Learning Sequential Patterns with Recurrent Neural Networks

by

Liam Johnston

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

Doctor of Philosophy

(Statistics)

at the

UNIVERSITY OF WISCONSIN–MADISON

2024

Date of final oral examination: 05/03/2024

The dissertation is approved by the following members of the Final Oral Committee:

Vivak Patel, Assistant Professor, Statistics
Hyunseung Kang, Associate Professor, Statistics and Biostatistics and Medical Informatics
Keith Levin, Assistant Professor, Statistics
Sameer Deshpande, Assistant Professor, Statistics
Prasanna Balaprakash, Director of AI Programs, Oak Ridge National Lab

To mom and dad.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest appreciation to my advisor, Vivak Patel, for his guidance both professionally and personally. It is difficult to express how grateful I am to have had the opportunity to work under his tutelage. Without Vivak's advice, unwavering support and patience, I do not believe that I would have finished this work. For that, I cannot thank him enough.

I would also like to thank my collaborators Prasanna Balaprakash and Yumian Cui for their help, support and general interest in my work. Furthermore, I would like to thank each of my committee members for reviewing my work and taking the time to be a part of my defense.

Lastly, I would like to thank all of the faculty, administrators and students that have helped shape my experience during my time in school. It has been a journey full of highs and lows, but truly one that I will not forget.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The central theme of this dissertation investigates the question, “how do recurrent neural networks (RNNs) learn?" We analyze this question empirically, where data was collected from a massive factorial type experiment where more than 40,000 RNNs were trained on a variety of sequential learning tasks. Using the data generated from this experiment, we explore different aspects of RNN learning in each of the three chapters of this dissertation.

In Chapter 1, we examine the question of how can one RNN be compared against another. We contextualize this question in relation to the standard methodology for comparing machine learning methods, namely, benchmarking. Specifically, we investigate which (if any) model behaviors (e.g., classification accuracy, training time, etc.) observed on one task, are architectural properties that will generalize to other related tasks. We analyze this question using linear models and the data generated from the large factorial experiment to statistically test whether model behaviors do indeed generalize across tasks. The conclusions of this analysis approach suggest that most behaviors commonly used to highlight architecture advantages are task specific (i.e., do not generalize across tasks). That is, an architecture’s task performance is idiosyncratic, and, in particular, is highly sensitive to parameter initialization and learning rate.

In the subsequent chapter we delve into the dynamics imbued by RNNs that make learning long-time dependent structure difficult, namely, the vanishing and exploding gradient problem (VEG). In particular, Chapter 2 is devoted to the characterization of VEG, the discussion of methods that have been developed to address this training ailment, and then introduce two novel methods for mitigating VEG: (i) a powerful optimization framework for controlling training dynamics using second-order sensitivity methods, and (ii) a simple recurrent architecture that is able to maintain gradient flow by the nature of its construction.

The development and experimentation of these new methods demonstrate their ability to deter VEG. However, similar to these method’s competitors, their ability to improve task performance is often inconsistent. This observation motivates the work of Chapter 3, where we reexamine the validity of the machine learning tenet, *if VEG occurs then an RNN learns long-term dependencies poorly*. We investigate this tenet empirically, using the data generated from the factorial experiment detailed in Chapter 1, along with a synthetic experiment designed to study the attractor dynamics of these models. In investigating this tenet, we introduce a simple data analysis approach that allows for the relationship between a model’s task performance and its exhibited gradient dynamics to be statistically quantified. Through this experimentation and analysis we provide evidence that demonstrates that

this tenet does not fully capture the relationship between an RNN's gradient dynamics and their ability to learn.

1 TASK GENERALIZATION

Benchmark tasks are the standard methodology for validating new methods and models in deep learning, and have grown to such a level of importance that models improving benchmark standards are commonly insinuated to be better suited to the benchmark task's underlying application domain. In this work, we assess whether any model behaviors observed on a benchmark task are reproduced in related tasks through an extensive experiment in which we train over 40,000 state-of-the-art recurrent neural networks on sequential classification learning tasks. Using this experiment, we introduce a data analysis approach to analyze model behaviors and their generalization. From this analysis, we observe that common model behaviors, such as classification accuracy on a benchmark task, do not generalize to related tasks. We observe only one model behavior to generalize across related tasks: certain models have a robustness to hyperparameter choices in avoiding training failures. Owing to our extensive experimentation, we provide evidence that validating new methods or models in deep learning should not be done using benchmarks, and claims about the superiority of a model over others owing to its performance on a benchmark task are dubious. Instead, we recommend that the validation of new models and training methods include our approach for analyzing task behaviors and their generalization to ensure a more holistic representation of the new technique.

