



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Dynamics of Real Time Recurrent Learning in Sparse Networks

Master Thesis

Kevin Danilo Lopez Andrango

August 7, 2024

Advisors: Nicolas Zucchet, Dr. João Sacramento
Department of Computer Science, ETH Zürich

First and foremost, I will switch to the language my parents taught me. Esta tesis está dedicada a ellos, Néstor Patricio López Farinango y Gessy Marlene Andrango Farinango, quienes con su apoyo incondicional han hecho esto posible una vez más. A mis dos hermanos, Alex Paul López Andrango, y una mención especial para Néstor Omar López Andrango, que a pesar de la diferencia de edad que nos separa, sin darte cuenta has sido un refugio para mi alma las veces que he regresado a Valencia. No me puedo olvidar de los amigos que he hecho durante este inolvidable trayecto: Youssef, Vazlav, Burak, Francesco, Matías, Laurenz y muchos más. Emigrar no es fácil, pero con personas como ustedes, uno puede encontrar un hogar en un país extraño.

Abstract

Learning online the parameters of a Recurrent Neural Network (RNN) holds the promise of offering better sample complexity when compared to offline algorithms such as Backpropagation Through Time. We benefit from the stable parametrization of the Linear Recurrent Unit, to study the training dynamics of the Real Time Recurrent Learning (RTRL) algorithm in the class of linear RNNs. We show empirically and justify theoretically that RTRL benefits from highly stochastic online updates. From the insights we develop in the linear setting, we then study its implications in highly sparse Continuous Time Recurrent Neural Networks where we observe similar results. Overall, our findings allow us to offer a possible explanation for the phenomena observed by other researchers when training highly sparse RNNs with RTRL.

Contents

Contents	iii
1 Introduction	1
1.1 Recurrent Neural Networks	1
1.2 Real Time Recurrent Learning	2
2 Scaling RTRL with SnAp-N	5
2.1 Sparse N-Step Approximation	5
2.2 Sparse Matrices	8
2.3 Sparse Jacobian Computation	9
2.4 Sparse JACREV	12
2.5 Masked Sparse Matrix Multiplication	13
3 RTRL Training Dynamics	21
3.1 Online vs Offline	21
3.2 Teacher Student Analysis	24
3.2.1 Online EVGP	24
3.2.2 Linear Recurrent Unit	25
3.2.3 Sample Complexity Study	27
3.3 Benefits of Noisy In-Sequence Updates	29
3.3.1 RTRL Toy Model	33
3.4 Implications In Sparse Networks	39
4 Conclusions	47
Bibliography	49

Chapter 1

Introduction

1.1 Recurrent Neural Networks

Computational neuroscience often utilizes firing rate models to simplify neural dynamics [1], focusing on average firing rates of neurons rather than individual spikes. These models encapsulate the neuron’s activity changes through continuous-time dynamical systems. Mirroring these, Recurrent Neural Networks (RNNs) [12] offer a discrete-time approximation of neural dynamics, akin to the Euler method used to approximate the solution of differential equations. This approach enables RNNs to replicate the progression of neural activities in discrete steps, serving as a bridge between computational neuroscience and machine learning, and opening avenues to modify RNNs for more biologically realistic neural modeling.

The training of RNNs leverages algorithms such as Real-Time Recurrent Learning (RTRL) [33], and backpropagation through time (BPTT) [32], both grounded in gradient descent. RTRL adjusts network weights in real-time with data input, facilitating dynamic adaptation, whereas BPTT unfolds the network across time to capture temporal dependencies. However, the practicality of both RTRL and BPTT diverges from biological neural networks [10]. BPTT’s extensive memory requirements for storing sequences and RTRL’s high computational complexity are not reflective of the more efficient and constrained nature of biological neural systems. Despite their computational capabilities, these methods underscore the existing gap between machine learning models and the operational principles of real neural networks.

In addition to the aforementioned issues, the training of RNNs faces the exploding-vanishing gradient problem [28], which significantly affects long-term learning. Recently [27], inspired by the remarkable performance of state space models in the long-range arena, introduced a new RNN unit, the **Linear Recurrent Unit**. Notably their work highlights the impressive expressiveness capacity of linear RNNs when combined with nonlinear multilayer perceptrons.

This discovery serves as a compelling motivation for studying linear RNNs as linear dynamical systems, where theoretical analysis becomes more tractable compared to their non-linear counterparts. A notable example of this research direction is presented in [30], where they trained RNNs on tasks inspired by neuroscience literature. The study revealed that the perturbation to the RNN initial connectivity matrix after training is of low rank. This observation was further explored under the assumption that the RNN operates linearly, providing valuable insights into the network’s learning dynamics.

In this thesis, we focus on RTRL, targeting two primary audiences. The first audience comprises research engineers interested in the nuances of implementing RTRL and optimizing its performance through approximation variants. The second audience includes computational neuroscientists seeking to understand RTRL. Chapter 2 is tailored to the first audience, assuming a general computer science knowledge that every engineer must have acquired during its training. Chapter 3, which we consider the most interesting from a computational perspective, is directed at the second audience.

Despite its apparent simplicity, the theoretical properties of RTRL are far from being understood. In Chapter 3, we aim to contribute to this understanding by investigating its dynamics through a series of carefully designed experiments, all firmly based on theoretical foundations. In line with the studies just introduced [30], [27], we will start by setting our focus in linear RNNs. This focus in the linear regime first, is intended to shed light on qualitative aspects of RTRL, potentially providing insights into its operation with non-linear RNNs. We consider important to emphasize that for our next studies our emphasis is on exploring qualitative differences rather than performance, particularly in comparison to the current standard training algorithm, BPTT.

Before diving into the next chapters, in the next section we introduce in more detail the core algorithm that serves as the foundation for this thesis.

1.2 Real Time Recurrent Learning

Following [23], consider a discrete dynamical system¹ parametrized by $\theta \in \Theta = \mathbb{R}^{\dim(\theta)}$, let $s_t \in \mathcal{S}_t = \mathbb{R}^{\dim(s)}$ be the state variable, for every $t \geq 1$ we define the transition function of the system by:

$$s_t = T_t(s_{t-1}, \theta) \tag{1.1}$$

note the dependence in t for the transition operator T_t . Given a cost function $\mathcal{L}_t : \mathcal{S}_t \mapsto \mathbb{R}$, we aim to compute the gradient $d_\theta \mathcal{L}_t$, this can be easily done by first differentiating from both sides 1.1:

$$\frac{ds_t}{d\theta} = \frac{\partial T_t}{\partial s_{t-1}} \frac{ds_{t-1}}{d\theta} + \frac{\partial T_t}{\partial \theta} \tag{1.2}$$

¹The derivation for RNNs is presented in Chapter 3.

and then by computing $d_{s_t} \mathcal{L}_t$ and applying chain rule we are done. If we happen to receive a new observation and therefore a new $\mathcal{L}_{t+1}$ we don't need to redo all the calculations from before in order to obtain the gradient, since by the recurrence in 1.2 we can just maintain in **memory** $d_{\theta} s_t$ and s_t , and then compute on the fly $\frac{\partial T_{t+1}}{\partial s_t}$ and multiply it with the previous term. What we have just described is the RTRL algorithm. Note however, and here is where we can appreciate the main drawback with RTRL, that the space required to store the Jacobian $d_{\theta} s_t$ is of order $\mathcal{O}(\dim(s) \times \dim(\theta))$. In an RNN the dimension of the parameter θ is h^2 where h denotes the size of the hidden units, therefore we require $\mathcal{O}(h^3)$ storage in RTRL which becomes prohibitive quickly when compared to BPTT². Even worse for RTRL note that in order to update the $d_{\theta} s_t$, in an RNN we perform a matrix multiplication between matrices of size $h \times h$ and $h \times h^2$, if we use the naive matrix multiplication algorithm then the computational complexity for RTRL is in the order of $\mathcal{O}(h^4)$. The previous statements had guided the field to look for approximations of the $d_{\theta} s_t$ the reader is referred to [3], [26], [31] among others. Note however that sparse RNNs where the number of zero entries dominate the number of the non-zero ones, are more amenable to RTRL and reduce their memory and computational requirements, since the number of parameters that actually contribute to the Jacobian gets reduced due to the sparsity of the network. But still even in this setting the computational and memory complexities keep being a bottleneck for the algorithm. In Chapter 2 we focus on the work of [24], where they propose a practical algorithm to reduce even more the memory and computational requirements from a sparse network by using a biased approximation to $d_{\theta} s_t$.

To the best of our knowledge we could not find any work done in studying the qualitative aspects of RTRL optimization, except for the work in [23] where the authors offer the first local convergence proof for RTRL optimization with online updates³. More recently [17] explore the use of tractable RTRL architectures in a Reinforcement Learning (RL) setting, their conclusion is that there is no practical advantage of RTRL, even when the Jacobian its not approximated, but no other analysis was made. Chapter 3 does this analysis even though in different settings as we don't consider in this work RL.

²BPTT uses vector Jacobian products and reverse mode differentiation, therefore storage is $\mathcal{O}(\dim(\theta))$.

³The notion of online updates is discussed extensively in Chapter 3.

Chapter 2

Scaling RTRL with SnAp-N

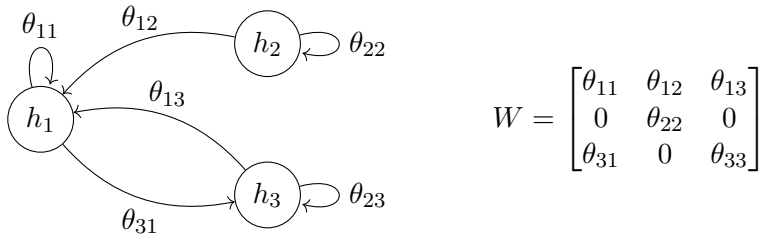
In the introductory chapter, we introduced the core algorithm that serves as the foundation for this thesis. We highlighted the significant challenges posed by memory space and computation time when training networks using RTRL.

In this chapter, we present an algorithm proposed by [24], which takes advantage of the sparsity of the networks to reduce memory and compute time. As outlined in their publication, the theoretical gains of this algorithm hinge on improved software support. We aim to bridge this gap by implementing optimized kernels for computing sparse Jacobians and performing masked matrix multiplications.

Our implementation achieves a **64x** speedup in wall-clock time compared to the original code provided by [24]. Furthermore, our framework is open-sourced and sufficiently flexible to facilitate the implementation of new architectures with ease.

2.1 Sparse N-Step Approximation

Consider the following connectivity matrix W of a recurrent neural network and the associated computational graph for one time step:



after one step of the RNN, the parameter θ_{22} will have influenced only h_2 . After two steps, via θ_{12} , it will begin to influence h_1 , and after three steps, it

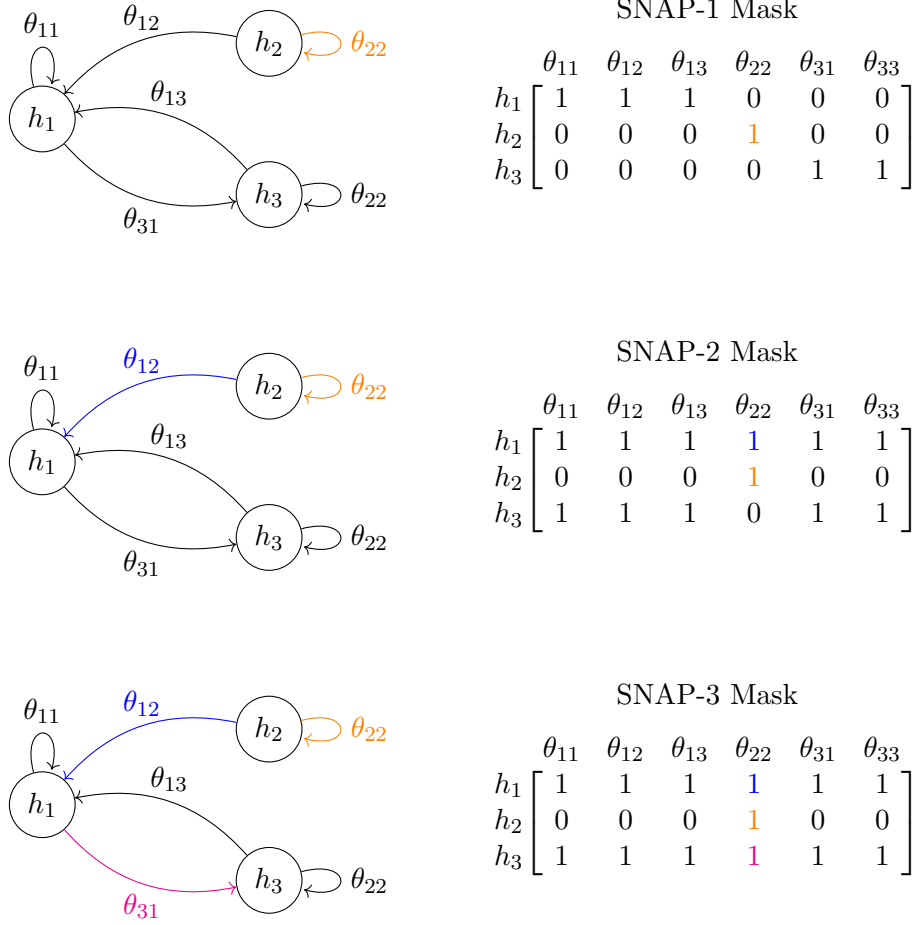


Figure 2.1: Masks induced by Sparse N-Step Approximation algorithm in $W = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ 0 & \theta_{22} & 0 \\ \theta_{31} & 0 & \theta_{33} \end{bmatrix}$

will have affected all the hidden units of the network. Clearly, this influence **after n steps** will impact $\frac{\partial h_{n,i}}{\partial \theta_{22}}$ in a different manner for each hidden unit i .

[24] introduced the **Sparse n-step Approximation** (SnAp-n) for the Jacobian of the network based on the previous observation. More formally, for all t :

$$\hat{J}_t(i, j) = \begin{cases} J_t(i, j) & \text{if } \theta_j \text{ influences hidden unit } h_{n,i} \\ 0 & \text{otherwise} \end{cases} \quad (2.1)$$

this approximation essentially defines a mask matrix M_n (Figure 2.1), that will induce sparsity in the Jacobian, based on the sparsity of the network.

This n -step approximation can be integrated within the algorithmic framework introduced by [3] for algorithms that approximate the J_t in RTRL (Algorithm 1). There they define a function $\text{Ap}_{\otimes}(\cdot)$ as a function that computes either a

biased or unbiased approximation of its input and stores it in the prescribed format $\circledast$. We choose to work within this framework since it provides a clear description of the terms that need to be computed.

Algorithm 1 One step of approximating RTRL

Input: input x_t , hidden state h_{t-1} , parameters θ , approximation $\hat{J}_{t-1}$ of J_{t-1} stored in prescribed format $\circledast$.

Output: hidden state h_t , approximation $\hat{J}_t$ of J_t in format $\circledast$.

Preliminary calculations

1: $h_t \leftarrow$ new hidden state, based on h_{t-1} , θ , x_t

2: $D_t \leftarrow \frac{\partial h_t}{\partial h_{t-1}}$, $I_t \leftarrow \frac{\partial h_t}{\partial \theta}$

Bring addends $D_t \hat{J}_{t-1}$, I_t in desired format $\circledast$

3: $A_1 \leftarrow \text{Ap}_{\circledast}(D_t \hat{J}_{t-1})$

4: $A_2 \leftarrow \text{Ap}_{\circledast}(I_t)$

Mix two addends A_1 , A_2

5: $\hat{J}_t \leftarrow \text{Mix}(A_1, A_2)$, where $\text{Mix}(x, y)$ is an approximator of $x + y$ in format $\circledast$

In the case of SnAp-n:

$$\text{Ap}_{\circledast, n}(X) = M_n \odot X$$

therefore the RTRL equations can now be written as :

$$\begin{aligned} \hat{J}_0 &= \mathbf{0} \\ \hat{J}_t &= \text{Ap}_{\circledast, n}(D_t \hat{J}_{t-1}) + \text{Ap}_{\circledast, n}(I_t) = \text{Ap}_{\circledast, n}(D_t \hat{J}_{t-1}) + I_t \end{aligned} \quad (2.2)$$

note that $\text{Ap}_{\circledast, n}(I_t) = I_t$ since for any $n \geq 1$, M_n its not performing any effective masking¹ on I_t .

Since the same mask is applied at each time step, we can exploit this uniformity to optimize computational efficiency. Specifically, as highlighted by the authors of SnAp-n, it is possible to transition from the naive matrix multiplication algorithm, which operates in $\mathcal{O}(mpk)$ time (for matrices $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times p}$), to an algorithm with a complexity of $\mathcal{O}(\text{nnz}(M)k)$, where nnz denotes the number of non-zero elements in the mask matrix M . Furthermore, if the storage format $\circledast$ is designed to leverage the sparsity of the elements involved in the computation, we can achieve significant memory savings proportional to the sparsity of the mask. In this chapter, we will present various algorithms designed to achieve these computational improvements. More specifically we will show how to compute $\text{Ap}_{\circledast, n}(D_t \hat{J}_{t-1})$, $\text{Ap}_{\circledast, n}(I_t)$, and $\text{Mix}(A_1, A_2)$, with memory cost bounded by $\mathcal{O}(\text{nnz}(M_n))$.

¹This only holds for RNNs in which a parameter θ_i only affects one hidden unit in one time step as noted by the authors.

2.2 Sparse Matrices

In order to implement Algorithm 1, we have to decide on the prescribed format $\circledast$ for storing the Jacobians. Given that we are inducing sparsity through SnAp-n, we can utilize one of the various formats developed over the past decades to represent sparse matrices. Among these, we opt to work with the Coordinate List (COO) format and the Compressed Sparse Row (CSR) format.

