5.5}$$

$$\text{clip}_{\epsilon}(\hat{v}) = \text{clip}(\hat{v}(\mathbf{s}_t, \mathbf{W}_{v,t}), 1-\epsilon, 1+\epsilon),$$

and $S(\pi(\mathbf{s}_t, \mathbf{W}_p))$ is the entropy of the policy. The coefficients c_1 and c_2 are hyperparameters that control the relative importance of the value loss and the entropy term in the objective function (Schulman et al., 2017). Finally, the agent can update the policy and the value function parameters by using the gradient information of the loss function in 5.3 w.r.t the parameters.

There are many implementation details to the PPO algorithm usually employed to improve its performance (Andrychowicz et al., 2021, Engstrom et al., 2020, and Henderson et al., 2018). Here, we opted for a minimal implementation of the PPO algorithm shown in Algorithm 4.

Algorithm 4 Online Control with minimal PPO

```
Inputs: a differentiable policy parametrization $\pi(a|s, \mathbf{W}_p)$.
Inputs: a differentiable state-value function parametrization $\hat{v}(s, \mathbf{W}_v)$.
Algorithm parameters: learning rate $\alpha$, rollout length $k$, number of epochs $n$, value coeffient $c_1$,
entropy coefiient $c_2$, and clip coeffient $\epsilon$.
loop
    Generate a trajectory $\mathbf{O}_0, A_0, R_1, \ldots, \mathbf{O}_k, A_k, R_k$
    Calculate GAE according to 5.2
    for epoch = 1, ..., n do
        Re-run network to update hidden states for the trajectory.
        Calculate the value loss according to 5.5.
        Calculate the policy loss according to 5.4.
        Calculate the entropy of the policy.
        Optimize the learnable parameters for both the policy and the value function
        using the gradient of the loss function in 5.3.
```

5.2 Classical Control Environment

We selected two classical control environments for our experiments: CartPole (Barto, Sutton, and Anderson, 1983), and Acrobot (Sutton, 1995). The two environments are episodic which means that the agent-environment interactions break into sequences of episodes. Each episode starts with a state sampled from a uniform distribution of possible starting states and ends with a special state, the terminal state. Both environments have continuous state space and discrete action spaces. We show an illustration for both CartPole and Acrobot in Figure 5.1.

In CartPole, the agent tries to balance a pole attached to a cart through an un-actuated joint.

The agent can apply force to the cart to move it in either the left or the right direction on a frictionless track. In the fully observable version of CartPole, the observation consists of the cart's position, the cart's velocity, the pole angle, and the pole's angular velocity. Each episode of CartPole starts with all observations sampled from a uniform distribution of values in the range $[-0.05, 0.05]$, in other words, the pole is balanced and the cart is in the middle of the track. The episode terminates when the pole angle or the cart position is out of an allowed range, $\pm 12°$ for the angle and ± 2.4 for the position, and we truncate the episode if the agent was able to balance the pole for 500 steps. The agent receives a reward of $+1$ every time step until the episode terminates or is truncated. Hence, the maximum sum of rewards the agent can achieve in CartPole is 500 (Towers et al., 2023).

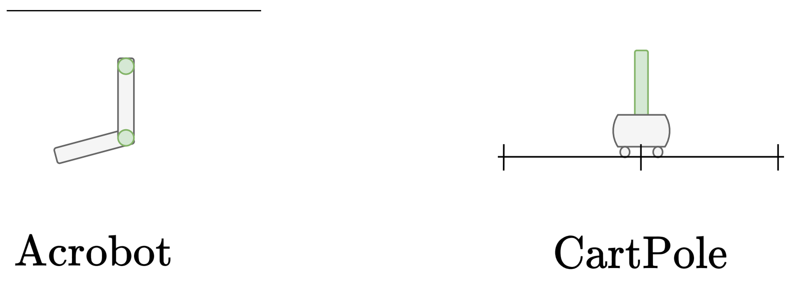


Figure 5.1: The CartPole and the Acrobot environments used in our experiments.