1.1 Introduction

Over the past decade, neural networks (NNs) have been deployed in significant real-world applications that include autonomous vehicles, natural language generation, and even scientific breakthroughs in protein structure prediction [78, 79, 50]. These successful applications have made improving NN models an active area of research. In the research community, the improvement provided by a novel NN model is often determined through testing and comparison with preexisting models on *benchmark tasks* – a task or suite of tasks, datasets and performance metrics used to compare machine learning methods [84, 82]. These benchmark tasks are instances of model applications (i.e., larger families of tasks) that are of either industry or academic interest. Hence, as more application areas are addressed using NNs, the number and variety of proposed benchmark tasks grow accordingly. As illustration, benchmark tasks for several application domains are displayed in Table 1.1, alongside of models that performed the benchmark, the application domain, and real-world applications (if available).

As Table 1.1 illustrates, benchmark tasks have been a springboard for models to be scaled-up and deployed for real-world, high-profile applications [102, 5, 27, 90]. As a

Disclaimer: Portions of this chapter are pulled directly from a co-authored paper written with Vivak Patel and Prasanna Balaprakash that is currently under review at *the international conference on automated machine learning* (*AutoML*).

Application Domain	Benchmark/Task	Models	Applications
Long Time Dependent Tasks	Sequential MNIST [26] Permuted MNIST [26] Adding Problem [42] IMDB Sentiment Analysis [68]	LSTM [42] Exponential RNN [65] UnICORNN [87] Antisymmetric RNN [16] Lipschitz RNN [32] Unitary RNN [3] Volume-preserving RNN [99]	
Natural Language Understanding	GLUE [109] The Penn Treebank [69] Super GLUE [108]	BERT [27] Transformer [102]	ChatGPT4 [79]
Computer Vision	ImageNet [25] COCO [66] KITTI [34] ObjectNet [7] LabelMe [88]	DaViT [28] ResNet [39] CoCa [112] RevCol [15] LiT[114] VGGNet [91]	
Reinforcement Learning	ALE [8] dm_control [101] Meta-World [113] MuJoCo [100]	R2D2 [52] Deep Q-Networks [73]	Go Chess Atari
Medicine	EyePACS (diabetic retinopathy) [51] APTOS (blindness detection) [53] CASP (protein structure) [57] FLIP (protein structure) [24] PTB-XL (electrocardiography) [107] PTB-XL+ (electrocardiography) [95, 94] CPSC, CPSC Extra (electrocardiography) [67]	1D CNN Ankh [30] ResNet/CNN [39, 6]	AlphaFold [50]

Table 1.1: Benchmark tasks, application domain, models that performed the benchmark, and real-world applications.

result of these successes, benchmark tasks: (1) appear to be appropriate representatives of the application domain in which they were designed to represent, and (2) accurately reflect a model's ability to learn other related (real-world) tasks (demonstrated by the success of real-world applications that employ models that were first shown introduced on benchmark tasks).

Yet, state-of-the-art large language models trained on an overwhelming corpus of data (in some sense a thorough benchmark task) still require specialization to achieve good performance on related tasks (see Section 1.2.3). This observation motivates this work. Specifically, we investigate whether benchmark tasks are a meaningful tool for assessing models. To this end, we introduce two concepts: *model behavior* and *task generalization*.

While we will be precise later (see Definitions 1.1 and 1.2), we define model behavior as any measurable quality of a model and/or training process; and task generalization as the ability of that quality to be reproduced on all tasks within a larger family of interest. Using these concepts, we ask,

> *Do any model behaviors on a benchmark task generalize to a larger family of related tasks?*

If the answer is positive, then benchmark tasks are useful assessment tools. If the answer is no, then the utility of benchmarks in comparing novel models or training methods is questionable.

To investigate this question, we conduct a massive experiment on seven recurrent architectures each of which was evaluated over a fixed hyperparameter grid (twenty unique hyperparameter configurations) on a set of thirty sequential classification tasks. In total, more than 40,000 RNNs were trained, requiring more than 14 years of computational time (measured by CPU hours) executed in parallel on Argonne National Lab's supercomputer, Theta [33, 44].

Using the data collected from these experiments, we formulate various model behaviors and test their respective abilities to generalize across tasks. Using a data analysis approach that is novel for this application, our overaching conclusions are:

1. Task performance (e.g., classification accuracy) attained at a maximizing hyperparameter configuration is not a consistent behavior that can be reproduced across families of related tasks.

2. The interactions between a NN model and a fixed grid of training hyperparameter configurations are unique to the architecture and specific task.

3. Some architectures are robust to hyperparameters allowing them to consistently complete all training epochs. Although this does not imply high task accuracy, this was the only behavior that we observed to be reliably reproduced from task-to-task in our experiments.

In summary, almost all model behaviors considered do not generalize from benchmark tasks to a larger family of tasks. In light of this, rather than using benchmark tasks, we recommend end-users to carefully tune well-established models with which they are familiar to their particular applications and computing environments. Furthermore, we recommend that the validation of new models and training methods include our approach for analyzing

task behaviors and their generalization to ensure a more holistic representation of the new technique.