COO stores an array of non-zero elements and their corresponding row and column indices. In contrast, the CSR (Figure 2.2) format enhances memory efficiency and computational performance by storing the non-zero elements in a single array (data), along with two auxiliary arrays indicating the start of each row (indptr) and the column indices (indices) of the elements, this structure allows for faster row access operations when compared to COO.

For the purposes of implementing SnAp-n we will choose COO as the prescribed format $\circledast$. This choice is motivated by the following considerations.

The first one being that efficient algorithms exist to convert it to other sparse formats, which might be more suitable for performing linear algebra operations. Converting from COO to CSR, when the indices are sorted lexicographically can be done by computing a prefix sum, which in turn can be performed efficiently using a parallel associative scan ([4]). This becomes important when in runtime, at every step of RTRL we have to switch to CSR for faster row access operations.

The second one is more SnAp-n specific. Suppose we can compute $A_1 \leftarrow \text{Ap}_{\circledast,n}(D_t \hat{J}_{t-1})$ and $A_2 \leftarrow \text{Ap}_{\circledast,n}(I_t)$ (See 2.3, 2.5) in any prescribed sparse format $\circledast$. For $n > 1$ the non-zero entries induced by SnAp-n in A_1 and A_2 will differ and consequently also their associated data structures, as A_2 is independent of n (Recall 2.2). Therefore we need to devise an efficient method to add both sparse terms. When using the COO format, we have that $A_2.\text{indices} \subseteq A_1.\text{indices}$ and implementing $\text{Mix}(A_1, A_2)$ becomes trivial. This is achieved by maintaining an array that maps each index in $A_2.\text{indices}$ to its corresponding position in $A_1.\text{indices}$, so that we have:

$$A_1.\text{indices}[A_2.\text{pos}[k]] = A_2.\text{indices}[k]$$

addition of both sparse terms can now be implemented as:

$$(A_1, A_2) \mapsto A_1.\text{data}.\text{at}[A_2.\text{pos}].\text{set}(A_1.\text{data}[A_2.\text{pos}] + A_2.\text{data}) \quad (2.3)$$

This approach however requires storing an extra array, $A_2.\text{pos}$, increasing the cost for storing A_2 , but note that in general $\text{nnz}(A_1) \gg \text{nnz}(A_2)$ for $n > 1$, so its the data structures from A_1 that will dominate the memory. Also 2.3 is a pure function, where in addition all the array shapes involved are static and known at compile time, which will come in handy at later sections. The

$$A = \begin{bmatrix} 10 & 0 & 0 & 12 & 0 \\ \textcolor{blue}{(0,0)} & & & \textcolor{blue}{(0,3)} & \\ 0 & 0 & 11 & 0 & 13 \\ & & \textcolor{blue}{(1,2)} & & \textcolor{blue}{(1,4)} \\ 0 & 16 & 0 & 0 & 0 \\ & \textcolor{blue}{(2,1)} & & & \\ 0 & 0 & 11 & 0 & 13 \\ & & \textcolor{blue}{(3,2)} & & \textcolor{blue}{(3,4)} \end{bmatrix}$$

COO Format:

$$\begin{aligned} \text{data} &= [10 \ 12 \ 11 \ 13 \ 16 \ 11 \ 13] \\ \text{indices} &= [\textcolor{blue}{(0,0)} \ \textcolor{blue}{(0,3)} \ \textcolor{blue}{(1,2)} \ \textcolor{blue}{(1,4)} \ \textcolor{blue}{(2,1)} \ \textcolor{blue}{(3,2)} \ \textcolor{blue}{(3,4)}] \end{aligned}$$

CSR Format:

$$\begin{aligned} \text{data} &= [10 \ 12 \ 11 \ 13 \ 16 \ 11 \ 13] \\ \text{indices} &= [\textcolor{blue}{0} \ \textcolor{blue}{3} \ \textcolor{blue}{2} \ \textcolor{blue}{4} \ \textcolor{blue}{1} \ \textcolor{blue}{2} \ \textcolor{blue}{4}] \\ \text{indptr} &= \begin{bmatrix} 0 & 2 & 4 & 5 & 7 \\ \textcolor{blue}{(0)} & \textcolor{blue}{(1)} & \textcolor{blue}{(2)} & \textcolor{blue}{(3)} & \textcolor{blue}{(4)} \end{bmatrix} \end{aligned}$$

Figure 2.2: Matrix A and its sparse representations in COO and CSR format.

mixing operation can also be computed avoiding any extra storage at the expense of $\Omega(\text{nnz}(M_n))$ more operations per step, reducing significantly the time performance.

Having motivated our choice for $\otimes$ in the following section we will present an algorithm to compute $\text{Ap}_{\otimes}(I_t)$ without explicitly computing the dense I_t . To this end, we will employ a **matrix coloring** technique that will substantially reduce the space required for storage, as well as the computational effort needed.

2.3 Sparse Jacobian Computation

Let $f: \mathbb{R}^4 \mapsto \mathbb{R}^4$, and assume that for regardless of the input to the function, the Jacobian always keeps the following sparse structure :

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & 0 & 0 \\ 0 & 0 & \frac{\partial f_2}{\partial x_3} & \frac{\partial f_2}{\partial x_4} \\ 0 & 0 & 0 & \frac{\partial f_3}{\partial x_4} \\ \frac{\partial f_4}{\partial x_1} & 0 & 0 & 0 \end{bmatrix} \quad (2.4)$$

Ideally we would like to take advantage of the sparsity in the Jacobian to minimize the number of evaluations required for its determination. [11] observed

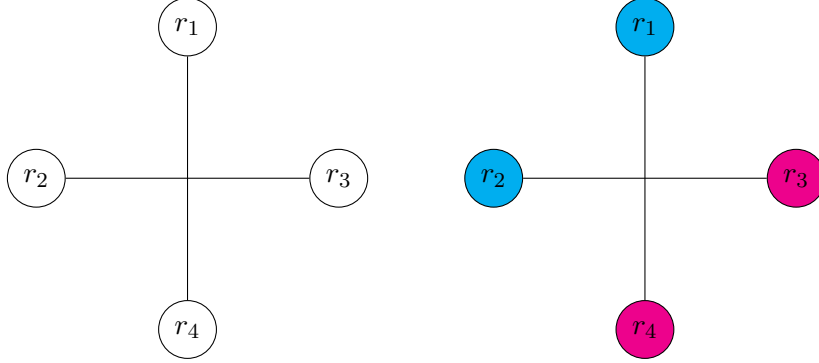


Figure 2.3: Induced graph $G(\mathbf{J})$ and its coloring.

that a group of rows could be determined through a single evaluation, provided no two rows in the group contain a nonzero element in the same column position. Such a group of rows is defined as *structurally orthogonal* to each other, which differs from the common notion of orthogonality in vector spaces. A set of structurally orthogonal groups that contains all the rows of the matrix is termed a *consistent partition* of the matrix. A matrix clearly can be trivially partitioned into a set of structurally orthogonal groups, one group for each row, the objective therefore its to find a consistent partition with the minimum number of groups.

Define now $G(\mathbf{J})$ to be a graph with vertices $V = \{r_1, r_2, r_3, r_4\}$ and edges (r_i, r_j) if and only if $i \neq j$ and rows r_i and r_j in 2.4 have a nonzero in the same column position (See figure 2.3). If we color the graph so that no two adjacent vertices have the same color, we can obtain a consistent partition of the matrix where the groups are defined by the color of each vertex. This connection between finding consistent partition of the matrix and coloring the associated graph, was first introduced by [9].

The resulting matrix coloring of $\mathbf{J}$ will look like this:

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & 0 & 0 \\ 0 & 0 & \frac{\partial f_2}{\partial x_3} & \frac{\partial f_2}{\partial x_4} \\ 0 & 0 & 0 & \frac{\partial f_3}{\partial x_4} \\ \frac{\partial f_4}{\partial x_1} & 0 & 0 & 0 \end{bmatrix}$$

Clearly we can define now $\mathbf{v}_1 = [1 \ 1 \ 0 \ 0]$ and $\mathbf{v}_2 = [0 \ 0 \ 1 \ 1]$, and perform a *blocked* vector-Jacobian product to obtain the *compressed Jacobian*, from which the non zero elements of $\mathbf{J}$ can easily be accessed and stored in $\otimes$:

$$\begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{bmatrix} \mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_2}{\partial x_3} & \frac{\partial f_2}{\partial x_4} \\ \frac{\partial f_4}{\partial x_1} & 0 & 0 & \frac{\partial f_3}{\partial x_4} \end{bmatrix} \quad (2.5)$$

Through this chapter we will refer to the matrix used to left multiply the Jacobian and retrieve the non-zero elements as the *Jacobian projection matrix* $\mathbf{V}$. For a detailed description of the algorithm, please refer to Algorithm 2.

Algorithm 2 Non-Zero Entries of Sparse Jacobian Using Graph Coloring.

Input: Jacobian $\mathbf{J}$, Sparsity Pattern $\mathbf{M}$.
Output: Compressed Jacobian $\mathbf{S}$.

- 1: Compute the induced graph $G(\mathbf{M})$.
- 2: Color $G(\mathbf{M})$ and save the groups $\{E_1, \dots\}$.
- 3: $c \leftarrow$ Number of colors used to color $G(\mathbf{M})$.
- 4: $n \leftarrow$ Number of rows in $\mathbf{M}$.
- 5: **for** $i \leftarrow 1$ to c **do**
- 6: $\mathbf{v}_i \leftarrow \mathbf{0} \in \mathbb{R}^n$
- 7: **for** $k \leftarrow 1$ to n **do**
- 8: $\mathbf{v}_i[k] \leftarrow 1$ if and only if row $k \in E_i$
- 9: **end for**
- 10: **end for**
- 11: $\mathbf{V} \leftarrow [\mathbf{v}_1^\top \dots \mathbf{v}_c^\top]^\top$
- 12: $\mathbf{S} \leftarrow \mathbf{VJ}$

In general for a function $f: \mathbb{R}^n \mapsto \mathbb{R}^m$, the number of entries required to store the Jacobian is reduced by a fraction of $\frac{c}{m}$, where c is the number of colors used in the algorithm. For a standard RNN with a hidden size of h , storing the dense $\frac{\partial h_t}{\partial \theta}$ requires $\mathcal{O}(h^3)$ memory. In contrast, for the compressed version, the memory requirement is reduced to $\mathcal{O}(h^2)$.

While the matrix coloring algorithm enables us to uniquely determine the non-zero entries in a sparse Jacobian, a significant challenge remains: the explicit Jacobian matrix is often unavailable and computing it directly is infeasible due to its large memory requirements. To address this, we can leverage automatic differentiation tools, particularly those that implement the vector-Jacobian product (VJP) operation. These tools allow us to compute the necessary derivatives implicitly, without forming the entire Jacobian matrix.

A prominent example of such a tool is **JAX** [5], a high-performance library designed for numerical computing and machine learning research. JAX provides robust automatic differentiation capabilities, allowing us to compute the necessary derivatives implicitly, without forming the entire Jacobian matrix. By integrating matrix coloring with JAX's VJP functionality, we can efficiently handle sparse Jacobians, thus circumventing the substantial storage and computational costs associated with the explicit Jacobian computation.

Implementing Algorithm 2 would be straightforward if we restricted ourselves to functions accepting a single rank-1 array, since JAX through its `jax.vjp` interface combined with `jax.vmap`, facilitates the computation of $\mathbf{S} \leftarrow \mathbf{VJ}$

in a one-liner. However, to provide a clean interface and generalize our implementation to accommodate any RNN architecture, we must support the computation of sparse Jacobians for functions that accept arbitrary PyTrees². For the remainder of this chapter we will refer to an arbitrary PyTree as θ , and to its leaves as θ_i .

2.4 Sparse JACREV

Assume for simplicity that $f : \Theta \times \mathbb{R}^n \rightarrow \mathbb{R}^m$. JAX of course allows us to compute the VJP for functions accepting an arbitrary PyTree (Figure 2.4), however there is one caveat. Once the VJP of f is obtained, the cotangent vector will be applied for every leaf in the PyTree. Since not every leaf will produce the same sparsity pattern in the Jacobian, and therefore not the same Jacobian projection matrix V_i , what we want is to compute $V_i d_{\theta_i} f$ using `jax.vjp` for every leaf in θ without computing $V_i d_{\theta_k} f$ for $k \neq i$, so that we don't waste space or perform unnecessary computations (Figure 2.5).

To the best of our knowledge there is no such functionality available, therefore we dig into the internals of JAX to allow such computations. Our implementation supports dense leaves and sparse leaves represented in the COO format, under the same function. Furthermore, in computing $\frac{\partial f}{\partial \theta_i}$ for a sparse θ_i the dimensions of the Jacobian will reflect the sparsity of the leaf, obtaining a Jacobian of dimensions $\mathbb{R}^{m \times \text{nnz}(\theta_i)}$. This allows us to obtain the memory gains in sparse recurrent neural networks, outlined by [24], even when there is no approximation made to the Jacobian when using the RTRL algorithm.

With the previous functionality in hand, we propose the following interface for implementing Algorithm 2:

- `def make_jacobian_projection(`
`jacobian_mask: PyTree`
`) -> PyTree:`

Must receive a PyTree with the same structure as θ , where at each leaf must contain the sparsity pattern of its corresponding Jacobian. It will then compute the coloring of the induced graph for each leaf using **networkx** ([14]) as described in section 2.3, and then return the Jacobian projection matrices in a PyTree with the same structure as θ .

- `def sp_jacrev(`
`f: Callable,`
`V: PyTree,`

²A PyTree is (Verbatim from [18]) any arbitrarily nested composition of 'node' types (dictionaries, lists, and tuples) containing 'leaf' types (all other Python types as well as Rank-N Arrays).

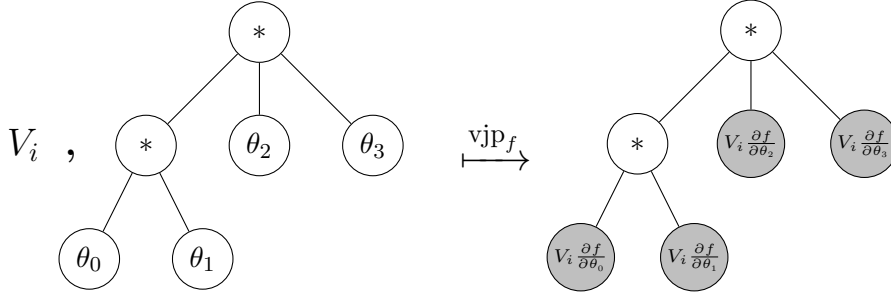


Figure 2.4: JAX VJP for a PyTree. Note that the contangent vector V_i is applied for every leaf, independently if we wanted to apply it just for one leaf in specific.

```

    argnums: int | sequence[int] = 0
) -> Callable:

```

Mimics almost one-to-one the current `jax.jacrev` that allows the user to compute the dense jacobians, with the difference that ours allows the user to pass a PyTree with the Jacobian projection matrices to obtain the sparse Jacobians. Our implementation is fully compatible with all the JAX high level operations (`jit`, `grad`, `vmap`). The returned callable will compute the sparse jacobians evaluated at the primals, and return them stored in COO.

We conduct benchmarks of our implementation to measure the predicted memory savings when computing $I_t := \frac{\partial h_t}{\partial \theta}$, for both a tanhRNN and standard GRU. As illustrated in Figure 2.6 and Figure 2.7, our approach demonstrates both memory and computation cost advantages when using the matrix coloring technique in comparison to the explicit computation of the Jacobian.

For profiling the memory we used **pprof** as recommended by the JAX developers. To ensure the correctness of our benchmarks, we provide the snapshots of the memory state for each method where we must see a linear tree in which the root must indicate the function being used (Figure 2.8).

Given that we can now compute I_t for any arbitrary recurrent neural network, $f(\theta, h_{t-1}, x_t)$, and store it in the prescribed format as defined by $\circledast$, in the next section we will implement a kernel to compute $\text{Ap}_{\circledast, n}(D_t \hat{J}_{t-1})$ exploiting the sparsity of the mask M_n involved. This will substantially reduce the computational cost, when the zero elements of the mask dominate the non-zero elements.

2.5 Masked Sparse Matrix Multiplication

Sparse matrix computations are crucial in scientific and engineering applications, enabling efficient handling of large-scale data. Algorithms for sparse

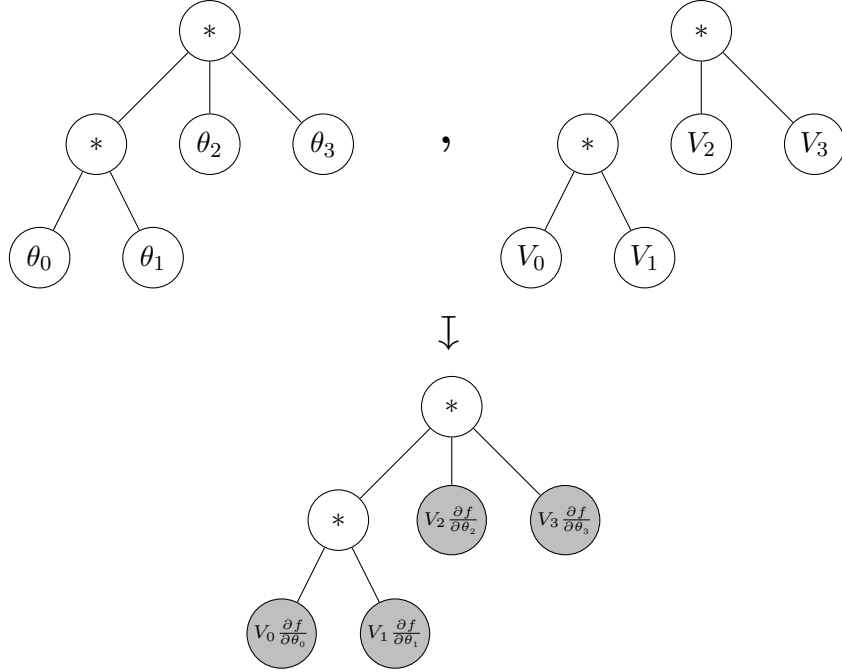


Figure 2.5: Custom vjp function where we accept the primal PyTree and another PyTree that contains at each leaf a different contingent vector that will be applied to the matching leaf in the primal PyTree.