In Acrobot, the agent applies torque to a joint of a double pendulum system and the goal is to raise the free end of the pendulum above a specific height. The double pendulum has two joints and two links. The first joint is fixed and the agent actuates the second joint. In the fully observable version of Acrobot, the observation consists of $\cos(\theta_1)$, $\sin(\theta_1)$, $\cos(\theta_2)$, $\sin(\theta_2)$, angular velocity of θ_1, and angular velocity of θ_2. θ_1 is first joint's angle, $\theta_1 = 0$ denotes a downward position, and θ_2 is the second joint's angle measured relative to θ_1. Each episode of Acrobot starts with the pendulum pointing downward and terminates when the free end of the pendulum reaches the target position defined by $-\cos(\theta_1) - \cos(\theta_1 + \theta_2) > 0.1$ or when the number of interaction steps exceeds 500 steps. Since the goal is to reach a specific position, the agent receives a -1 reward for each step until it reaches the target position where it receives a 0 reward and the episode terminates (Towers et al., 2023).

CartPole and Acrobot are fully observable, meaning that the observation vectors contain all the information needed to learn an optimal policy. We start our experiments by getting baseline results for PPO in the fully observable setting, before introducing partial observability. We train an agent with two feedforward layers for the actor network, two feedforward layers for the critic network, and one shared layer for learning representation before feeding it to both the actor and the critic networks. The agent is using PPO to learn the policy and the state-value function. We show the performance of this agent in Figure 5.2. As expected, the agent was able to learn an optimal policy.

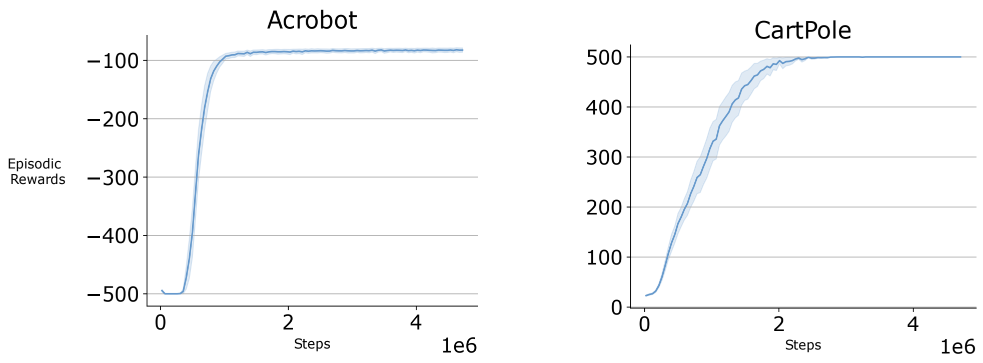


Figure 5.2: Experimental results on the fully observable version of CartPole and Acrobot. Since the environments here were fully observable, we used feedforward networks for both the actor and the critic networks.

5.3 Partial Observability in Classical Control

To introduce partial observability, we modified the observation vector of the two environments to contain only positional information. We started from the environments implementations by Lange, 2022 and created wrappers for the observations to mask out all the velocity information. The idea of masking out the velocity information to introduce partial observability was previously explored in the literature and was shown to give good diagnostic experiments (Morad et al., 2022, Duan et al., 2016). Table 5.1 shows the difference between the full and the partial observation vectors for both CartPole and Acrobot. We also created an additional variation of the environments where we added noise to the observation vector to increase the difficulty. The noise was randomly sampled from a normal distribution with zero mean and 0.1 standard deviation.

Environment	Full observation	Partial Observation
CartPole	Cart Position. Cart Velocity. Pole Angle. Pole Angular Velocity.	Cart Position. Pole Angle.
Acrobot	$\cos(\theta_1)$. $\sin(\theta_1)$. $\cos(\theta_2)$. $\sin(\theta_2)$. θ_1 Angular velocity. θ_2 Angular velocity.	$\cos(\theta_1)$. $\sin(\theta_1)$. $\cos(\theta_2)$. $\sin(\theta_2)$.

Table 5.1: Differences between the fully observable and the partially observable settings.