Chapter Organization

In Section 1.2, we provide a background of related work and discuss issues with deep learning that make task generalization a highly skeptical feature. Following, we formally define model behavior and task generalization, and introduce a data analysis approach for studying these properties in Section 1.3. In Section 1.4, we detail the experimental design used to investigate model behaviors and task generalization, and, in Section 1.5, we formulate various model behaviors and analyze their ability to generalize across tasks using both qualitative assessment and formal hypothesis tests. Lastly, in Section 1.6, we conclude.

1.2 Background

1.2.1 Validity of Comparisons on Benchmarks

Benchmarking in computer science dates back to the early 1960s and was first formalized as a systematic method for measuring the computational speed of a computer system [63]. Since then, this framework has been adopted by various areas of computer science, including deep learning, to facilitate the evaluation and comparison of new methodologies. However, in the case of deep learning, the degree to which a benchmark task is a valid comparison tool is not clear [77, 72, 45, 12]. In particular, [82] highlights the limitations of emphasizing benchmark tasks as milestones toward artificial general intelligence, stating "... benchmarks – due to their instantiation in particular data, metrics and practice – cannot possibly capture anything representative of the claims to general applicability being made about them."

1.2.2 Dependence on Uncontrolled Variables

Benchmark tasks are used as a means to validate or compare models *ceteris paribus* using standardized metrics. Although comparison practices have improved in DL, the complexity of optimizing NNs still involves numerous uncontrolled implementation decisions including: data pre-processing, hyperparameter tuning, and randomization.

1. Typically benchmark tasks are associated with specific data that is used for training and testing. While this consistency may suggest an identical starting point for

cross-study comparison, it is customary for data to undergo pre-processing prior to training (e.g., standardization, rotation, word embedding), which can have a significant impact on task performance [89].

2. Current strategies for defining and training NNs require numerous hyperparameters (e.g., weight dimensions, learning rate, batch size, layer depth, regularization, etc.). Not only will choices in hyperparameters impact a model's task performance, but the sensitivity to these choices may vary substantially across different models and tasks. Consequentially, if a new method improves a benchmark standard, determining whether the improvement should be attributed to the novelty of the method, or to a more thorough tuning process is unclear (particularly when improvement is marginal) [70, 31, 41]. Moreover, while ablation studies can aid in conveying the sensitivity of a method to hyperparameter choice, whether its conclusions generalize across tasks is typically left unexplored.

3. Lastly, some uncontrolled sources of experimental variation appear to be unavoidable and non-negligible. For instance, randomization — commonly used in both weight initialization and stochastic training routines — has the capacity to degrade performance from state-of-the-art to mediocre [83]; and an engineer's prior experience has been shown to be strongly correlated with resource utilization efficiency and task performance [2].

In this chapter, a model's performance on a benchmark task is shown to be dependent on the hyperparameter choices, randomization, and, to an extent, data representation.

1.2.3 Task Generalization and Large Language Models

Recently, advancements in natural language processing (NLP) have suggested that large language models (LLMs) have the ability to generalize across a variety of language tasks [14, 27]. These models are developed using an expensive pre-training routine that learns a language representation from a large amount of unlabeled text data [79]. Then, using the learned representation from pre-training, an LLM is specialized to elicit multi-task functionality.

The most successful of these strategies uses the pre-trained language representation as a starting point to fine-tune task-specific models from labeled task training data [64, 27]. In other words, even for state-of-the-art LLMs, it is understood that a general model has a

substandard performance on related tasks. An alternative methodology uses prompts to provide context to guide the model towards task-specific outcomes [76, 14, 118, 110]. While these methods have shown promise, they still also indicate that the input corresponding to the task must be tailored to the LLM [117], further corroborating how specific model performance is to a given task.

These results align with our experimental observations. Namely, although we investigate significantly smaller models than LLMs, we observe that task performance does not generalize across task families. In particular, there are task idiosyncrasies such that learning one task does not imply the ability to learn another related task. Similarly, LLMs do not acquire multi-task functionality by default, but only when explicitly tuned or prompted to specific tasks.

1.3 Generalization of Task Behaviors

1.3.1 Problem Set-Up

Our objective is to understand which (if any) model behaviors are consistently reproduced on all data tasks within a particular family of tasks. As such, we focus our attention to behaviors that are data task agnostic, or in other words, can be measured on any data task. With this in mind, we formulate our analysis and subsequent definition of model behavior taking value in $\mathcal{Y}$, and three spaces that all supervised tasks require: a model or architecture space, $\mathcal{A}$; a hyperparameter space $\mathcal{H}$; and a task space $\mathcal{T}$. These three spaces describe task information known a priori to training and will be the subject for defining *model behaviors*.