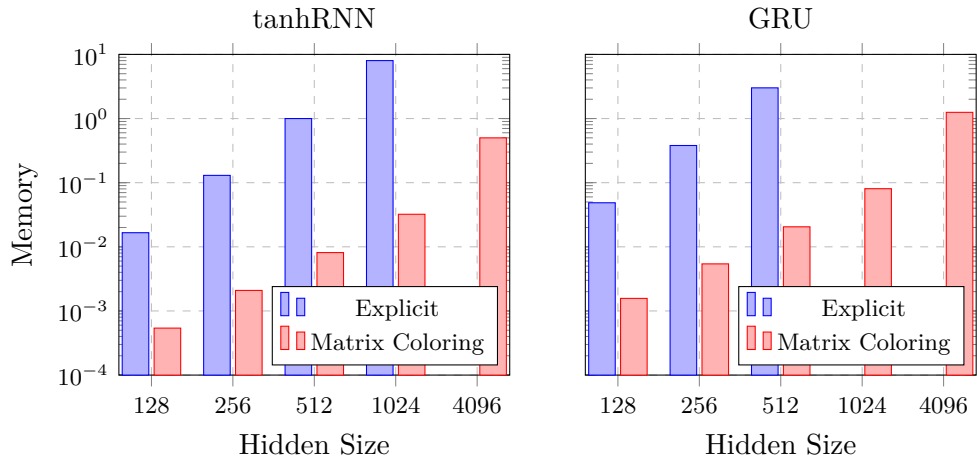


Figure 2.6: Memory allocated in the GPU, **log scale**, to compute $I_t := \frac{\partial h_t}{\partial \theta}$ for different architectures in a NVIDIA RTX 3090. Bars not showed indicate that we could allocate the dense Jacobians in memory. Note that it requires storage $\mathcal{O}(H^3)$ where H is the number of hidden units.

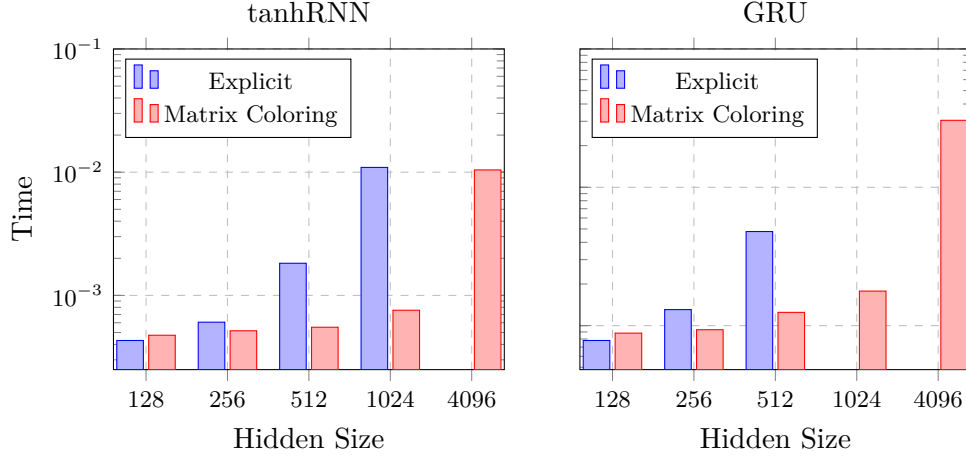


Figure 2.7: Time to compute $I_t := \frac{\partial h_t}{\partial \theta}$ for different architectures in a NVIDIA RTX 3090. Bars not showed indicate that we could allocate the dense Jacobians in memory. Note that it requires storage $\mathcal{O}(H^3)$ where H is the number of hidden units.

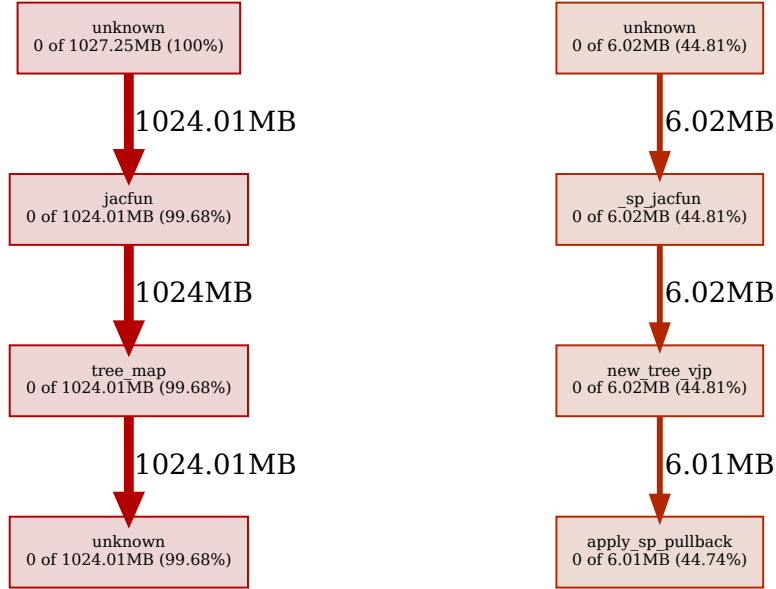


Figure 2.8: Flamegraph obtained by pprof of `jax.jacrev` and `sp_jacrev`, for a tanhRNN with a hidden size of 512 units. We omit the rest of the leaves below the last node for better visualization purposes.

matrix operations, such as sparse matrix-sparse matrix multiplication (SpMM) and dense-sparse matrix multiplication (DSMM) reduce memory usage and computational complexity by exploiting the sparsity of the operands.

In recent years, JAX has made significant progress enhancing its support for sparsity on GPUs by integrating with cuSPARSE, NVIDIA’s library for sparse matrix operations. Despite these advancements numerous operations provided by cuSPARSE are not yet fully supported within JAX.

In this section, we introduce an algorithm to address a specific operation:

$$R = M \odot DX \tag{2.6}$$

where $D \in \mathbb{R}^{m \times k}$ is dense, $X \in \mathbb{R}^{k \times p}$ is a sparse matrix stored in $\circledast^3$, and M is a masking matrix also in $\circledast$. Since the mask matrix induces sparsity in DX , we aim for the resulting R to be in $\circledast$ as well. This operation is particularly relevant to us since recall that $\text{Ap}_{\circledast,n}(D_t \hat{J}_{t-1}) = M_n \odot (D_t \hat{J}_{t-1})$. Therefore implementing 2.6 efficiently will allow us to finalize the implementation of Algorithm 1 under the SnAp-n approximation.

Implementing 2.6 can be done in $\mathcal{O}(\text{nnz}(M)k)$ using the most naive algorithm, where for each $M_{i,j} \neq 0$ we perform the dot product $D_{i,:} X_{:,j}$. This algorithm is also **embarrassingly parallelizable**, excluding any parallelism within the dot product itself. To exploit this parallelism we could use `jax.vmap` in combination with JAX’s support of the COO⁴ format. However as we found, this approach is highly inefficient. The inefficiency arises from accessing a column in the COO format, which is inherently slow. One possible solution is to convert X from the COO format to a CSR format, and compute instead $M^\top \odot (X^\top D^\top)$, as CSR allows for efficient row access. Unfortunately, the JAX library does not support row slicing in CSR format. We believe the reason for this limitation is that accessing a row in CSR format does not satisfy JAX’s static shape constraints. Specifically, in CSR format, a row is accessed using `X.data[indptr[i]:indptr[i+1]]`, and this form of dynamic slicing is not permitted within just-in-time compiled (JIT) functions, which is one of the main reasons to use JAX.

Considering these constraints, we opted to implement the **Masked Dense Sparse Matrix Multiplication (Masked DSMM)** as a primitive operation in JAX. This implementation ensures compatibility with JIT compiled functions. The primitive operates on CSR sparse matrices, and the algorithm is detailed in Algorithm 3.

For the CPU support we implemented the primitive in NUMBA ([20]) and take advantage of their `cfunc` decorator that creates a compiled function callable

³ $\circledast := \text{COO}$

⁴JAX actually prioritizes support for BCOO (Batch COO) and BCSR (Batch CSR). We work with these formats but in the text we just name them COO and CSR respectively.

Algorithm 3 Masked Sparse-Matrix Dense-Matrix Multiplication

Input: Mask M , A_{CSR} , B .
Output: $C_{\otimes}$.

```

1: procedure MASKED-SPMM( $M$ ,  $A_{\text{CSR}}$ ,  $B$ )
2:    $N \leftarrow |M|$ 
3:   for  $n \leftarrow 1$  to  $N$  do in parallel
4:      $i, j \leftarrow M[n]$ 
5:      $r_s \leftarrow A.\text{indptr}[i]$ 
6:      $r_e \leftarrow A.\text{indptr}[i + 1]$ 
7:     if  $r_s - r_e = 0$  then
8:        $C.\text{data}[n] \leftarrow 0$ 
9:     else
10:       $\mathbf{v} \leftarrow A.\text{data}[r_s : r_e]$ 
11:       $\text{nnz\_cols} \leftarrow A.\text{cols}[r_s : r_e]$ 
12:       $C.\text{data}[n] \leftarrow \mathbf{v} \cdot B[:, \text{nnz\_cols}]$ 
13:    end if
14:  end for
15:   $C.\text{indptr} \leftarrow M$ 
16: end procedure

```

from foreign C code. We then take the address of this JITed function and register it in the XLA compiler through their custom call API. For the GPU support the process becomes more intricate and requires knowledge of Python and C programming. The steps required to add a custom GPU primitive in JAX can be seen in Figure 2.9.

Before delving into CUDA programming, it is prudent to check whether existing libraries already implement the desired kernel. To the best of our knowledge, the primitive described in equation 2.6 is absent in current GPU Sparse General Matrix Multiplication (SpGEMM) libraries. We identified only two relevant libraries: GraphBlast ([34]) and cuSPARSE.

GraphBlast implements the GraphBlas API specification ([6]), which stipulates that a compliant library should include the primitive in 2.6. However, an examination of GraphBlast’s source code reveals that it does not support masking when one of the operands is dense, as required for our application. Although we could convert our first operand D to a sparse format to use their implementation, this approach is infeasible. For our use case D represents $\frac{\partial h_t}{\partial h_{t-1}}$, which can indeed be sparse and constant. However, outside the regime of a Linear RNN, the sparsity of the dynamics Jacobian varies depending on the input. This variability violates JAX’s static shape constraints when constructing a COO or CSR matrix, as the number of non-zero entries must be static and known at compile time.

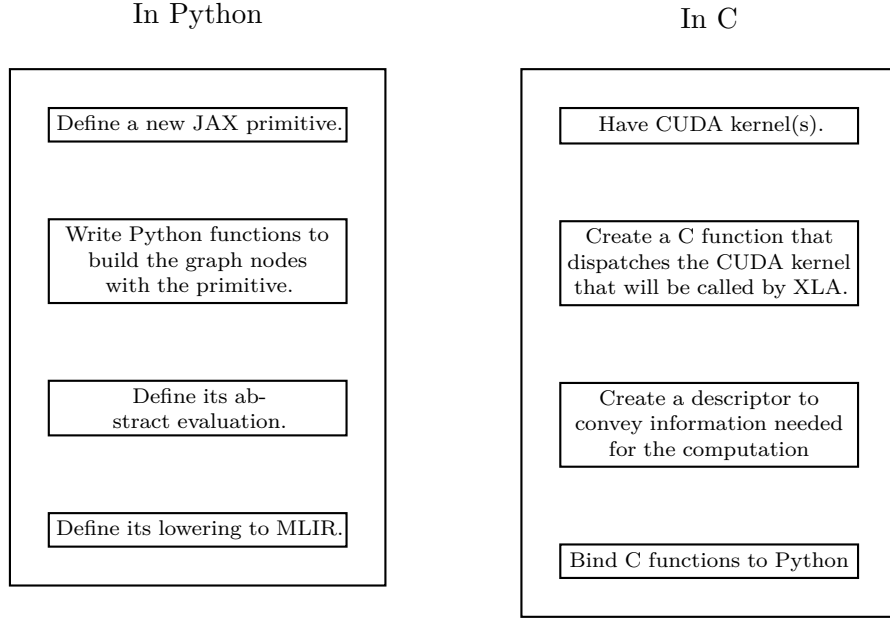


Figure 2.9: Steps required to add a new CUDA primitive to JAX.

On the other hand, cuSPARSE offers the function `cusparsesDDMM` for masking the output, which only works with dense inputs to produce a sparse output. For RNNs trained with Real-Time Recurrent Learning (RTRL), reverting to a dense representation of the Jacobians is prohibitive, as discussed in the previous section. This limitation in existing libraries motivated us to create our own custom kernels, specifically tailored to address the requirements of our Masked Dense Sparse Matrix Multiplication (Masked DSMM) operations.

To invoke a CUDA kernel, we must specify the grid size and block dimensions. Since Algorithm 3 parallelizes over the non-zero elements of the mask we opted for a one-dimensional grid where the grid size is determined by $\lceil \text{nnz}(M)/B \rceil$. Here B indicates the number of threads per block, which we set to be 1024.

With the Masked DSMM primitive established, we can now fully implement Algorithm 1. Figure 2.10 showcases benchmarks of our implementation, contrasting it with the original code provided by [24] for SnAp-1. Our implementation demonstrates an order of magnitude increase in speed compared to the original code, thereby validating the optimizations developed throughout this chapter. Specifically, our implementation aligns more closely with the theoretical improvements outlined in their publication, particularly when comparing SnAp-1 with Backpropagation Through Time (BPTT) for hidden sizes less than 512. Figure 2.11 reports the time required to compute the gradients for a full sequence of input-output pairs, rather than just a single step.

For highly sparse networks where SnAp-2 and SnAp-3 are applicable, we are

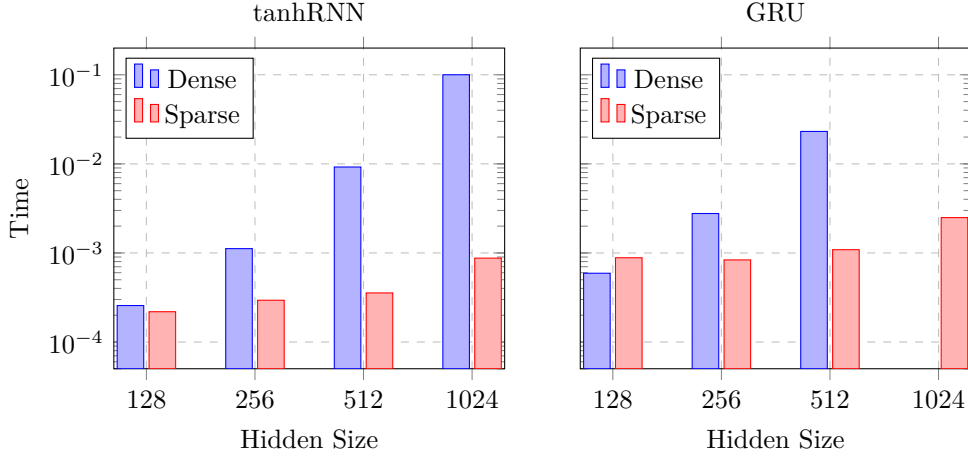


Figure 2.10: Time to compute one step of SnAp-1 for different architectures in a NVIDIA RTX 3090.

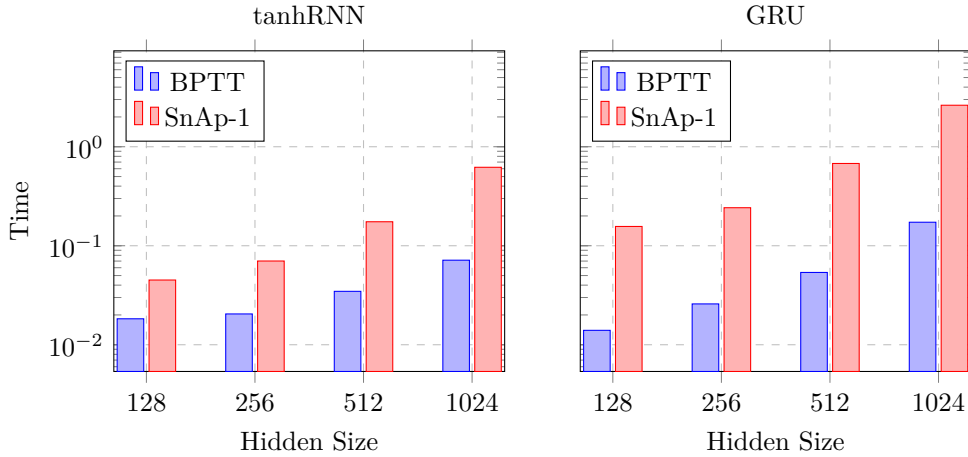


Figure 2.11: Time to compute ∇L for BPTT and SnAp-1 in a NVIDIA RTX 3090.

unable to provide benchmarks as the corresponding code was not released. However during the development of the library we observed that the performance of our Masked DSMM diminishes as the density⁵ of the Jacobian falls below 0.01. Dense Matrix Multiplications are highly optimized operations that exploit memory coalescence, with cache timing being a critical factor at this level of parallelism. Our implementation achieves good memory coalescence when accessing rows of the sparse matrix. However, accessing columns results in numerous cache misses due to non-contiguous memory access patterns. This is a big limitation of our kernel, but we believe there is a lot of improvement can be made, since we just implemented the most naive algorithm we could think of. There is an existing line of research that attempts to improve the

⁵Density is defined as $d := 1 - s$, where s is the level of sparsity.

memory coalescence when using sparse operands, we refer the interested reader to [25], [16],[22], [7] among others.