We trained the previous agent on these new environments to confirm that our new environments are partially observable. Figure 5.3 shows the agent's performance when trained in the new partially observable environments. We can see that the agent can no longer solve these problems when the observation vector only contains positional information. These results confirm that these new environments are partially observable and that we need to add a recurrent layer to the agent.

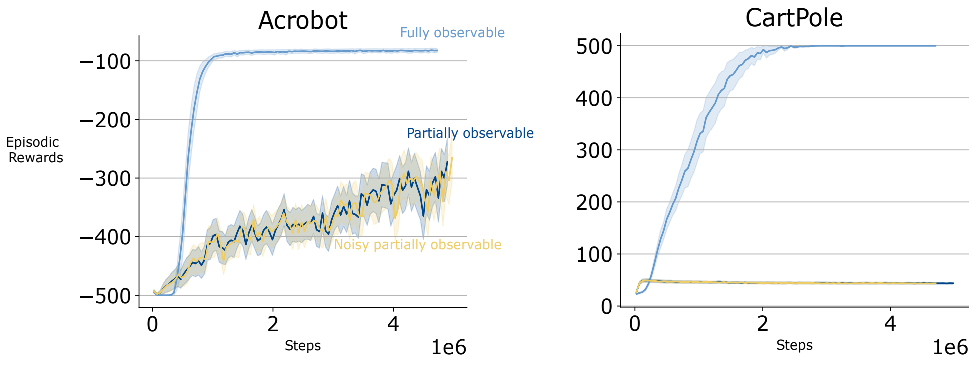
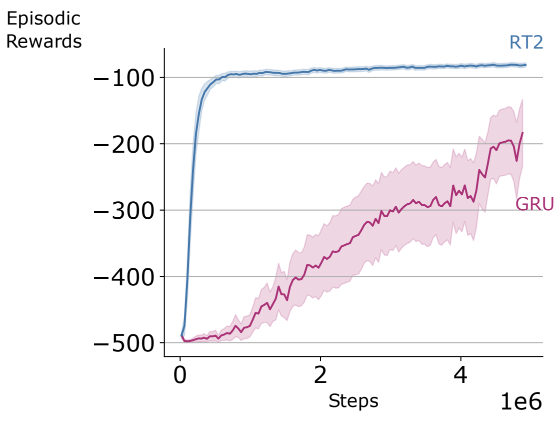


Figure 5.3: Experimental results on the partially observable version of CartPole and Acrobot. We can see that a feedforward architecture is unable to solve these problems.

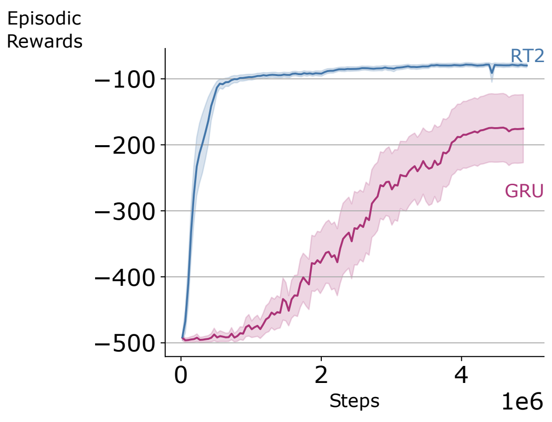
In our final experiment, we compare two recurrent agents, the RT2 agent, and the GRU agent, on the partially observable control tasks. Both agents have a recurrent representation layer that processes the observation and feeds it to the actor and the critic networks. The actor and the critic networks have two feedforward layers, each with 64 hidden units. For the GRU agent, the recurrent layer has 64 recurrent units. For the RT2 agent, the recurrent layer has 110 recurrent units. These configurations allow both agents to have the same number of learnable parameters. We did not constrain the computational budget in this experiment, which means that since GRU is using T-BPTT, it will be using more computational resources to train the same number of parameters.

Figure 5.4 shows the results on the partially observable Acrobot and CartPole environments. In the partially observable Acrobot, the RT2 agent performed similarly to our first fully observable experiment in 5.2, which means that the RT2 agent could overcome the partial observability completely. The GRU agent was also able to reach a good policy compared to the feedforward baseline 5.3. However, the performance was lower than that of RT2. In the partially observable CartPole, both RT2 and GRU agents achieved similar performance that was also better than the feedforward baseline in 5.3. However, both agents had lower performance than the fully observable case 5.2, which suggests that this environment might still be too complex for the agents and needs further investigation.