Definition 1.1 (Model Behavior)**.** *A model behavior is a function* $B : \mathcal{A} \times \mathcal{H} \times \mathcal{T} \to \mathcal{Y}$ *that maps a model architecture, hyperparameter configuration and task information to a measure of performance.*

We define the task generalization of a model behavior as the ability of a behavior to be reproduced across all tasks within a larger task family of interest.

Definition 1.2 (Task Generalization)**.** *The task behavior,* $B(a, h, t)$*, generalizes over the task family,* $\mathcal{T}$*, if*

$$B(a, h, t) \approx B(a, h, t') \tag{1.1}$$

for all $t, t' \in \mathcal{T}$*.*

1.3.2 Modeling Behaviors & Testing Task Generalization

We use the experiment described in Section 1.4 as an observational study to investigate model behaviors over a family of thirty sequential learning tasks. The experiment can be described by a design matrix, $X = [\mathcal{A}|\mathcal{H}|\mathcal{T}]$, where columns correspond to the experimental factors: model architecture, training hyperparameters and task; and rows correspond to individual training attempts. For each training attempt, a model behavior, Y, is observed/measured (e.g., classification accuracy).

In order to understand the primary contributors to a model behavior B using a sampling of its values, Y, we use the following descriptive model:

$$Y = \beta_0 + \beta_{\mathcal{A}} \times \mathcal{A} + \beta_{\mathcal{H}} \times \mathcal{H} + \beta_{\mathcal{A},\mathcal{H}} \times [\mathcal{A} * \mathcal{H}] + \epsilon. \tag{1.2}$$

This model approximates the behavior, Y, as a function of the training attempt architecture and hyperparameter configuration over the family of experimental tasks.

To investigate task generalization, we fit the task-specific linear model,

$$\hat{Y}^{(t)} = \hat{\beta}_0^{(t)} + \hat{\beta}_{\mathcal{A}}^{(t)} \times \mathcal{A} + \hat{\beta}_{\mathcal{H}}^{(t)} \times \mathcal{H} + \hat{\beta}_{\mathcal{A},\mathcal{H}}^{(t)} \times [\mathcal{A} * \mathcal{H}] + \epsilon^{(t)}, \tag{1.3}$$

where t indexes the experimental task and also defines the corresponding rows/indices of $\{X, Y\}$ used for estimating (1.3). The regression coefficient $\beta_{\mathcal{A}}^{(t)}$ is estimated from observation pairs belonging to task t, and encodes the effect a particular architecture has on the behavior Y. If this effect generalizes across tasks, then $\hat{\beta}_{\mathcal{A}}^{(t)} \approx \hat{\beta}_{\mathcal{A}}^{(t')}$ for all $t, t' \in \mathcal{T}$. We formalize this question of task generalization as the pairwise comparison hypothesis test,

$$H_0 : \ \beta_{\mathcal{A}}^{(t)} = \beta_{\mathcal{A}}^{(t')} \quad \text{vs.} \quad H_A : \ \beta_{\mathcal{A}}^{(t)} \neq \beta_{\mathcal{A}}^{(t')}. \tag{1.4}$$

In total, we perform $\binom{|\mathcal{T}|}{2}$ pairwise hypothesis tests and use a Bonferroni correction to alleviate the likelihood of erroneous conclusions from multiple-comparisons.

1.4 Experimental Design

The experimental design can be fully described by the experimental factors and levels of Table 3.2. Alternatively, our design can be described as training $|\mathcal{A}| = 7$ RNN architectures using $|\mathcal{H}| = 20$ hyperparameter configurations on $|\mathcal{T}| = 30$ sequential classification tasks.[1]

[1]We denote a hyperparameter configuration as a unique combination of recurrent dimension, learning rate and training length; and task as a unique combination of dataset, order and noise orientation.

In total there are 4,200 unique combinations of $\mathcal{A}$, $\mathcal{H}$ and $\mathcal{T}$, of which each is replicated ten times for a total model count of 42,000 training attempts.

	Factor	Levels
$\mathcal{A}$	Architecture	Basic RNN [86] LSTM [42] GRU [21] Lipschitz RNN [32] Antisymmetric RNN [16] Exponential RNN [65] UnICORNN [87]
$\mathcal{H}$	Recurrent dimension	{128, 256}
	Learning rate	$\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$
	Training length	{25 epochs, 50 epochs}
$\mathcal{T}$	Dataset	CIFAR10 ($\mathsf{T} = 1024$) Fashion MNIST ($\mathsf{T} = 784$) IMDB ($\mathsf{T} = 500$) MNIST ($\mathsf{T} = 784$) Reuters ($\mathsf{T} = 500$)
	Order	{sequential, permuted}
	Noise orientation	{none, post, uniform}

Table 1.2: Experimental factors and levels.