We conclude this chapter mentioning that the source code for all discussed algorithms is available as an open-source package in [snapjax](#). In addition to the SnAp-N approximations we implement the exact RTRL algorithm for any class of single-layer RNNs. For general multilayer RNNs we implement the approximation rule introduced in [37], and if we restrict to layers following:

$$\begin{aligned} h_t^1 &= f_1(h_{t-1}^1, x_t) \\ h_t^l &= f_l(h_{t-1}^l, h_t^{l-1}) \\ y_g &= g(h_t^L) \end{aligned}$$

we implement the exact RTRL gradient computations.

Chapter 3

RTRL Training Dynamics

In this chapter we discuss an algorithmic bias in RTRL that to the best of our knowledge has not been previously discussed before. We offer a simple explanation of this bias, and we explore further consequences in highly sparse networks where RTRL becomes tractable as discussed in chapter 1.

3.1 Online vs Offline

We start this chapter by introducing the notation that will be used throughout the rest of the text. We define the transition function as $f: \Theta \times \mathcal{S} \times \mathcal{X} \rightarrow \mathcal{S}$, where Θ denotes a finite dimensional vector space where the parameters live, $\mathcal{S}$ denotes the same for the hidden states and $\mathcal{X}$ as well for the inputs to the RNN. A single sequence of inputs will be defined as $\mathbf{x}_{1:T} := (x_1, x_2, \dots, x_T)$ where T denotes the end of the sequence. For when needed we define $f_\theta := f(\theta, \cdot, \cdot)$ and to further declutter notation given a fixed input x_t we define $f_t := (\cdot, \cdot, x_t)$. For a **fixed** parameter θ , given an initial condition h_0 , the sequence of hidden states produced by the network can be recursively defined as:

$$h_t = f(\theta, h_{t-1}, x_t) \quad (3.1)$$

Under this formalism an RNN can be considered a non-autonomous discrete dynamical system.

In what follows we focus on gradient-based learning where the objective is to minimize at each time step t , a cost function $\mathcal{L}_t: \Theta \rightarrow \mathbb{R}$ that measures the network's performance on a given task, as a function of the parameters θ . Note that t can be unbounded, but in this chapter we focus on the case where $t \leq T$. This allows us to define a cumulative loss function:

$$\mathcal{L}(\theta) := \sum_{t=1}^T \mathcal{L}_t(\theta) \quad (3.2)$$

that will measure the overall performance of the network.

In chapter 1 we introduced two different algorithms to compute the gradient $d_\theta \mathcal{L}$. Backpropagation Through Time (BPTT) and Real Time Recurrent Learning (RTRL). Both algorithms are equivalent, in the sense that they compute the same gradient $d_\theta \mathcal{L}$, the difference however relies on how this gradient is computed. BPTT computes the cumulative loss gradient in two passes, a first forward pass where the inputs x_t are processed, and a second backward pass where the gradients are computed. In contrast RTRL allows for computing both in just a single pass, however this comes at a significant computational and memory cost as explained in preceding chapters. More specifically RTRL computes each $d_\theta \mathcal{L}_t$ explicitly as the network runs, whereas BPTT will only compute them implicitly. To better illustrate this difference we can derive the gradient equations for both methods:

In RTRL:

$$\begin{aligned} \sum_{t=1}^T \frac{d\mathcal{L}_t}{d\theta} &= \sum_{t=1}^T \frac{d\mathcal{L}_t}{dh_t} \frac{dh_t}{d\theta} \\ &= \sum_{t=1}^T \frac{d\mathcal{L}_t}{dh_t} \left(\frac{\partial h_t}{\partial h_{t-1}} \frac{dh_{t-1}}{d\theta} + \frac{\partial h_t}{\partial \theta} \right) \quad (\text{RTRL}) \end{aligned} \quad (3.3)$$

In BPTT:

$$\begin{aligned} \sum_{t=1}^T \frac{d\mathcal{L}_t}{d\theta} &= \sum_{t=1}^T \sum_{s=1}^t \frac{\partial \mathcal{L}_t}{\partial h_s} \frac{\partial h_s}{\partial \theta} && (\text{Parameter sharing}) \\ &= \sum_{s=1}^T \left(\sum_{t \geq s}^T \frac{\partial \mathcal{L}_t}{\partial h_s} \right) \frac{\partial h_s}{\partial \theta} && (\text{Exchange of summation}) \\ &= \sum_{s=1}^T \frac{d\mathcal{L}}{dh_s} \frac{\partial h_s}{\partial \theta} && (\text{Total derivative}) \\ &= \sum_{s=1}^{T-1} \left(\frac{d\mathcal{L}}{dh_{s+1}} \frac{\partial h_{s+1}}{\partial h_s} + \frac{d\mathcal{L}_s}{dh_s} \right) \frac{\partial h_s}{\partial \theta} \quad (\text{BPTT}) \end{aligned} \quad (3.4)$$

after the exchange of summations in BPTT we are not computing $d_\theta \mathcal{L}$ by adding each $d_\theta \mathcal{L}_t$, but rather different terms. RTRL computes $d_\theta \mathcal{L}$ by adding each $d_\theta \mathcal{L}_t$ explicitly as we process the inputs.

Being able to obtain the gradient of the instantaneous loss $d_\theta \mathcal{L}_t$ as we process the inputs x_t is the core feature of RTRL. It allows the update of the parameters θ in an online fashion, without having to neither accumulate the gradients nor wait until the sequence has been fully processed. This is what we refer as the **online** training setting, where numerous updates of the parameters θ are performed as we process the inputs, in contrast to the **offline** training

setting in which only one update of the parameters is performed at the end. Throughout the rest of this chapter to remove unnecessary wording we will refer to RTRL with online updates as the online algorithm, and to BPTT as the offline algorithm. A single online update of the parameters while processing an input will be referred to as an **in-sequence** update, to better emphasize its nature.

To make the difference more clear between both training settings, we can make explicit the dependence of the parameters θ on t . In the offline setting we are not performing any update as we run the network, therefore we have $\theta_t = \theta_1$ for $1 \leq t \leq T$, whereas in the online setting in general we have $\theta_i \neq \theta_j$ for $1 \leq i < j \leq T$. We remark that this change of parameters as the network runs only happens while training, and when doing inference on a new input sequence the parameters are fixed for both methods¹.

Online updates of the network parameters can exhibit different training dynamics when compared to their counterpart. This difference arises by noting that while in the offline setting:

$$\frac{dh_t}{d\theta}(\theta_1) = \frac{\partial f_t}{\partial h}(h_{t-1}) \frac{dh_{t-1}}{d\theta}(\theta_1) + \frac{\partial f_t}{\partial \theta}(\theta_1)$$

in online setting we have:

$$\frac{dh_t}{d\theta}(\theta_t) = \frac{\partial f_t}{\partial h}(h_{t-1}) \frac{dh_{t-1}}{d\theta}(\theta_{t-1}) + \frac{\partial f_t}{\partial \theta}(\theta_t)$$

therefore, except for the first time step, in the online setting we are no longer computing the real Jacobian $d_\theta h_t$ but a biased one, that will a priori better reflect the influence of the most recently used parameters $\{\theta_t, \theta_{t-1}, \theta_{t-2}, \dots\}$ in comparison to the initial ones $\{\theta_1, \theta_2, \dots\}$. If we use gradient descent, this bias will imply that we are no longer following the precise negative gradient direction. The difference at each step is nevertheless $\mathcal{O}(\eta_t)$, so one can mitigate this bias by reducing the learning rate. The bias of the $d_\theta h_t$ when using online updates, was indeed noted by the original authors of RTRL [33]:

While the resulting algorithm is no longer guaranteed to follow the gradient of total error, the practical differences are often slight, with the two versions becoming more nearly identical as the learning rate is made smaller.

By making η_t small we are potentially aligning more closely to BPTT training dynamics. Even in this *small learning rate regime* formally analyzing the online algorithm becomes substantially challenging since standard stochastic gradient descent theory does not apply directly, and to this date we could only find one

¹While this is strictly not necessary in the online setting, we believe that a more fair comparison between both algorithms can be done in this way.

work where they proved local converge of RTRL when the network parameters start within a radius close enough to a local minima [23].

Local converge itself will not really tell us much about the qualitative differences on the training dynamics when the network has just been initialized with random parameters. It is the purpose of this chapter to try elucidate the qualitative differences, if any, that arise when using RTRL with online updates in comparison to BPTT. To the best of our knowledge no prior work has been done in this regard, and even though numerous studies have been done in either improving, or analyzing the performance of RTRL in different scenarios, as well as reducing its computational cost, we could not find any prior work in trying to uncover the qualitative differences that could potentially arise.

3.2 Teacher Student Analysis

3.2.1 Online EVGP

As argued in [15] if there is any hope of understanding Recurrent Neural Networks we first need to understand them in the special case where we remove the non-linearities. Albeit in principle this simplifies the analysis, removing the non-linearities however makes the online algorithm more prone to the exploding-vanishing gradient problem [28], when compared to the offline one. To see this, lets first define $J_t := d_{\theta} h_t(\theta_t)$, $D_t := \delta f_t / \delta h(h_{t-1})$, and $I_t := \delta f_t / \delta \theta(\theta_t)$. Equation 3.3 can then be rewritten as:

$$\begin{aligned} \sum_{t=1}^T \frac{d\mathcal{L}_t}{dh_t} \frac{dh_t}{d\theta} &= \sum_{t=1}^T \frac{d\mathcal{L}_t}{dh_t} J_t \\ &= \sum_{t=1}^T \frac{d\mathcal{L}_t}{dh_t} (D_t J_{t-1} + I_t) \end{aligned}$$

unrolling the recurrence and starting with $J_0 = \mathbf{0}$, we can see that:

$$J_t = D_t D_{t-1} \cdots D_2 I_1 + \sum_{k=2}^{t-1} D_t D_{t-1} \cdots D_{k+1} I_k + I_t$$

In the class of linear RNNs² and when using the offline algorithm the **state-transition** Jacobian becomes constant $D_t = W$, therefore by computing J_t we are also computing W^{t-1} . This becomes an issue if the spectral radius³ $\rho(W)$ is strictly bigger than 1 since the gradients are computed using J_t and therefore as t grows we enter the exploding gradient regime. This can be

² $\theta = (W, U)$, $f(\theta, h_{t-1}, x_t) = Wh_{t-1} + Ux_t$

³Let $\lambda_1, \dots, \lambda_n$ be the eigenvalues of a matrix $A \in \mathbb{C}^{n \times n}$. The spectral radius of A is defined as $\rho(A) = \max \{|\lambda_1|, \dots, |\lambda_n|\}$.

partially mitigated by ensuring that at initialization the randomly sampled W satisfies the condition $\rho(W) \leq 1$, however an update of the weights could easily break this inequality.

Even though this is a big concern in both algorithms, note that in the online setting the state-transition Jacobian D_t is not constant anymore, since we are updating W at every time step t . With no non-linearity regulating the matrix products in J_t , a single *bad* in-sequence update could already set the RNN in the unstable regime. Therefore even in the linear setting, studying the dynamics of the online algorithm becomes more difficult. Recall that reducing the learning rate to smaller values in order to control the perturbation of an in-sequence update is ideally not what we are aiming for, since as mentioned before this will lead to training dynamics similar to BPPT. To further make progress in our study, we are going to take advantage of the **Linear Recurrent Unit**, recently introduced by [27].

3.2.2 Linear Recurrent Unit

Let $W \in \mathbb{R}^{k \times k}$ be the connectivity matrix of a linear RNN, $U \in \mathbb{R}^{k \times k}$ the input matrix, and $C \in \mathbb{R}^{d \times k}$ the readout matrix. Given h_0 , and a sequence $\mathbf{x}$ the output y_t is computed as:

$$h_t = Wh_{t-1} + Ux_t, \quad y_t = Ch_t$$

this in fact is equivalent to computing the convolution:

$$\begin{aligned} h_t &= \sum_{j=0}^{t-1} W^j U x_{t-j} \\ y_t &= Ch_t \end{aligned} \tag{3.5}$$

note that diagonalizable matrices are dense in $\mathbb{R}^{k \times k}$ [35], therefore under a small perturbation of W we can decompose it as $W = P\Lambda P^{-1}$, where $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_k) \in \mathbb{C}^{k \times k}$, and $P \in \mathbb{C}^{k \times k}$ is an invertible matrix. Left multiplying equation 3.5 by P^{-1} on both sides, and letting $\tilde{C} := CP$, $\tilde{B} := P^{-1}B$, we obtain:

$$y_t = \tilde{C} \sum_{j=0}^{t-1} \Lambda^j \tilde{B} x_{t-j} \tag{3.6}$$

If the purpose of our study is to first understand the training dynamics of RTRL in the class of linear RNNs, we might as well train linear RNNs where instead of learning $\theta = (W, U, C)$, we learn $\theta = (\Lambda, \tilde{B}, \tilde{C})^4$ and work with complex matrices. Expressiveness will not be lost in this parametrization, but

⁴By this change of parametrization the terms Λ , $\tilde{B}$, $\tilde{C}$ are not related anymore by an invertible P , so to compute the output we take the real part of $\tilde{C}\tilde{h}_t$.

3. RTRL TRAINING DYNAMICS

it still does not ensure stable training dynamics and the EVGP remains a concern in both online and offline training settings.

Recently [27] took the previous approach one step further, and propose parametrizing the elements of Λ as:

$$\lambda_j := \exp(-\nu_j + i\theta_j)$$

where it is the vectors $\nu \in \mathbb{R}^k$ and $\theta \in \mathbb{R}^k$ that will be learned during training. It is easy to see that under this parametrization starting with $\nu_j \geq 0$ for all j guarantees $\rho(\Lambda) < 1$ at initialization. However post-update stability cannot be guaranteed since a ν_j can become negative. To address this, they introduced an additional positive non-linearity leaving the final parametrization as:

$$\lambda_j := \exp(-\exp(\nu_j) + i\theta_j) \quad (3.7)$$

and now stability is ensured post-update. For learning long-range sequences where $\lambda_j \rightarrow 1$, the authors still noted a explosion in the forward pass. To see why⁵, assume a one dimensional network $\lambda \in \mathbb{C}$, and let $\tilde{B} = 1$. Then under white-noise input x_t we can see that:

$$|\tilde{h}_t| = \left| \sum_{j=0}^{t-1} \lambda^j x_{t-j} \right|$$

(Absolute value in $\mathbb{C}$)

$$\implies |\tilde{h}_t|^2 = \left(\sum_{j=0}^{t-1} \lambda^j x_{t-j} \right) \left(\sum_{i=0}^{t-1} \lambda^i x_{t-i} \right)^*$$

(For $i \neq j$, $\mathbb{E}[x_{t-j}x_{t-i}] = 0$)

$$\implies \mathbb{E}[|\tilde{h}_t|^2] = \mathbb{E} \left[\left(\sum_{j,i=0}^{t-1} \lambda^j x_{t-j} (\lambda^i)^* (x_{t-i}) \right) \right] = \left(\sum_{j=0}^{t-1} |\lambda|^{2j} \mathbb{E}[x_j^2] \right)$$

(Geometric series with $r = |\lambda|^2 < 1$)

$$\implies \lim_{t \rightarrow \infty} \mathbb{E}[|\tilde{h}_t|^2] = \frac{1}{1 - |\lambda|^2}$$

therefore as $\lambda \rightarrow 1$ the forward pass explodes since the contribution from x_j takes longer to decay. In order to address this issue they propose to normalize $\tilde{B}x_t$ by an element-wise factor γ defined at initialization as $\gamma := \sqrt{1 - |\lambda|^2}$.

Given the preceding normalization and the exponential parametrization of λ (3.7) the **Linear Recurrent Unit (LRU)** is formally defined by [27] as:

$$h_t = \lambda h_{t-1} + \gamma \odot (Bx_t) \quad (3.8)$$

$$y_t = \text{Re}[Ch_t] + Dx_t \quad (3.9)$$

⁵Derivation from [27].

where $\theta = (\lambda, B, C, D, \gamma)$. The LRU plays a significant role for our next studies as in-sequence updates will no longer break the stability of the linear RNNs. Furthermore as shown by [37], the diagonal nature of the connectivity matrix in the LRU allows for drastically reducing the computational and memory requirements induced by RTRL. This is due to the fact that complex neurons in the LRU do not affect each other and therefore tracking the sensitivities requires only $\mathcal{O}(|\theta|)$ storage, while updating the Jacobian $d_\theta h_t$ gets reduced from a matrix multiplication to a dot product. For more details about the properties of the LRU and its online training algorithm the reader is encouraged to check the excellent work done by [27] and [37].

Having introduced the LRU and its benefits when used in combination with the RTRL algorithm, in the next section we will start studying its dynamics empirically in a teacher-student setting.

3.2.3 Sample Complexity Study

Our investigation begins with the question: Does the substantially higher update frequency of RTRL compared to BPTT provide any benefits in a Linear System Identification (LSI) task? Specifically, we aim to determine whether it results in improved performance or faster convergence.