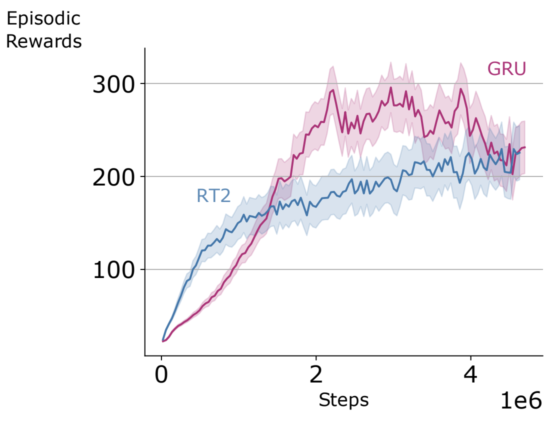
Finally, we show the learning rate sensitivity curves for both agents in Figure 5.5, and the rest of the PPO hyper-parameters used in all the experiments in Table 5.2.



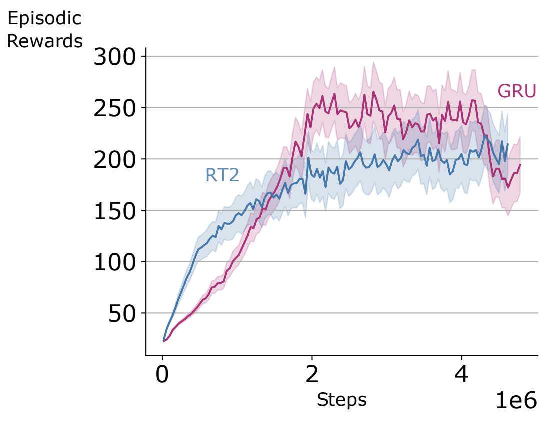
(a) Partially Observable Acrobot (Without Noise).



(b) Partially Observable Acrobot (With Noise).



(c) Partially Observable CartPole (Without Noise)



(d) Partially Observable CartPole (With Noise).

Figure 5.4: Results from the partially observable control experiments. For Acrobot, the solid lines correspond to the mean of 20 runs. For CartPole, the solid lines correspond to the mean over 40 runs. The shaded areas are the standard error.

Hyper-Parameter	Value
Rollout length k	256
Discount factor γ	0.99
λ	0.9
ϵ	0.2
c_1	1
c_2	0

Table 5.2: PPO hyper-parameters used in all the experiments.

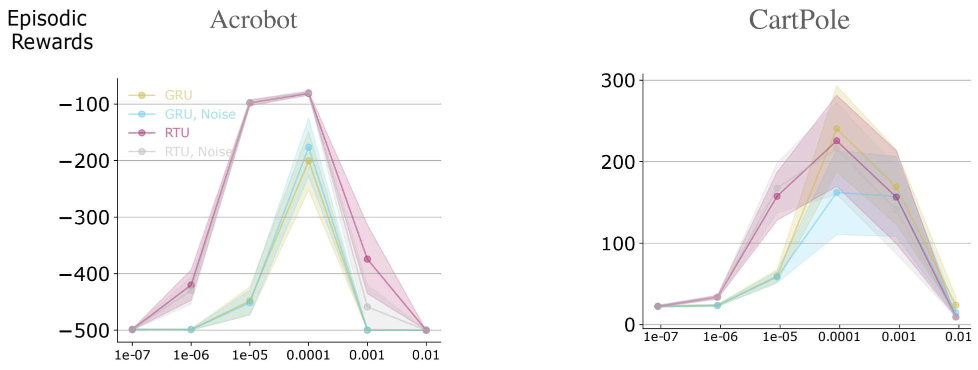


Figure 5.5: Learning rate sensitivity curves. The solid points correspond to the mean rewards from the last 600k steps, averaged over 10 runs and the shaded areas are the standard error. The curves used in the noisy environments are denoted with *Noise* in their labels.