Each dataset contributes six task variations (i.e., combinations of the factors "Order" and "Noise Orientation" in Table 3.2) where the original input sequence is altered by either a fixed permutation and/or augmented with additional noise tokens. Table 1.3 below, details how the input sequence is altered with each task variation; and in Fig. 1.1, we visualize how different task variations present a sample observation from the MNIST dataset.

Order × Noise Orientation	Input Sequence	Time Horizon
sequential × none	$\{u_t\}_{t=1}^{500}$	500
permuted × none	$\{u_{\pi(t)}\}_{t=1}^{500}$	500
sequential × post	$\{u_1, u_2, \ldots, u_{500}, n_1, n_2, \ldots, n_{1000}\}$	1500
permuted × post	$\{u_{\pi(1)}, u_{\pi(2)}, \ldots, u_{\pi(500)}, n_1, n_2, \ldots, n_{1000}\}$	1500
sequential × uniform	$\{u_1, n_1, n_2, u_2, n_3, n_4, \ldots, u_{499}, n_{997}, n_{998}, u_{500}, n_{999}, n_{1000}\}$	1500
permuted × uniform	$\{u_{\pi(1)}, n_1, n_2, u_{\pi(2)}, n_3, n_4, \ldots, u_{\pi(499)}, n_{997}, n_{998}, u_{\pi(500)}, n_{999}, n_{1000}\}$	1500

Table 1.3: Effect of factors order and noise orientation on task input sequence where we denote $\{u_t\}_{t=1}^{500}$ as the original (unaltered) input sequence; $\{n_t\}_{t=1}^{1000}$ as a noise sequence drawn uniformly from the distribution of the original input sequence; and π a fixed permutation which is applied to the original input sequence.

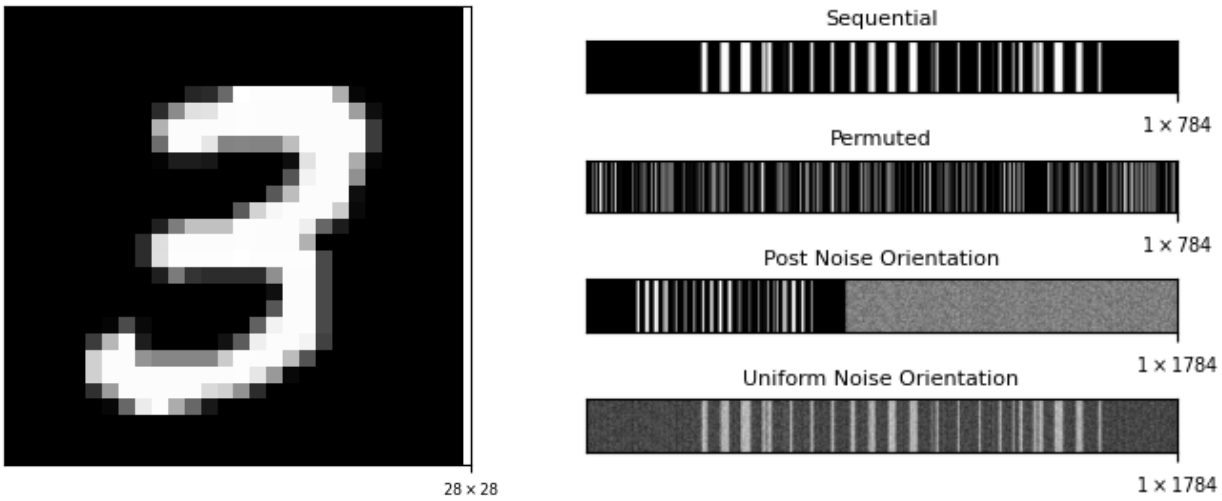


Figure 1.1: An example of how input (feature) sequences change based on task variation (i.e., combination of the experimental factors *order* and *noise orientation*) for an observation from the MNIST dataset. The left image corresponds to the 28×28 MNIST image. For the sequential task variation, the image is squashed to a sequence of $28 \times 28 = 784$ pixels (right, first row). The permuted task variation applies a fixed permutation to the sequential task's sequence (right, second row). A noise sequence of length $T = 1000$ is appended to the original input sequence for tasks where the experimental factor *noise orientation* $=$ $\{post, uniform\}$. For post noise tasks, the additional noise is appended to the end of the original input sequence (right, third row); and for uniform noise tasks, noise is uniformly dispersed throughout the original input sequence (right, fourth row).

Recall that our objective is to understand whether model behaviors observed on one task generalize to a larger family of related tasks. In general, determining the degree in which

tasks are related is difficult to characterize. However, with respect to benchmarks, we believe that the community sentiment is driven by the application/domain area. Our experimental design reflects this where $\mathcal{T}$ consists of tasks that all require learning sequential patterns, and $\mathcal{A}$ consists of architectures designed specifically for learning sequential patterns.

Owing to the scale of the conducted experiments, we present our analysis considering three different task families of interest using the thirty experimental tasks.