Following [36], we will work under a teacher-student setting where the teacher is formally defined as:

$$h_{t+1} = Ah_t + Bx_{t+1} \quad \text{and} \quad y_t = Ch_t + Dx_t. \quad (3.10)$$

with $h_t \in \mathbb{R}^n$, $x_t \in \mathbb{R}$ drawn i.i.d. from $\mathcal{N}(0, 1)$, $A \in \mathbb{R}^{n \times n}$, $B \in \mathbb{R}^{n \times 1}$, $C \in \mathbb{R}^{1 \times n}$ and $D \in \mathbb{R}^{1 \times 1}$. The difficulty of the task is determined by the spectral radius of A , since given an input x_{t-j} its contribution to y_t is modulated by $\rho(A)^j$ as can be seen from equation 3.6. Thus bigger values of $\rho(A)$ will force the student to *remember* inputs from further back in time. Since this $\rho(A)$ is a parameter we want to control in our experiments, after sampling $A_{i,j} \stackrel{\text{iid}}{\sim} \mathcal{N}(0, 1/n)$, we complex diagonalize it⁶, and then we apply the following transformation [36] to its spectrum:

$$\lambda_i \leftarrow (\nu_{\min} + (\nu_{\max} - \nu_{\min}) \tanh(|\lambda_i|)) \exp\left(i \frac{\text{angle}(\lambda_i)}{\pi} \theta_0\right)$$

note that this transformation ensures that $\nu_{\min} \leq |\lambda_i| \leq \nu_{\max}$. In case A contained complex eigenvalues they must come in conjugate pairs. By using $\exp(\cdot)$ in the transformation and noting that $\exp(i\theta)^* = \exp(-i\theta)$, the resulting matrix after the transformation is also real.

Controlling $\nu_{\min}$ and $\nu_{\max}$ in the teacher is not enough for our purposes as we also need to control the spectrum of the student. The student will be a LRU, where its spectrum is controlled by sampling λ using lemma 3.2 from [27]:

⁶You can do this almost surely.

3. RTRL TRAINING DYNAMICS

Let u_1, u_2 be independent uniform random variables on the interval $[0, 1]$. Let $0 \leq r_{\min} \leq r_{\max} \leq 1$. Compute $\nu = -\frac{1}{2} \log(u_1 \rho + r_{\min}^2)$, where $\rho = r_{\max}^2 - r_{\min}^2$ and $\theta = 2\pi u_2$. Then $\exp(-\nu + i\theta)$ is uniformly distributed on the ring in $\mathbb{C}$ between circles of radii $r_{\min}$ and $r_{\max}$.

For the experiments the hidden size of the teacher is fixed to 10 and for the student 64. We will gradually increase $[r_{\min}, 1)$ and $[\nu_{\min}, 1)$ for the student and the teacher respectively. We choose a sequence size of $T = 300$ to ensure that the input sequence is longer than the characteristic timescale we want to learn⁷. For every value $\nu_{\min}$, we train the student LRU with BPTT and RTRL under the same conditions with the same budget of samples, which varies from $\{1000, 5000, 20000, 500000\}$. We work under the *purest* version of SGD with batch size 1. The loss $\mathcal{L}$ is the mean squared error over the prediction and the target sequence. To be fair with both algorithms and since gradient descent optimization is highly dependent on the learning rate, we sweep over 15 learning rates starting from 0.01 and decreasing logarithmically. For the optimization algorithm we use **Adam** [19] with its default hyperparameters in combination with cosine annealing without warm restarts⁸. When the best evaluation loss is found at one of the extremes of the allocated learning rates, we manually finetune it until the best learning rate does not live in the extremes. We repeat everything for 4 different seeds. The grid search is done using Ray Tune [21], and we use the framework introduced in the previous chapter for implementing RTRL, where we conducted unit tests in all the optimization algorithms to ensure their correctness. Details of all the hyperparameters that we sweep over can be found in table 3.1.

Intuitively to correctly explore the parameter space Θ for learning long-range dependencies, we want the $d_{\theta} h_t$ to not be biased. This is ensured in BPPT, as there are no in-sequence updates. However for RTRL this term does not actually reflect the influence of the parameters at the beginning of the sequence. Since θ_0 has had an influence in h_{t-k} and h_{t-k} in turn influences h_t as well, if the loss $\mathcal{L}$ at time step t is highly dependent in the state of h_{t-k} , we should take into account the θ_0 that leads h_0 to h_{t-k} as well. By allowing online updates we are not really tracking the real influence over all time steps. For tasks with short-time scale as measured by the spectral radius of the teacher, we should expect that the significantly higher number of updates of RTRL help converging faster towards a smaller loss. This advantage however should be lost as we give more sample budget both BPTT and RTRL, and the spectral radius of the teacher increases as well.

Results of the empirical sample complexity experiment for LSI can be seen

⁷Characteristic of the teacher is $\frac{1}{1-\rho(A)}$.

⁸Annealing the learning rate in some form is a necessary condition in SGD to converge towards a ball of small radius centered at a local minima.

Grid Search	
Algorithms	{RTRL, BPTT}
Samples	{1000, 5000, 20000, 50000}
Seeds	{7, 14, 21, 28}
Teacher h_{dim}	10
Student h_{dim}	64
Batch Size	1
η	$0.01 \times 2^{-1 * [0,1,\dots,15]}$
r_{min}	{0.32, 0.42, 0.78, 0.92, 0.99}
r_{max}	1
Scheduler	Cosine Annealing (No warm restarts)
β_1	0.9
β_2	0.99

Table 3.1: Experimental details of the grid search performed for the empirical sample complexity study.

in figure 3.1. The results align with our intuition as the performance of RTRL outperforms BPPT only on small values of ν_{min} and when the budget of samples is small. Eventually the differences vanish and BPTT catches up in performance. For bigger values of ν_{min} , RTRL does not present any advantage and when the budget is increased towards 50000 performs an order of magnitude worse than BPTT.

3.3 Benefits of Noisy In-Sequence Updates

Recall that in the previous experiment we used Adam with its default hyperparameters as our optimizer. Lets take a closer look at the update function defined by Adam⁹:

$$\begin{aligned}
g_t &\leftarrow \text{d}_\theta f_t(\theta_{t-1}) \\
m_t &\leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \\
v_t &\leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \\
\theta_t &\leftarrow \theta_{t-1} - \eta \cdot \frac{m_t}{\sqrt{v_t} + \varepsilon} \\
\theta_t &=: \mathcal{U}(\beta_1, \beta_2, \eta, \text{d}_\theta f_t(\theta_{t-1}))
\end{aligned}$$

Adam defines three¹⁰ hyperparameters β_1 , β_2 and η that are left for the experimenter to tune. Intuitively β_1 controls the exponential moving average

⁹For a general function f , and for simplicity we ignore the bias correction steps.

¹⁰The epsilon was left there for numerical stability but turns out that practitioners are also tuning it.

3. RTRL TRAINING DYNAMICS

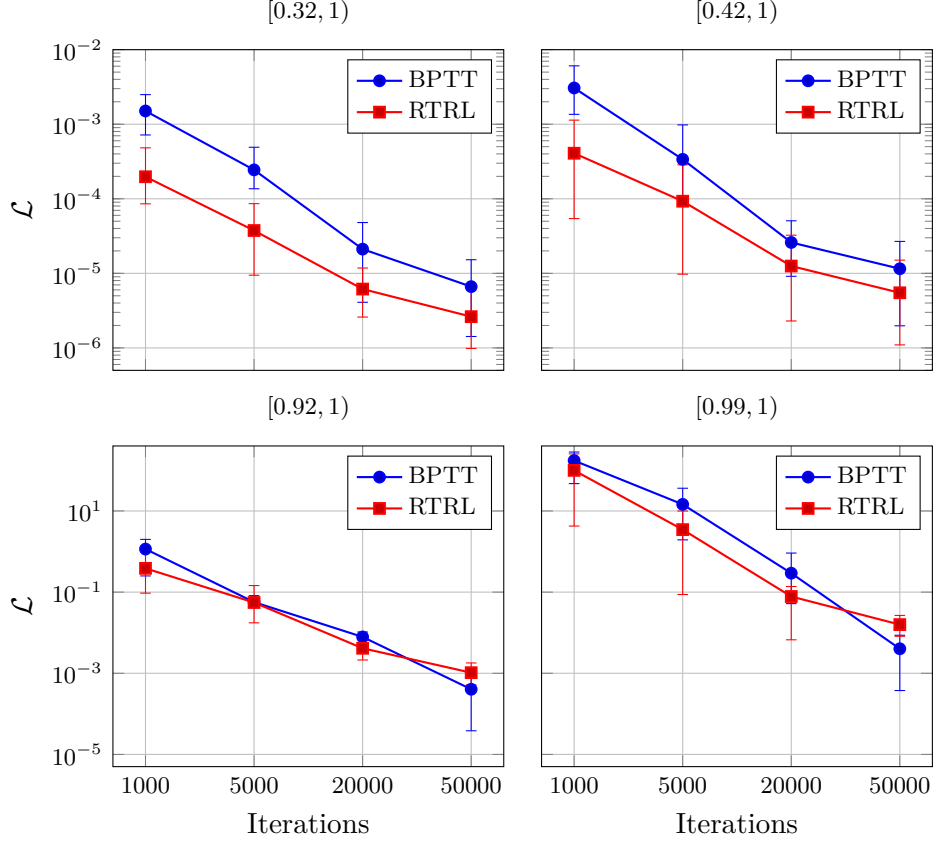


Figure 3.1: Sample complexity study for different values of $(\nu_{\min}, 1]$. We report the mean over all the seeds, where the error bars denote the min and max over the seeds. Both BPTT and RTRL are in their best performing setting, found with grid search.

of the gradients, since as $\beta_1 \rightarrow 1$ greater smoothing is performing and more weight is given to the average gradient m_t . One will expect that for learning a long-range task, if we are performing in-sequence updates, we want to give more importance to the average gradient m_t , the intuition for this is that since in the online setting the parameters now define a in-sequence trajectory $\theta_{1:T}$, the update directions from the instantaneous loss gradients $d_\theta \mathcal{L}_t$ do not necessarily need to be aligned¹¹. This leads the trajectory to fluctuate significantly, which we control using β_1 with Adam. The interpretation that β_1 controls these fluctuations of the gradients, can be easily checked by tracking the updates $\mathcal{U}(\beta_1, \eta, d_\lambda \mathcal{L}_t)$ ¹² given by Adam to the eigenvalues, as we do in Figure 3.2. Whether reducing this fluctuations is beneficial for RTRL, as we presuppose, is what we discuss next.

Contrary to our intuition the closer $\beta_1 \rightarrow 0$ the better RTRL performs, even

¹¹Neither orthogonal, fluctuation is what matters.

¹²We fixed the β_2 and not write it explicitly to simplify notation.

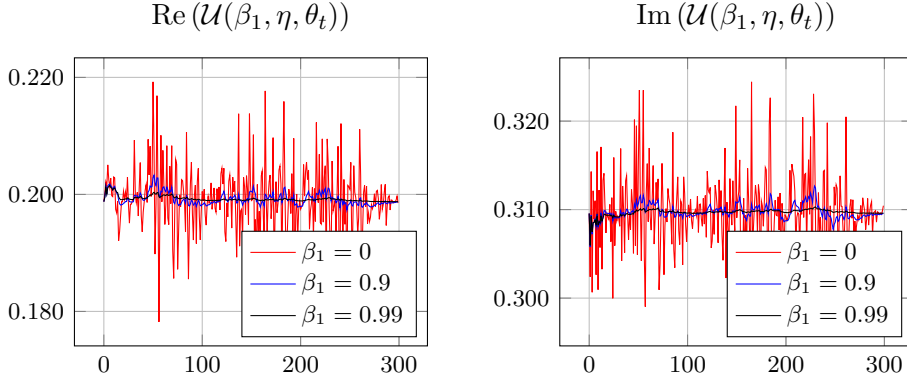


Figure 3.2: Toy example to confirm our intuition regarding the role of the β_1 in Adam for in-sequence updates. One dimensional teacher in $[0.99, 1)$, one dimensional student dimensional student in $[0, 1)$. One single iteration of RTRL over an input sequence.

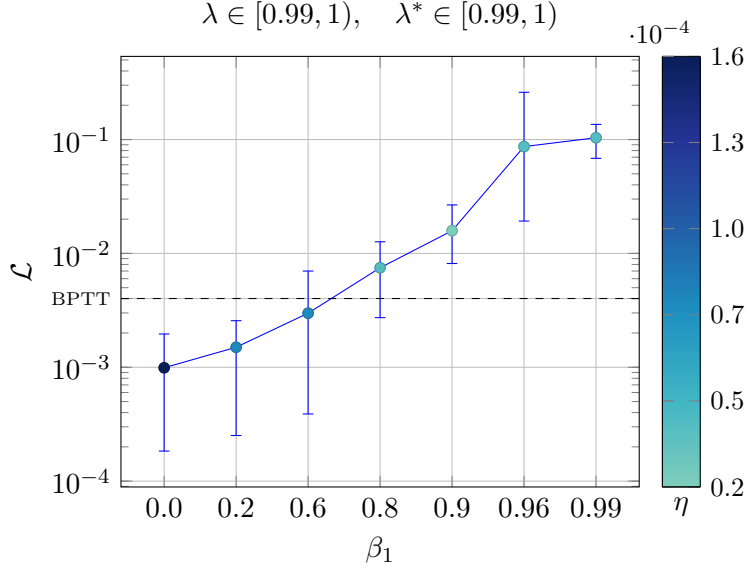


Figure 3.3: Performance of Adam in the LSI task for different values of β_1 . The colorbar reflects the value of the initial learning rate η . Cosine Annealing is applied for every run. Mean of 4 different seeds, errors bars indicating the min and max evaluation loss.

in the harder task of our previous experiment where $\nu_{\min} = 0.99$. In figure 3.3, we can see a sweep over different values of β_1 , where for each β_1 , we follow a similar grid search procedure as in the previous study, ensuring that for every β_1 we find the best configuration, ensuring a fair comparison. The best mean performance is achieved with $\beta_1 = 0.0$ i.e no smoothing at all over the in-sequence gradients, and even more surprising, it is achieved with the maximum initial learning rate η when compared to best performing learning rates in the other β_1 scenarios.

In order to get more insights we start by analyzing the backward sensitivity of the system, which we define as:

$$t \mapsto \left\| \frac{dh_T}{dh_t} \right\| \quad (3.11)$$

where the norm is the spectral norm i.e the largest singular value. Intuitively this term measures how sensitive the final hidden state h_T is to a perturbation of a hidden state in a distant past t . For tasks with long-range dependencies we want this term to increase during training in case it is small at time steps the task cares about.

Note that in the LRU, the state-transition Jacobian is constant $d_{h_{t-1}}h_t = \text{diag}(\lambda)$, and because the spectral norm of a diagonal matrix is equal to its spectral radius we have that $t \mapsto |\lambda_{\max}|^{T-t}$. So function 3.11 is completely characterized by the spectral radius of $\text{diag}(\lambda)$, which allows us to study it with just a single scalar. Therefore we track the max eigenvalue after every iteration of both RTRL and BPTT, as well as tracking the min eigenvalue and then mean of $|\lambda_j|$. We do this in a setup where the student starts with a spectrum in $[0, 1)$, so that we recover Glorot initialization [13]. The teacher will be set to the hardest difficulty where $\nu_{\min} = 0.99$, in addition to $|\lambda_{\max}|$. For both algorithms we follow the same grid search procedure as in the previous study in order to compare them in their best performing settings. Since we want to better understand the observation from figure 3.3, in RTRL we check for $\beta_1 = 0$ and $\beta_1 = 0.9$, whereas in BPTT we only check for $\beta_1 = 0.9$. Note that in the class of Linear RNNs, as mentioned before, the state-transition Jacobian is constant¹³ and is equal to the connectivity matrix, therefore we should expect in both settings that the spectral radius of $\text{diag}(\lambda)$ increases towards higher values, closer to where the spectral radius of the teacher is. Otherwise they could not learn the long-range dependencies.

In figure 3.4 we can see the evolution of the quantities just described. From the plots we note three qualitative differences when considering noisy in-sequence updates ($\beta_1 = 0.0$) in comparison to BPTT and smoother in-sequence updates ($\beta_1 = 0.9$) in RTRL:

1. The spectral radius converges faster towards its theoretical limit¹⁴.
2. The norm of the minimum eigenvalue converges faster towards 0.
3. Overall the norm of the eigenvalues of the state-transition Jacobian tend to converge faster towards higher values.

in order to better comprehend this empirical observations, in the next subsection we will argue for a simple toy model that we believe explains the previous qualitative differences.

¹³Not while training in RTRL, only when doing inference.

¹⁴Recall that in the LRU the spectral radius is upper bounded by 1.