Chapter 6

Conclusion and Future Directions

In this thesis, we introduced the RT2 algorithm, a combination of a new recurrent architecture, RTUs, and Real-Time Recurrent Learning, RTRL. RT2 was motivated by the idea of finding a recurrent architecture for which RTRL updates are tractable and ideally have a linear complexity without restricting the representability of the recurrent architecture. We based our new recurrent architecture on a simple idea: Instead of learning a whole matrix, we can learn the eigenvalues corresponding to this matrix. We then addressed the technical issues that arise when learning eigenvalues, including how to represent complex eigenvalues and generate real-valued outputs from complex representations. Resolving these questions led us to the formalization of RTUs. Finally, we derived RTRL update rules for RTUs and showed that it has linear computational and memory complexities.

We studied the performance of RT2-based agents through a set of empirical experiments in both RL prediction and control domains. We conducted our first set of experiments on a benchmark inspired by animal learning. In this benchmark, the agent needs to predict a stimulus occurrence based on another stimulus. We asked several empirical questions in these experiments: 1) How do different agents learn under limited computational resources? 2) How does the performance of different agents change when given more computational resources? 3) How does the performance of different agents change as a function of the number of learnable parameters when the computational resources are not restricted? While answering these questions, we observed that RT2-based agents always outperformed GRU-based agents even when GRU-based agents were using more computational resources than RT2.

In the second set of experiments, we modified two classical control environments by masking out all the velocity information from their observation vector. These modifications introduce a different source of partial observability; instead of remembering the occurrence of a stimulus as in animal learning, the agent here needs to accumulate information from several timesteps. We showed that the RT2-based agent outperformed the GRU-based agent in one environment, and both agents performed the same in the other environment.

While this thesis presented a proof of concept for RT2, several future directions remain to explore. Firstly, from the analysis in 3.2, the vanishing/exploding gradient problem is only partially solved in RT2, and additional analysis is required to prevent it completely. Second, combining RTRL-based updates with RL algorithms that use replay buffers is still an under-explored area of research (Irie, Gopalakrishnan, and Schmidhuber, 2023). Finally, it is clear that linear complex-valued diagonal RNNs learn the eigenvalues of the dense RNN, but it is unclear whether or not this holds for non-linear complex-valued diagonals, such as RTUs.

References

Andrychowicz, Marcin et al. (2021). “What Matters for On-Policy Deep Actor-Critic Methods? A Large-Scale Study.” In: *International Conference on Learning Representations*.

Barto, Andrew G., Sutton, and Charles W. Anderson (1983). “Neuronlike adaptive elements that can solve difficult learning control problems.” In: *IEEE Transactions on Systems, Man, and Cybernetics*.

Cho, Kyunghyun et al. (2014). “On the properties of neural machine translation: Encoder-decoder approaches.” In: *arXiv preprint arXiv:1409.1259*.

Cooijmans, Tim and James Martens (2019). “On the variance of unbiased online recurrent optimization.” In: *arXiv preprint arXiv:1902.02405*.

Dai, Zihang et al. (2019). “Transformer-XL: Attentive Language Models beyond a Fixed-Length Context.” In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*.

Duan, Yan et al. (2016). “Benchmarking deep reinforcement learning for continuous control.” In: *International conference on machine learning*.

El-Naggar, Nadine, Pranava Madhyastha, and Tillman Weyde (2023). “Theoretical Conditions and Empirical Failure of Bracket Counting on Long Sequences with Linear Recurrent Networks.” In: *EACL 2023*.

Engstrom, Logan et al. (2020). “Implementation Matters in Deep Policy Gradients: A Case Study on PPO and TRPO.” In: *International Conference on Learning Representations*.

Espeholt, Lasse et al. (2018). “Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures.” In: *International conference on machine learning*.

Gruslys, Audrunas et al. (2018). “The Reactor: A fast and sample-efficient Actor-Critic agent for Reinforcement Learning.” In: *International Conference on Learning Representations*.