1. $\mathcal{T}_f$: tasks consisting of all thirty experimental tasks.

2. $\mathcal{T}_o$: tasks consisting the five data tasks that correspond to each dataset with no additional noise appended to the input sequence.

3. $\mathcal{T}_d$: tasks that share a common dataset.

In the context of task generalization for family $\mathcal{T}_f$, we study the ability of model behaviors to generalize across all thirty experimental tasks; for $\mathcal{T}_o$, we study the behaviors that generalize across the ten tasks (5 data sets by 2 orderings) that do not have additional noise appended to their input sequences; and for $\mathcal{T}_d$ we study the behaviors that generalize to all tasks that use the same data set (e.g., there are 6 CIFAR10 tasks, MNIST has 6 tasks, etc.).

1.4.1 Data Collected and Measured

For each training attempt, we collect the (average) classification accuracy computed both over the training set and a disjoint evaluation set. However, directly studying classification accuracy in terms of task generalization is problematic. Tasks differ in the number of label classes, K, making the range of classification accuracy dependent on K (e.g., $\mathcal{Y} = [\frac{1}{K}, 1]$). As a consequence, the meaning of a particular level of classification accuracy will change depending on the task. For example, for a task where $K = 10$, an accuracy of 50% could be a significant level of learning (with balanced labels); while for a task with $K = 2$, accuracy of 50% is no better than random class assignment.

We mitigate this issue by introducing a task standardized measure of classification accuracy, *task accuracy*, which we denote by Y_{accuracy} and define as,

$$Y_{\text{accuracy}} = \frac{\text{test accuracy} - \frac{1}{K}}{\max\left(\text{test accuracy}\right) - \frac{1}{K}} \tag{1.5}$$

where K corresponds to the number of task class labels; and the maximum in the denominator is taken over all experimental training attempts performed on the respective task. When labels are balanced, (1.5) standardizes performance to $[0, 1]$ where zero indicates

learning did no better than random label assignment, and one indicates the maximum task accuracy observed over all task training attempts (of which there are 1,400).[2]

In the subsequent section, we direct our emphasis to the behavior Y_{accuracy} as it is the typical metric used to understand model performance in classification problems. In addition, we formulate and study behaviors related to training RNNs: convergence time and numerical error rate. For convergence time, we study the variable Y_{epochs}, which we define as the number of training epochs completed prior to the model attaining its maximum task accuracy. For training error rate, we study the variable Y_{error}, which we define as the proportion of task training attempts (on a per-model basis) that encountered numerical failure during training.

1.5 Task Behaviors

To better facilitate our discussion we organize model behaviors into two categories which we coin as *first-order behaviors* and *second-order behaviors*. Our delineation of these two categories was inspired by current practices in model reporting where typically only the best hyperparameter configuration is considered. Hence, we define first-order behaviors as behaviors measured on a model under its maximizing hyperparameter configuration (denoted by $h^* \in \mathcal{H}$), and second-order behaviors as behaviors that consider both maximizing and non-maximizing hyperparameter configurations. Accordingly, first-order behaviors are formulated as simple linear models,

$$Y^{(t)} = \beta_0^{(t)} + \beta_{\mathcal{A}}^{(t)} \times \mathcal{A} + \epsilon^{(t)} \quad \text{for each } t \in \mathcal{T}, \tag{1.6}$$

and second-order behaviors with multivariate linear models that include covariates for hyperparameters and interactions,

$$Y^{(t)} = \beta_0^{(t)} + \beta_{\mathcal{A}}^{(t)} \times \mathcal{A} + \beta_{\mathcal{H}}^{(t)} \times \mathcal{H} + \beta_{\mathcal{A},\mathcal{H}}^{(t)} \times [\mathcal{A} * \mathcal{H}] + \epsilon^{(t)} \tag{1.7}$$

In Table 1.4, we summarize the model behaviors investigated (described next), and indicate whether the behavior generalizes across tasks.

[2]When labels are not balanced, either in the training set or testing set, it is possible for this metric to be less than zero. In practice, this can occur, but tends not to as the RNNs tend to all learn reasonably well when the optimizer does not diverge during training.