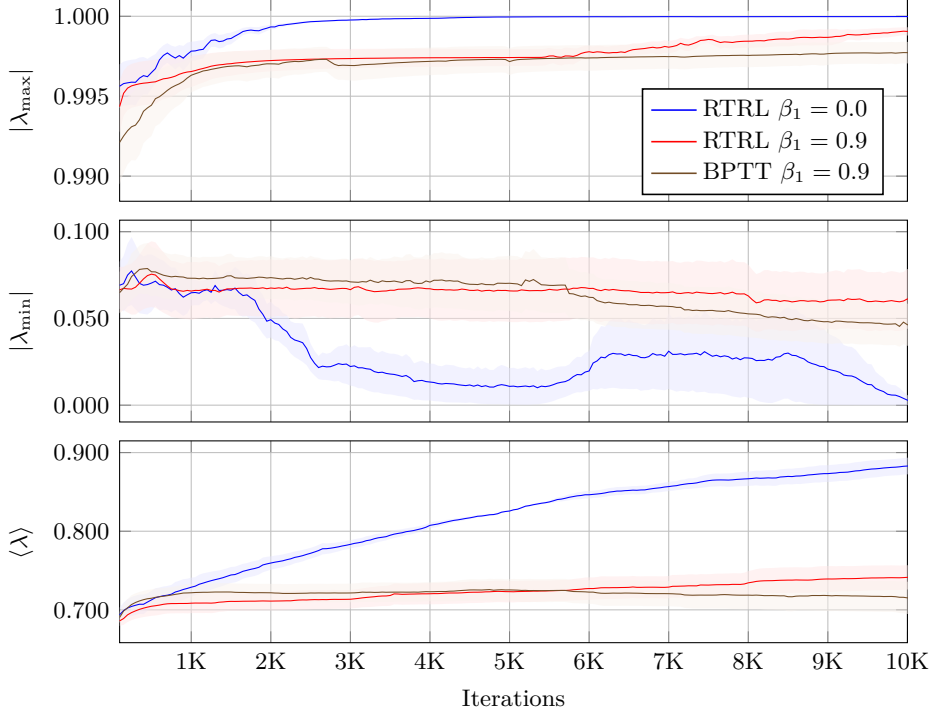


Figure 3.4: Evolution of the norm of max, min and average eigenvalue during the first 10K iterations. Student initialized in $[0.0, 1)$. Teacher initialized with in $[0.99, 1)$. Shaded areas indicate the standard error of the mean over the different seeds.

3.3.1 RTRL Toy Model

To understand the qualitative phenomena of the previous study we start by recalling the LRU parametrization¹⁵:

$$\lambda := \exp(-\exp(\nu) + i\theta)$$

where $\nu \in R^n$ and $\theta \in R^n$. Since the quantity we are interested is the norm of λ_j , to simplify the analysis we set $\theta = \vec{0}$ as it does not contribute¹⁶ to the norm. Therefore the parametrization we study is left as:

$$\lambda := \exp(-\exp(\nu)) \quad (3.12)$$

and the in-sequence update equations¹⁷ when performing gradient descent follow:

$$\nu_{t+1} = \nu_t - \eta \frac{dL_t}{dh_t} \frac{dh_t}{d\lambda} \frac{d\lambda}{d\nu} \quad (3.13)$$

$$\lambda_{t+1} = \exp(-\exp(\nu_{t+1})) \quad (3.14)$$

¹⁵Here exp denotes element-wise exponential.

¹⁶Recall that by Euler's identity $|\exp(a + ib)| = |\exp(a)| |\exp(ib)| = \|\exp(a)\|$.

¹⁷We drop the subscript j to simplify notation.

3. RTRL TRAINING DYNAMICS

Define $e_t := \eta \frac{dL_t}{dh_t} \frac{dh_t}{d\lambda}$, and note that $-\frac{d\lambda}{d\nu} = \exp(\nu_t - \exp(\nu_t))$, hereby equation 3.13 becomes:

$$\begin{aligned}\nu_{t+1} &= \nu_t + e_t \exp(\nu_t - \exp(\nu_t)) \\ \nu_{t+1} &=: g(\nu_t, e_t)\end{aligned}\tag{3.15}$$

we can treat 3.15 as a non-autonomous discrete dynamical system, since e_t will be varying at each time step t . Even though an oversimplification, is still worth assuming constant e_t , so that we can treat 3.15 as an autonomous dynamical system, where the analysis becomes easier. In the one-dimensional setting the direction of e_t is either a positive one (+1) or negative (-1), so assuming $|e_k| = 1$, we can plot the dynamics of the system in a Cobweb diagram (Figure 3.5) together with the LRU parametrization. From the plots it is clear that:

$$\begin{aligned}\lim_{t \rightarrow \infty} g(\nu_t, +1) = +\infty &\implies \lim_{t \rightarrow \infty} \lambda_t = 0 \\ \lim_{t \rightarrow \infty} g(\nu_t, -1) = -\infty &\implies \lim_{t \rightarrow \infty} \lambda_t = 1\end{aligned}$$

Of course during optimization neither e_t will be constant nor $|e_t| \neq 1$. The term e_t will be regulated by the learning rate η , and the directions will differ during the in-sequence updates. In RTRL e_t fluctuates significantly during the forward pass (Figure 3.2), and this fluctuation can be controlled by β_1 when using Adam.

Assuming, as previously argued, that e_t fluctuates significantly, let's replace e_t by a random variable $\varepsilon_t \sim \mathcal{N}(0, \sigma^2)$. The fluctuation is now governed by σ . Consider a Markovian random walk in $\mathbb{R}$. With sufficient iterations, the larger the σ the quicker the random walk reaches its first-passage time for a given threshold. Recalling that in RTRL we perform T additional updates to λ compared to BPTT, the previous analysis suggests that under in-sequence updates, convergence towards either 0 or 1 occurs more rapidly in the LRU, even if these updates exhibit considerable fluctuation. In fact larger fluctuations, as in the random walk, imply earlier first-passage times over values of ν_t that result in closer values of λ_t to the two poles. We can see now that as $\beta_1 \rightarrow 0$, the effective σ increases, leading to a faster increase in the spectral radius while training, thereby enhancing performance in the harder task where $\nu_{\min} = 0.99$, which is what we see in figure 3.4.

In figure 3.6 we confirm our analysis, where we sample a white noise sequence from $\mathcal{N}(0, \sigma^2)$ to play the role of e_t and we track the evolution of $g(\nu_t, e_t)$, where $\lambda_0 = 0.5$, for different values of σ . As we can appreciate from the figure, the largest value of σ results in faster convergence towards the two poles, 0 and 1, but eventually for every value of σ convergence to the poles should occur given enough iterations. We repeat the analysis for 10^3 white noise sequences, each of them of length $T = 10^4$, and in figure 3.6 we show their distribution at

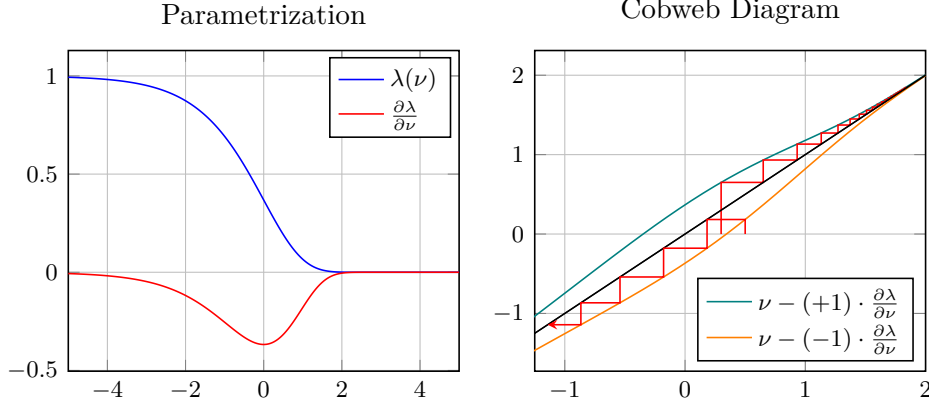


Figure 3.5: **Left:** Parametrization of the LRU for real eigenvalues. **Right:** Cobweb Diagram assuming constant backpropagated errors, (+1) or (-1)

different time steps, for different values of σ . The higher density around 0 in the histogram at 10K can be explained by noting that:

$$\left| \frac{d_\nu \lambda(\nu)}{d_\nu \lambda(-\nu)} \right| = \frac{\exp(\nu - \exp(\nu))}{\exp(-\nu - \exp(-\nu))} = \Omega(\exp(-\nu))$$

which implies that converge happens in fewer iterations towards 0 than towards 1, biasing the distribution in this simplified RTRL toy model for the LRU.

From the previous analysis we can already start to understand the qualitative differences we observed in the LSI study between the different β_1 scenarios and BPTT. If $\beta_1 \rightarrow 0$ the $|\lambda_j|$ of the LRU should show convergence towards a bimodal distribution in 0 and 1, which in turn will increase the spectral radius. This is what we indeed observe in figure 3.7, when we track the trajectories of the $|\lambda_j|$ during training of the LSI task. Only one seed is displayed for making the visualization easier, nevertheless to strengthen our hypothesis about the qualitative differences between RTRL and BPTT, in figure 3.8 we plot the complementary cumulative distribution function¹⁸ $\Pr(|\lambda| > x)$ (CCDF) at the end of training, now computed among the different seeds for both RTRL and BPTT.

Note that the $1 - F(x)$ displayed in figure 3.8 is computed at the last iteration of training, nevertheless given the big gap between BPTT in RTRL it suggests the following. For sufficiently big N , there exists a threshold α such that for all $n \geq N$:

$$1 - F_n^B(x) \leq 1 - F_n^R(x) \quad \text{for } \alpha \leq x \leq 1. \quad (3.16)$$

where with B we denote BPTT and with R we denote RTRL. We cannot offer a proof for this, but nevertheless, we give support for this claim in figure

¹⁸Actually the empirical CCDF i.e the ECCDF

3. RTRL TRAINING DYNAMICS

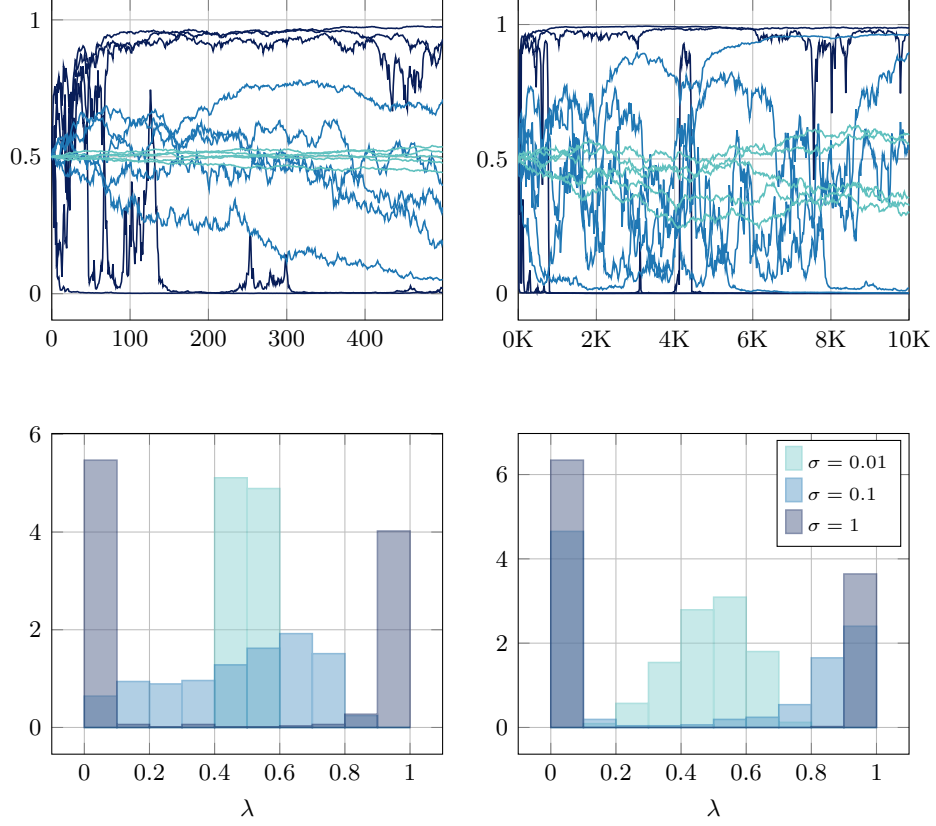


Figure 3.6: Top: Sampled trajectories of the toy RTRL model, for different standard deviations for $T \in \{500, 10^4\}$ steps. **Bottom:** Distribution at $T \in \{500, 10^4\}$ of $N = 1000$ sampled trajectories for different standard deviations

3.9, where we show CCDF for both methods while training from the first iteration until iteration 1000, the upper bound on 1 its given by the stable parametrization of the LRU. How small can this N be, we believe will become relevant when we study non-linear RNN’s in the next section.

Interpreting e_t as ε_t was partly motivated by figure 3.2, where we verified, albeit in a simplistic setting, that the updates provided by Adam i.e $\mathcal{U}(\beta_1, \eta, d_\nu \mathcal{L}_t)$, were highly stochastic in nature. The cautious reader must have noted that our definition of e_t in the toy model, does not equate exactly to our LSI experiments are being performed. To see why note that when using Adam, the actual update equations for ν follow $\nu_{t+1} = \nu_t + \mathcal{U}(\beta_1, \beta_2, \eta, d_\theta \mathcal{L}_t)$ therefore:

$$\nu_t + \mathcal{U}(\beta_1, \beta_2, \eta, d_\theta \mathcal{L}_t) \neq \nu_t - \eta \frac{dL_t}{dh_t} \frac{dh_t}{d\lambda} \frac{d\lambda}{d\nu} = \nu_t - e_t \frac{d\lambda}{d\nu} \quad (3.17)$$

to further strength our observations and justify our interpretation of e_t we

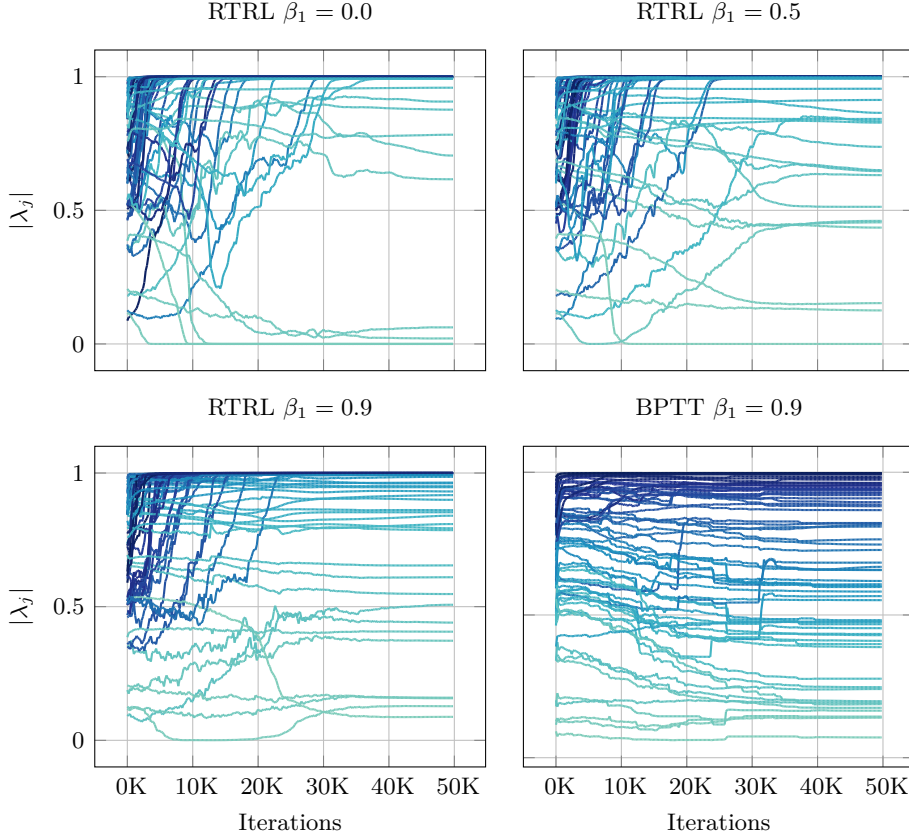


Figure 3.7: Trajectories of the $|\lambda_j|$ during training in RTRL for different values of β_1 and in BPTT. Coloring reflects the order of $|\lambda_j|$, darker being the highest, lighter being the lowest. Only one seed is showed for easier visualization

need to decompose $\mathcal{U}(\cdot)$ into the two terms, e'_t and $d_\nu \lambda$ so that:

$$\mathcal{U}(\beta_1, \beta_2, \eta, d_\nu \mathcal{L}_t) = e'_t \frac{d\lambda}{d\nu} \quad (3.18)$$

which matches the same equation as in 3.15, note that in the LRU ν is a vector in $\mathbb{C}^k$ and λ as well, therefore to solve for e'_t , we could right multiply both sides by the inverse matrix $(d_\nu \lambda)^{-1}$. The inverse is guaranteed to exist since the matrix $d_\nu \lambda$ is diagonal¹⁹. If our interpretation of RTRL and the influence of β_1 is a valid one, then we must see as well that when computing the element-wise $\text{Var}[e'_n]$ (using as observations all the time steps during an input sequence $\mathbf{x}$ given at iteration n)²⁰, for different values of β_1 the variances must satisfy²¹:

$$\text{Var}_{0.99}[e'_n] \leq \text{Var}_{0.9}[e'_n] \leq \text{Var}_{0.8}[e'_n] \cdots \leq \text{Var}_{0.0}[e'_n]$$

¹⁹Unless there is a zero element in its diagonal, to be in the safe side and avoid division zero we could add an epsilon term

²⁰We ignore the time dependence t since in our model we sample iid noise.

²¹The subscript denotes which value we used for β_1 .