Hausknecht, Matthew and Peter Stone (2015). “Deep recurrent q-learning for partially observable mdps.” In: *2015 aaai fall symposium series*.

Henderson, Peter et al. (2018). “Deep reinforcement learning that matters.” In: *Proceedings of the AAAI conference on artificial intelligence*.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory." In: *Neural computation.*

Huang, Feiqing et al. (2023). "Encoding Recurrence into Transformers." In: *The Eleventh International Conference on Learning Representations.*

Irie, Kazuki, Anand Gopalakrishnan, and Jürgen Schmidhuber (2023). *Exploring the Promise and Limits of Real-Time Recurrent Learning.* arXiv: `2305.19044 [cs.LG]`.

Javed, Khurram et al. (2023). "Online Real-Time Recurrent Learning Using Sparse Connections and Selective Learning." In: *arXiv preprint arXiv:2302.05326.*

Kapturowski, Steven et al. (2019). "Recurrent Experience Replay in Distributed Reinforcement Learning." In: *International Conference on Learning Representations.*

Lange, Robert Tjarko (2022). *gymnax: A JAX-based Reinforcement Learning Environment Library.* Version 0.0.4.

Li, Xiujun et al. (2015). "Recurrent reinforcement learning: a hybrid approach." In: *arXiv preprint arXiv:1509.03044.*

Menick, Jacob et al. (2021). "Practical Real Time Recurrent Learning with a Sparse Approximation." In: *International Conference on Learning Representations.*

Morad, Steven et al. (2022). "POPGym: Benchmarking Partially Observable Reinforcement Learning." In: *The Eleventh International Conference on Learning Representations.*

— (2023). "POPGym: Benchmarking Partially Observable Reinforcement Learning." In: *The Eleventh International Conference on Learning Representations.*

Ollivier, Yann, Corentin Tallec, and Guillaume Charpiat (2015). *Training recurrent networks online without backtracking.*

Orvieto, Antonio et al. (2023). *Resurrecting Recurrent Neural Networks for Long Sequences.*

Parisotto, Emilio et al. (2020). "Stabilizing transformers for reinforcement learning." In: *International conference on machine learning.* PMLR.

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio (2013). "On the difficulty of training recurrent neural networks." In: *International conference on machine learning.*

Pavlov, P Ivan (1927). *Conditioned reflexes: an investigation of the physiological activity of the cerebral cortex.*

Radford, Alec et al. (2019). "Language Models are Unsupervised Multitask Learners." In.

Rafiee, Banafsheh et al. (2020). "From eye-blinks to state construction: Diagnostic benchmarks for online representation learning." In: *Adaptive Behavior.*

Schulman, John et al. (2015). "High-dimensional continuous control using generalized advantage estimation." In: *arXiv preprint arXiv:1506.02438.*

Schulman, John et al. (2017). *Proximal Policy Optimization Algorithms.*

Sutton (1995). “Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding.” In: *Advances in Neural Information Processing Systems*. Ed. by D. Touretzky, M.C. Mozer, and M. Hasselmo.

— (2020a). *Markov and Agent State.*

— (2020b). *Toward a New Approach to Model-based Reinforcement Learning.*

Sutton and Andrew G. Barto (2018). *Reinforcement learning: an introduction.* Second edition. Adaptive computation and machine learning series. The MIT Press.

Tallec, Corentin and Yann Ollivier (2017). *Unbiased Online Recurrent Optimization.*

Towers, Mark et al. (2023). *Gymnasium.*

Vaswani, Ashish et al. (2017). “Attention is all you need.” In: *Advances in neural information processing systems.*

Williams and Peng (1990). “An efficient gradient-based algorithm for on-line training of recurrent network trajectories.” In: *Neural computation.*

Williams and Zipser (1989). “A learning algorithm for continually running fully recurrent neural networks.” In: *Neural computation.*

Yang, Zhilin et al. (2019). “XLNet: Generalized Autoregressive Pretraining for Language Understanding.” In: *Neural Information Processing Systems.*

Zeng, Ailing et al. (2022). *Are Transformers Effective for Time Series Forecasting?* arXiv: `2205.13504 [cs.AI]`.