Behavior	Analysis Space	Analysis Method	Metric	Tested Effect	Test (H_0)	Task Generalization
Average task accuracy	$h^* \times \mathcal{T}_f$	Linear model (1.8)	$Y_{accuracy}$	$\mathcal{A}$	$\beta_{\mathcal{A}}^{(t)} = \beta_{\mathcal{A}}^{(t')}$	✗
Variance of task accuracy	$h^* \times \mathcal{T}_o$	Brown-Forsythe Test (Table 1.6)	$Y_{accuracy}$	$\mathcal{A}$	$\sigma^2_{a,t} = \cdots = \sigma^2_{a,\lvert\mathcal{T}\rvert}$	✗
Convergence time	$h^* \times \mathcal{T}_f$	Fig. 1.3 and appendix Table A2	Y_{epochs}	$\mathcal{A}$	NA	✗
Average task accuracy	$\mathcal{H} \times \mathcal{T}_o$	Linear model (1.10)	$Y_{accuracy}$	$\mathcal{A}$	$\beta_{\mathcal{A}}^{(t)} = \beta_{\mathcal{A}}^{(t')}$	✗
	$\mathcal{H} \times \mathcal{T}_o$	Linear model (1.10)	$Y_{accuracy}$	$\mathcal{H}$	$\beta_{\mathcal{H}}^{(t)} = \beta_{\mathcal{H}}^{(t')}$	✗
	$\mathcal{H} \times \mathcal{T}_o$	Linear model (1.10)	$Y_{accuracy}$	$\mathcal{A} \times \mathcal{H}$	$\beta_{\mathcal{A},\mathcal{H}}^{(t)} = \beta_{\mathcal{A},\mathcal{H}}^{(t')}$	✗
Training error rate	$\mathcal{H} \times \mathcal{T}_f$	(Table 1.7)	Y_{error}	$\mathcal{A}$	NA	✗/✓

Table 1.4: Model behaviors, investigated hyperparameter and task spaces, linear model formula (if applicable), factor/coefficient tested, formal hypothesis test (if applicable), and indicator of task generalization. The sectioning of the table delineates first-order (top) and second-order (bottom) indicators.

1.5.1 First-order Behaviors

Average Task Accuracy

Define the model behavior *average task accuracy* as $\mathbb{E}[Y_{accuracy}|(\mathcal{A}, h^*, t)]$. For each task, we estimate this behavior with the simple linear model,

$$\hat{Y}_{accuracy}^{(t)} = \hat{\beta}_0^{(t)} + \hat{\beta}_{\mathcal{A}}^{(t)} \times \mathcal{A} + \epsilon^{(t)}. \tag{1.8}$$

This collection of linear models is able to estimate the relationship between task accuracy and architecture well, where the majority of models capture more than 70% of the variation in $Y_{accuracy}$ (summarized in Table A1 of the appendix).

Using the testing framework described in Section 1.3, we perform pairwise hypothesis tests akin to (1.4), in order to statistically test whether average task accuracy is a task generalizable behavior.

Recall that we formulate task generalization as the null hypothesis of (1.4), which suggests that the effect, $\beta_{\mathcal{A}}^{(t)}$, remains (statistically) constant across the task space $\mathcal{T}$. Unfor-