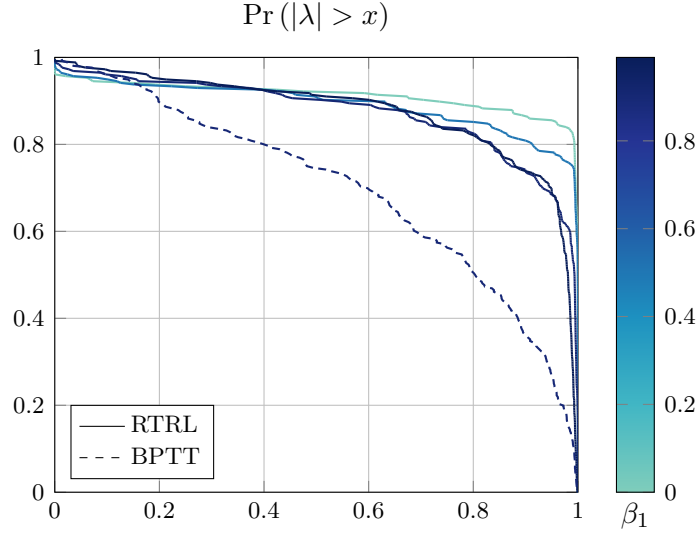


Figure 3.8: $\Pr(|\lambda| > x)$ after 50K Iterations of training. Computed over 4 different seeds for different values of β_1 and for BPTT.

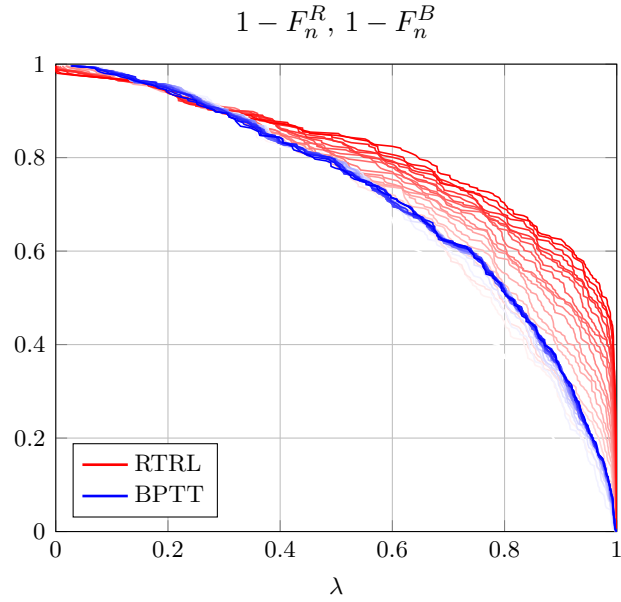


Figure 3.9: In Red: $1 - F_n^R$ for $n \leq 1000$. In Blue: $1 - F_n^B$ for $n \leq 1000$. $R := \text{RTRL}$ and $B := \text{BPTT}$. Runs over 4 seeds. Lighter color indicates earlier iterations, darker color indicates iterations at the end.

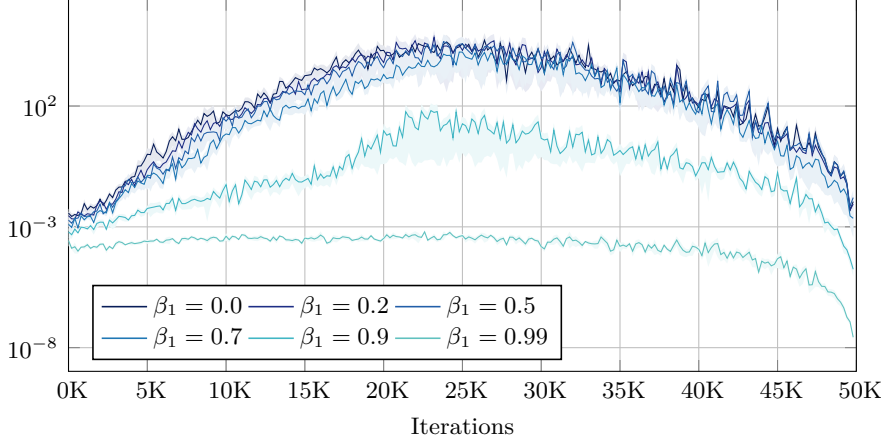


Figure 3.10: Mean of the vector $\sqrt{\text{Var}[e']_n}$ during training for different values of β_1 .

Recall that e'_t is a vector with elements in $\mathbb{C}$ therefore we use the following definition of variance for a complex random variable:

$$\text{Var}[X] = \text{Var}[\text{Re}(X)] + \text{Var}[\text{Im}(X)] \quad (3.19)$$

In figure 3.10 we can see these values computed using 4 different seeds where we used the same initial learning rate η . Note that they closely match our interpretation of e_t .

Overall the previous experiments further validate the use of our toy model for RTRL, when trying to study its dynamics. Though it may sound simple, to the best of our knowledge we have not seen any study in the literature following our approach. This might partially be because of timing, since the LRU which is a key component in our study has only been introduced recently. We conclude this section by reminding the reader that we started studying RTRL in class of linear RNNs, with the hope of giving us a better understanding of its training dynamics in the non-linear setting. In the next section we start to study RTRL in non-linear RNNs.

3.4 Implications In Sparse Networks

Recently [24] showed the advantage of imposing sparsity in the parameters of the network by proposing a biased approximation to $d_\theta h_t$ which reduces the memory and computational complexity of RTRL, as explained in the preceding chapter. Their experiments conclude that training highly sparse RNNs (GRU and LSTM) with RTRL outperforms BPTT when learning a curricular copy task. Studying RTRL in RNNs with gated mechanisms to understand this phenomena. Is out of the scope of this thesis and left for future work, here instead we focus on Continuous Time Recurrent Neural Networks (CTRNN), using some of the insights derived from the previous section in Linear RNNs.

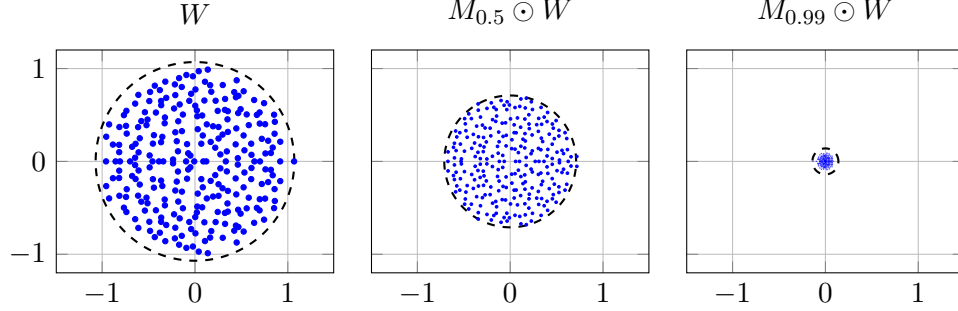


Figure 3.11: Distribution of the eigenvalues for a random matrix with i.i.d entries from $\mathcal{N}(0, 1/256)$, when applying masks of different levels of sparsity.

Following [30] the class of CTRNN considered here are derived from the discretization of the following continuous-time dynamical system:

$$\dot{h}(t) = -h(t) + W\varphi(h(t)) + \sqrt{N} \sum_{i=1}^{N_{\text{in}}} \mathbf{u}_i x_i(t). \quad (3.20)$$

where $\varphi := \tanh$, N represents the number of hidden units, N_{in} the number of input channels, $W \in \mathbb{R}^{N \times N}$, and $U = [\mathbf{u}_1, \dots, \mathbf{u}_{N_{\text{in}}}] \in \mathbb{R}^{N \times N_{\text{in}}}$. The readouts of the network are computed by:

$$y_i(t) = \frac{\mathbf{c}_i^T \varphi(h(t))}{\sqrt{N}} \quad \text{For } i \text{ in } \{1, \dots, N_{\text{out}}\} \quad (3.21)$$

where N_{out} represents the number of output channels and $\mathbf{c}_i \in \mathbb{R}^N$. Similar to [30] we are interested in the training dynamics of networks where $N \gg 1$ and the dimensions of N_{in} , and N_{out} are small. However we depart from their setup by forcing sparsity in the connectivity matrix W in an attempt to understand the results from [24]. We do this by sampling a random binary matrix M_s with i.i.d entries $M_{i,j} \sim \text{Bernoulli}(1-s)$, where s will control the amount of sparsity imposed in the connectivity matrix. The transformation to W is simply:

$$W \mapsto M_s \odot W \quad (3.22)$$

The first insight to understand [24] comes from the following fact. Define $W' := M_s \odot W$, if the entries $W_{i,j} \stackrel{\text{iid}}{\sim} \mathcal{N}(0, 1/N)$ then:

$$\mathbb{E}[W'] = \mathbb{E}[(M_{i,j}W_{i,j})^2] = \mathbb{E}[(M_{i,j})^2] \mathbb{E}[(W_{i,j})^2] = \frac{(1-s)}{N}$$

since $\mathbb{E}[W'] = 0$ and $E[|W'|^4] \leq C$, then²² by Bai and Yin [2], we have that:

$$\Pr \left(\limsup_{N \rightarrow \infty} \rho(W') \leq \sqrt{(1-s)} \right) = 1 \implies \mathbb{E} \left[\limsup_{N \rightarrow \infty} \rho(W') \right] \leq \sqrt{1-s}$$

²²A Normal distribution has finite 4-th moment.

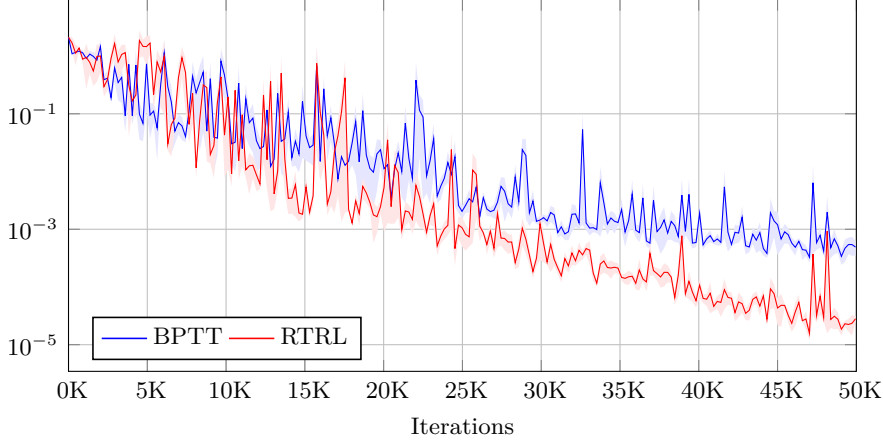


Figure 3.12: Mean Squared Loss curves RTRL and BPTT in the evaluation distribution. Average over 4 different seeds. Both RTRL and BPTT in their best performing setting after running a comprehensive grid search. Shaded regions represent the standard error of the mean of the quantity of interest.

and by reverse Fatou’s Lemma:

$$\limsup_{N \rightarrow \infty} \mathbb{E} [\rho(W')] \leq \sqrt{1-s} \quad (3.23)$$

which basically tells us that for $N \gg 1$ (as in our setting and the one in [24]), under Glorot initialization of the connectivity matrix W , imposing a random mask on the weights to induce sparsity, effectively decreases its spectral radius. This behavior is illustrated in Figure 3.11. In the regime of linear RNNs, as discussed in the previous section, RTRL should escape this small spectral radius faster than BPTT. As a result, this could potentially enhance performance when the objective is to learn long-range dependencies, which is what [24] observed in their experiments although in highly sparse gated RNN’s.

To test the hypothesis that RTRL outperforms BPTT when training highly sparse CTRNN we will follow a similar approach as in the preceding section were now we substitute the LSI task, with the Romo [29] task. The task is a delayed-discrimination task, where the network is presented with two positive pulses of different amplitudes separated by a delay. Following an additional delay, the network is required to remember which of the two pulses had greater amplitude. We choose this task as it is a simple one to start studying RTRL and we can easily control the specific time steps t at which the network has received its inputs and has to start *recalling* the largest pulse.

As in all of our previous experiments to compare BPTT and RTRL in their best performing settings, we follow a similar grid search as in the sample complexity study. For RTRL we set $\beta_1 = 0$ in light of the previous section and for BPTT we set $\beta_1 = 0.9$. The sparsity of the network is imposed by

3. RTRL TRAINING DYNAMICS

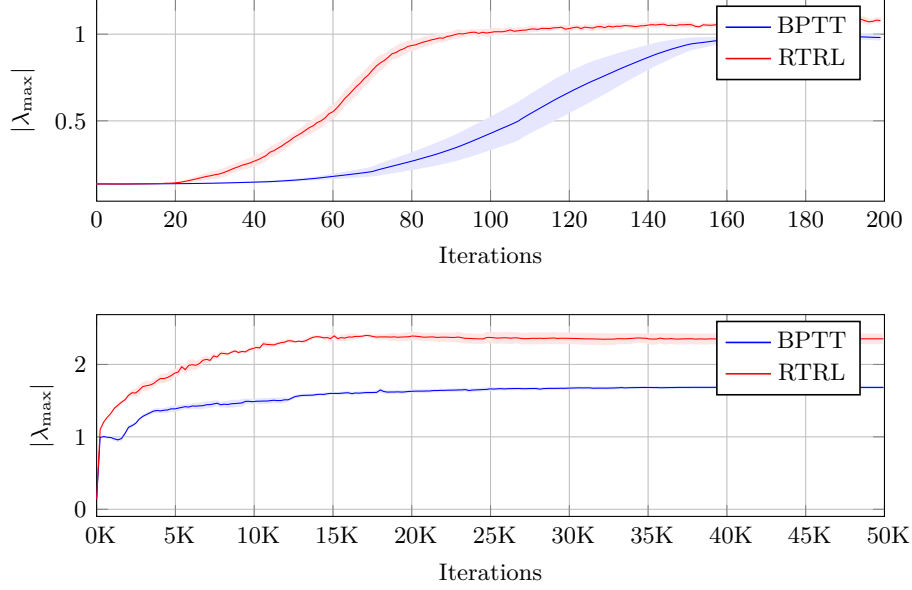


Figure 3.13: Evolution of the spectral radius $\rho(W')$. Average over 4 different seeds. Both RTRL and BPTT in their best performing setting after running a comprehensive grid search. Shaded regions represent the standard error of the mean of the quantity of interest. **Top:** Evolution during the first 200 iterations. **Bottom:** Evolution for the full training loop.

sampling a random binary mask as explained before were we set s to **0.98**. For the size of the hidden units we choose $N = 256$ following [30] and also to obtain the upper bound in 3.23. For discretizing 3.20 we choose as $dt = 0.3$. The evaluation loss is the mean squared error over the readouts of the network. The number of iterations for both RTRL and BPTT is set 50K. Similar to the LSI task we set the batch size to 1. In figure 3.12 we can see the evaluation loss curves for both methods in their best performint setting. Clearly RTRL outperforms BPTT. This partial result strengthen our hypothesis about the advantages of RTRL in highly sparse CTRNNs. In figure 3.13 we can also see how the spectral radius converges substantially faster to a bigger value in RTRL²³. We have run control experiments where after sampling the random sparse connectivity matrix, we scale it so that we start in both algorithms with $\rho(W') = 1$, and the result in terms of performance were almost the same, with RTRL clearly outperforming BPTT. Therefore there has to be some other qualitative phenomena explaining the gap in performance.

Note that in the non-linear regime, the spectral radius is not a good indicative anymore of how the network is learning, in the case of the LRU we studied that term since as commented before, the backward sensitivity function can be fully characterized by a single scalar. This is not true in the case of non-linear RNN's. The state-transition Jacobian of CTRNN after discretization is not

²³Since we are not using anymore the LRU, this term is not bounded anymore by 1.

constant anymore and follows:

$$J(t) = \frac{\partial h_t}{\partial h_{t-1}} = \text{diag}(1 - dt) + dtW \text{diag}\left(\left(1 - \varphi(h_{t-1})^2\right)\right) \quad (3.24)$$

In an attempt to understand the difference in performance between both methods we recall the following standard result in the theory of continuous-time dynamical systems. Assume we have an ODE, defined as:

$$\dot{h}(t) = f(h(t)) \quad (3.25)$$

where $h \in \mathbb{R}^N$ and represents the hidden states of a non-linear RNN. We can differentiate equation 3.25 from both sides with respect to h_0 and after an exchange of derivatives²⁴ we get the following equation:

$$\frac{d}{dt} \left(\frac{\partial h_t}{\partial h_0} \right) = J(t) \frac{\partial h_t}{\partial h_0} \quad (3.26)$$

note that $\delta h_t / \delta h_0$ its closely related to the function 3.11 we were studying in the previous section, were instead the role of the time variable t is inverted. Intuitively $\delta h_t / \delta h_0$ measures how sensitive is h_t with respect to a perturbation to the initial condition h_0 . Following [8] lets define as well $A(t) := \delta h_t / \delta h_0$, equation 3.26, can be rewritten then as an ODE following:

$$\frac{dA(t)}{dt} = J(t) A(t) \quad (3.27)$$

where $A(0) = I$. Similar to [8] we are going to make the assumption that $J(t)$ is constant, although a **strong** assumption its going to help us understand the qualitative differences of RTRL when compared to BPTT, at least in the Romo task. The **strong** assumption, thus allows us to get the solution for the ODE at 3.27:

$$A(t) = e^{Jt} \quad (3.28)$$

assuming that J is diagonalizable we then get:

$$A(t) = e^{Jt} = P e^{\Lambda(J)t} P^{-1} \quad (3.29)$$

where $\Lambda(J)$ denotes the eigenvalues of J . Using the fact that the trace bounds the spectral norm, we have then:

$$\left| \text{tr} e^{\Lambda(J)t} \right| \leq N \|A(t)\| \quad (3.30)$$

combining this lower bound, with the results from figure 3.9 (where we showed empirically that early in training of a LRU, RTRL has much higher probability

²⁴Exchanging derivatives in this case can be proved to be correct.

3. RTRL TRAINING DYNAMICS

than BPTT of $P(|\lambda| > x)$ for bigger values of x), we can see that RTRL also early in training will increase the trace in 3.30 since:

$$\left| \text{tr } e^{\Lambda(J)t} \right| = \left| \sum_{i=0}^N e^{\lambda_i t} \right| \quad (3.31)$$

and as t grows this term will be dominated by the λ_j with highest absolute value, which is precisely what RTRL increases from what we observed in the previous experiments. Therefore RTRL will push the lower bound in 3.30 to higher values, biasing as well the spectral norm of the forward sensitivity towards higher values when compared to BPTT early in training, which is needed to in order to start learning distant dependencies in a sequence [28]. Note that we have made strong assumptions in the previous derivation, which don't necessarily have to match the real experimental setting. In what follows we study what we have just argued.

In light of the previous experiments and from 3.23 we hypothesize that RTRL bias Highly Sparse CTRNNs to two regimes during training, the first one, suggested by 3.30, where the network tries to *memorize* as much as possible, and then learns to *forget* the irrelevant inputs for the task at hand. Actually we hypothesize that both BPTT and RTRL use the same strategy since the spectral radius of the network at initialization its quite small, but RTRL will enter the regime of *memorizing* much faster.

To study this hypothesis we follow a similar approach as in the previous section. As a reminder, we want to analyze:

$$t \mapsto \left\| \frac{dh_T}{dh_t} \right\| \quad (3.32)$$

For every new input sequence this function changes, therefore we are not interested in the absolute magnitudes of 3.32 within a sequence, we are interested in their relative values within a sequence. For example, we could be interested in comparing the value at a fixed time step t but at different iterations of the training algorithm say $n_1 < n_2$. At iteration n_1 the $d_{h_t} h_T^{n_1}$ could be much smaller in norm than $d_{h_t} h_T^{n_2}$ at n_2 . However that does not indicate that network its less sensitive to a perturbation in h_t at iteration n_1 , as it could be that the $d_{h_t} h_T^{n_1}$ its the biggest value in norm within its sequence indicating the contrary. Therefore if not normalizing this quantity we cannot compare them in fair manner during training at different iterations. For this reason the actual term we track is the following one. Define $c_n := \max_{0 \leq t \leq T} \left\| \frac{\partial h_T^n}{\partial h_t} \right\|$ for iteration n of the algorithm, the quantity we track can be defined then as:

$$t \mapsto \left\| \frac{\partial h_T^n}{\partial h_t} \right\| c_n^{-1} \quad (3.33)$$

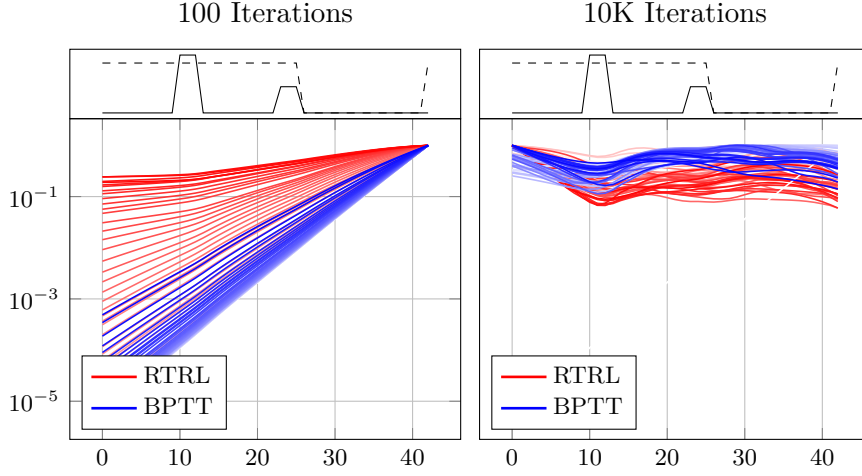


Figure 3.14: **Left:** Normalized $\delta h_T / \delta h_t$ for the first 100 iterations of the algorithm. **Right:** Normalized $\delta h_T / \delta h_t$ for the first 10000 iterations of the algorithm. Lighter color indicate earlier iterations, darker colors indicate iterations close to the end. Only one sample, and only seed is displayed for easier visualization.

this quantity is bounded in $[0, 1]$ therefore we could also integrate it, and track as well:

$$n \mapsto \int_0^T \left\| \frac{\partial h_T^n}{\partial h_t} \right\| c_n^{-1} dt \quad (3.34)$$

so that we can display in a simple way the evolution of the overall backward sensitivity of the algorithms. In figure 3.14 we can see 3.33 for a single seed and for only one input sequence at different number of iterations during training, already supporting our intuition about RTRL increasing norm of the backward sensitivity faster early in training. In figure 3.15 we can see 3.34, which gives even more support to the hypothesis that RTRL converges faster to a bigger value of 3.34, and then decreases this term faster as well compared to BPTT. We find interesting that even though their respective evaluation losses follow similar trends at the beginning of training, there exists a qualitative difference hidden in the dynamics of 3.34.

We conclude this short section and the chapter by remarking that our study although small in nature, offers an possible explanation for the results obtained in [24]. RTRL benefits from highly stochastic in-sequence updates, and it benefits even more when the network is highly sparse, as we showed experimentally and motivated theoretically. We also posit a potential negative consequence: the tendency of RTRL to push the spectral norm of the backward sensitivity towards higher values may impair its performance in denoising tasks. Gated RNNs mitigate this issue through their gating mechanisms. Future work will explore this potential phenomenon in greater detail and attempt to generalize our findings to other tasks inspired by the neuroscience literature. The research in [24] play a significant role in the development for this chapter as we were

3. RTRL TRAINING DYNAMICS

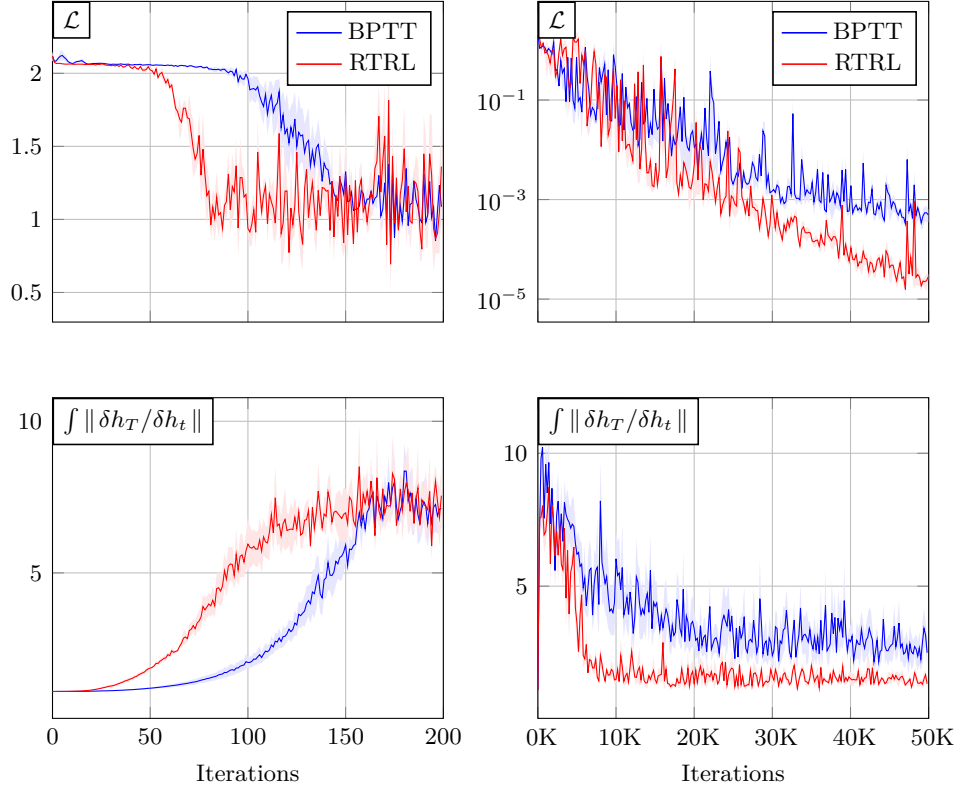


Figure 3.15: Top Row: Evaluation loss curves. For 200 and 50K Iterations. **Bottom Row:** Integral of the normalized backward sensitivity for 200 and 50K Iterations. Shaded areas indicate the standard error of the mean for the quantity of interested. Average over 4 different seeds. Both RTRL and BPTT in their best performing settings, found by a comprehensive grid search.

quite intrigued by their results in networks where we the sparsity was close to 98%

Chapter 4

Conclusions

This thesis investigated the dynamics and engineering challenges of RTRL within sparse RNNs. The core contributions of this research are twofold: the development of highly optimized implementations for existing algorithms and the detailed exploration of RTRL’s training dynamics through theoretical and empirical analyses.

Our main technical achievement is the optimized implementation of the Sparse 1-Step Approximation algorithm, originally proposed [24]. By creating and fine-tuning optimized kernels for sparse Jacobian computations and masked matrix multiplications, we achieved substantial performance improvements. Our benchmarks demonstrated a 64x speedup in wall-clock time compared to existing implementations, validating the practical applicability of our approach.

Chapter 3 delved into the training dynamics of RTRL, contrasting it with BPTT. Through rigorous experiments and analysis first in the setting of linear RNNs and benefiting from the stable parametrization of the LRU, we observed that RTRL benefits from highly stochastic in-sequence updates. This stochasticity aids in faster convergence in highly sparse CTRNNs as we motivated theoretically and showcased experimentally, offering a possible explanation for the results observed in [24].

Bibliography

- [1] L. F. Abbott. “Decoding neuronal firing and modelling neural networks”. In: *Quarterly Reviews of Biophysics* 27.3 (1994), pp. 291–331. DOI: [10.1017/S0033583500003024](https://doi.org/10.1017/S0033583500003024).
- [2] Z. D. Bai and Y. Q. Yin. “Limiting behavior of the norm of products of random matrices and two problems of Geman-Hwang”. In: *Probability Theory and Related Fields* 73.4 (1986), pp. 555–569.
- [3] Frederik Benzing et al. “Optimal Kronecker-Sum Approximation of Real Time Recurrent Learning”. In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, Sept. 2019, pp. 604–613. URL: <https://proceedings.mlr.press/v97/benzing19a.html>.
- [4] Guy E. Blelloch. *Prefix Sums and Their Applications*. Tech. rep. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University, Nov. 1990.
- [5] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: <http://github.com/google/jax>.
- [6] Benjamin Brock et al. “The graphblas c api specification”. In: *GraphBLAS.org, Tech. Rep* (2021).
- [7] Aydin Buluç and John R. Gilbert. “Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments”. In: *SIAM Journal on Scientific Computing* 34.4 (2012), pp. C170–C191. DOI: [10.1137/110848244](https://doi.org/10.1137/110848244). eprint: <https://doi.org/10.1137/110848244>. URL: <https://doi.org/10.1137/110848244>.
- [8] Bo Chang et al. “AntisymmetricRNN: A Dynamical System View on Recurrent Neural Networks”. In: *International Conference on Learning Representations*. 2019. URL: <https://openreview.net/forum?id=ryxepo0cFX>.

- [9] Thomas F. Coleman and Jorge J. Moré. “Estimation of Sparse Jacobian Matrices and Graph Coloring Blems”. In: *SIAM Journal on Numerical Analysis* 20.1 (1983), pp. 187–209. DOI: [10.1137/0720013](https://doi.org/10.1137/0720013). eprint: <https://doi.org/10.1137/0720013>. URL: <https://doi.org/10.1137/0720013>.
- [10] Francis Crick. “The recent excitement about neural networks”. In: *Nature* 337.6203 (1989), pp. 129–132. ISSN: 1476-4687. DOI: [10.1038/337129a0](https://doi.org/10.1038/337129a0). URL: <https://doi.org/10.1038/337129a0>.
- [11] A. R. CURTIS, M. J. D. POWELL, and J. K. REID. “On the Estimation of Sparse Jacobian Matrices”. In: *IMA Journal of Applied Mathematics* 13.1 (Feb. 1974), pp. 117–119. ISSN: 0272-4960. DOI: [10.1093/imamat/13.1.117](https://doi.org/10.1093/imamat/13.1.117). eprint: <https://academic.oup.com/imamat/article-pdf/13/1/117/2141139/13-1-117.pdf>. URL: <https://doi.org/10.1093/imamat/13.1.117>.
- [12] Jeffrey L. Elman. “Finding Structure in Time”. In: *Cognitive Science* 14.2 (1990), pp. 179–211. DOI: https://doi.org/10.1207/s15516709cog1402_1. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1207/s15516709cog1402_1. URL: https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog1402_1.
- [13] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterton. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. URL: <https://proceedings.mlr.press/v9/glorot10a.html>.
- [14] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. “Exploring Network Structure, Dynamics, and Function using NetworkX”. In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.
- [15] Moritz Hardt, Tengyu Ma, and Benjamin Recht. “Gradient descent learns linear dynamical systems”. In: *Journal of Machine Learning Research* 19.29 (2018), pp. 1–44.
- [16] Guyue Huang et al. “GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks”. In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2020, pp. 1–12. DOI: [10.1109/SC41405.2020.00076](https://doi.org/10.1109/SC41405.2020.00076).
- [17] Kazuki Irie, Anand Gopalakrishnan, and Jürgen Schmidhuber. “Exploring the Promise and Limits of Real-Time Recurrent Learning”. In: *International Conference on Learning Representations (ICLR)*. Vienna, Austria, May 2024.

-
- [18] Patrick Kidger and Cristian Garcia. “Equinox: neural networks in JAX via callable PyTrees and filtered transformations”. In: *Differentiable Programming workshop at Neural Information Processing Systems 2021* (2021).
 - [19] Diederick P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *International Conference on Learning Representations (ICLR)*. 2015.
 - [20] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. “Numba: a LLVM-based Python JIT compiler”. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM ’15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340052. DOI: [10.1145/2833157.2833162](https://doi.org/10.1145/2833157.2833162). URL: <https://doi.org/10.1145/2833157.2833162>.
 - [21] Richard Liaw et al. “Tune: A Research Platform for Distributed Model Selection and Training”. In: *arXiv preprint arXiv:1807.05118* (2018).
 - [22] Weifeng Liu and Brian Vinter. “An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data”. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014, pp. 370–381. DOI: [10.1109/IPDPS.2014.47](https://doi.org/10.1109/IPDPS.2014.47).
 - [23] Pierre-Yves Massé and Yann Ollivier. “Convergence of online adaptive and recurrent optimization algorithms”. In: *arXiv preprint arXiv:2005.05645* (2020).
 - [24] Jacob Menick et al. “Practical Real Time Recurrent Learning with a Sparse Approximation”. In: *International Conference on Learning Representations*. 2021. URL: <https://openreview.net/forum?id=q3KSThy2GwB>.
 - [25] Srđan Milaković et al. “Parallel algorithms for masked sparse matrix-matrix products”. In: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’22. Seoul, Republic of Korea: Association for Computing Machinery, 2022, pp. 453–454. ISBN: 9781450392044. DOI: [10.1145/3503221.3508430](https://doi.org/10.1145/3503221.3508430). URL: <https://doi.org/10.1145/3503221.3508430>.
 - [26] Asier Mujika, Florian Meier, and Angelika Steger. “Approximating real-time recurrent learning with random kronecker factors”. In: *Advances in neural information processing systems* 31 (2018).
 - [27] Antonio Orvieto et al. “Resurrecting recurrent neural networks for long sequences”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 26670–26698.
 - [28] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. “On the difficulty of training recurrent neural networks”. In: *International conference on machine learning*. Pmlr. 2013, pp. 1310–1318.

- [29] R Romo et al. “Neuronal correlates of parametric working memory in the prefrontal cortex”. en. In: *Nature* 399.6735 (June 1999), pp. 470–473.
- [30] Friedrich Schuessler et al. “The interplay between randomness and structure during learning in RNNs”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 13352–13362. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/9ac1382fd8fc4b631594aa135d16ad75-Paper.pdf.
- [31] Corentin Tallec and Yann Ollivier. “Unbiased Online Recurrent Optimization”. In: *International Conference on Learning Representations*. 2018. URL: <https://openreview.net/forum?id=rJQDjk-0b>.
- [32] P.J. Werbos. “Backpropagation through time: what it does and how to do it”. In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560. DOI: [10.1109/5.58337](https://doi.org/10.1109/5.58337).
- [33] Ronald J. Williams and David Zipser. “A Learning Algorithm for Continually Running Fully Recurrent Neural Networks”. In: *Neural Computation* 1.2 (1989), pp. 270–280. DOI: [10.1162/neco.1989.1.2.270](https://doi.org/10.1162/neco.1989.1.2.270).
- [34] Carl Yang, Aydın Buluç, and John D. Owens. “GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU”. In: *ACM Trans. Math. Softw.* 48.1 (Feb. 2022). ISSN: 0098-3500. DOI: [10.1145/3466795](https://doi.org/10.1145/3466795). URL: <https://doi.org/10.1145/3466795>.
- [35] Zhinan Zhang. *The Jordan Canonical Form of a Rational Random Matrix*. Working Paper S1574-0358(04)70392-5. Available at SSRN: <https://ssrn.com/abstract=3142770>. Science Direct, Nov. 2002.
- [36] Nicolas Zucchet and Antonio Orvieto. “Recurrent neural networks: vanishing and exploding gradients are not the end of the story”. In: *arXiv preprint arXiv:2405.21064* (2024).
- [37] Nicolas Zucchet et al. “Online learning of long-range dependencies”. In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023. URL: <https://openreview.net/forum?id=Wa1GGPqjUn>.



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

DYNAMICS OF REAL TIME RECURRENT
LEARNING IN SPARSE NETWORKS

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

Kevin Danilo

First name(s):

López Andrango

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the '[Citation etiquette](#)' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

07/08/2024

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.