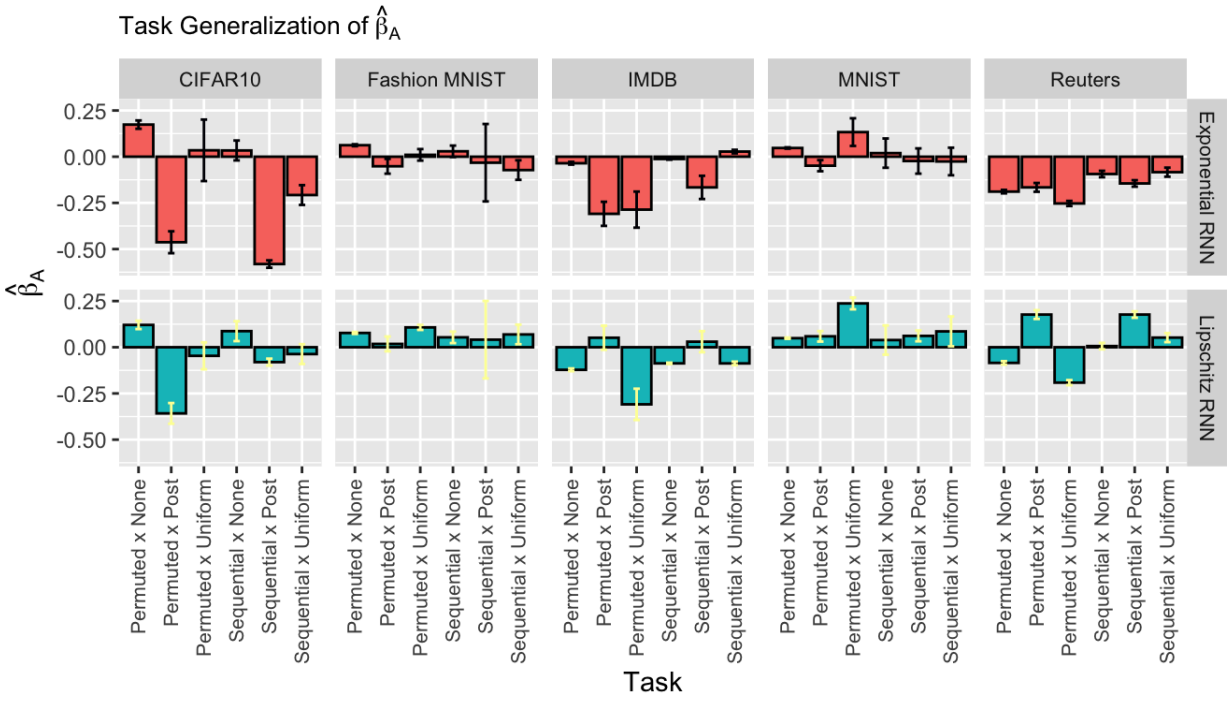


Figure 1.2: Estimated coefficient, $\hat{\beta}_{\mathcal{A}}$, for two architectures (Exponential RNN and Lipschitz RNN) across thirty experimental tasks partitioned by task dataset (column facet) and RNN architecture (row facet).

tunately, this is not observed. We include test summaries in Table 1.5, and use Fig. 1.2 to illustrate their conclusions.

In Fig. 1.2 we display the estimated coefficient, $\hat{\beta}_{\mathcal{A}}$, for two RNN architectures, where $\hat{\beta}_{\mathcal{A}}$ is estimated relative to a reference architecture, for one of the thirty tasks.[3] In the case that the estimated architecture effect generalizes across tasks, then $\hat{\beta}_{\mathcal{A}}$ would vary little across tasks. However, as demonstrated in Fig. 1.2, the estimated coefficient $\hat{\beta}_{\mathcal{A}}$ varies significantly depending on the task. We observe this trend for all tested RNN architectures and display the proportion of pairwise tests for each architecture that rejected the null hypothesis in Table 1.5.

[3]We construct Fig. 1.2 using Antisymmetric RNN [16] as the reference architecture.

	Fail to Reject H_0	Reject H_0
Antisymmetric RNN	0.35	0.65
Basic RNN	0.26	0.74
Exponential RNN	0.58	0.42
GRU	0.60	0.40
Lipschitz RNN	0.38	0.62
LSTM	0.56	0.44
UnICORNN	0.33	0.67

Table 1.5: Proportion of pairwise comparison tests that resulted in statistically different $\beta_{\mathcal{A}}$ (constructed via task-specific model (1.8)) for seven RNN architectures at testing level of 0.05.

Variance of Task Accuracy

The behavior *variance of task accuracy* describes the variation in task accuracy accrued from reinitialization and training of a model at h^*. To interrogate this behavior we define $\sigma^2_{a,t}$ as the variance of task accuracy for architecture, $a \in \mathcal{A}$, and task, $t \in \mathcal{T}$, and perform a Brown-Forsythe test (see [13] for test details) for equality of group variances within a given architecture for the task family $\mathcal{T}_o$:

$$H_0 : \ \sigma^2_{a,t_1} = \sigma^2_{a,t_2} = \cdots = \sigma^2_{a,t_{|\mathcal{T}_o}} \quad \text{vs.} \quad H_A : \text{Not all } \sigma^2_{a,t_i} \text{ are equal} \tag{1.9}$$

for $t_i \in \mathcal{T}_o$. The results of this test (one for each architecture displayed in Table 1.6) indicate that the variance of task accuracy is not consistent over the task family, and thus, not a behavior that generalizes across tasks.

	test statistic (dfs)	p value	H_0 decision
Antisymmetric RNN	572.06 (9/37.35)	3.4×10^{-37}	reject
Basic RNN	188.29 (9/32.54)	1.9×10^{-25}	reject
Exponential RNN	618.77 (9/34.41)	2.4×10^{-35}	reject
GRU	4.58 (9/11.27)	1.0×10^{-3}	reject
Lipschitz RNN	557.86 (9/46.31)	4.2×10^{-44}	reject
LSTM	5.75 (9/16.49)	1.0×10^{-3}	reject
UnICORNN	518.07 (9/31.91)	5.5×10^{-32}	reject

Table 1.6: Results of Brown-Forsythe test for equality of group variances across family $\mathcal{T}_o$ where each group is comprised of training attempts that maximized each respective architecture's test accuracy at the terminal training epoch.

Convergence Time

We measure the first-order behavior *convergence time* with the variable, Y_{epoch}, which we define as the number of training epochs completed prior to the RNN attaining its maximum task accuracy. For this behavior we consider the data generated during the first 25 epochs for each training attempt. To study this behavior we compute the median and standard deviation of Y_{epoch} for each architecture and task (appendix Table A2) and use Fig. 1.3 to illustrate our observations.

In Fig. 1.3 we draw boxplots of Y_{epoch} for each task variation (horizontal axis) across architectures (column facets) and datasets (row facets). Clearly, this behavior varies across the experimental task space (both with respect to task dataset and task variation). Interestingly, for some architectures and datasets, convergence time across the six task variations appears to be fairly consistent (e.g., top left panel of Fig. 1.3). However, again, this consistency does not generalize across task datasets for any architecture.

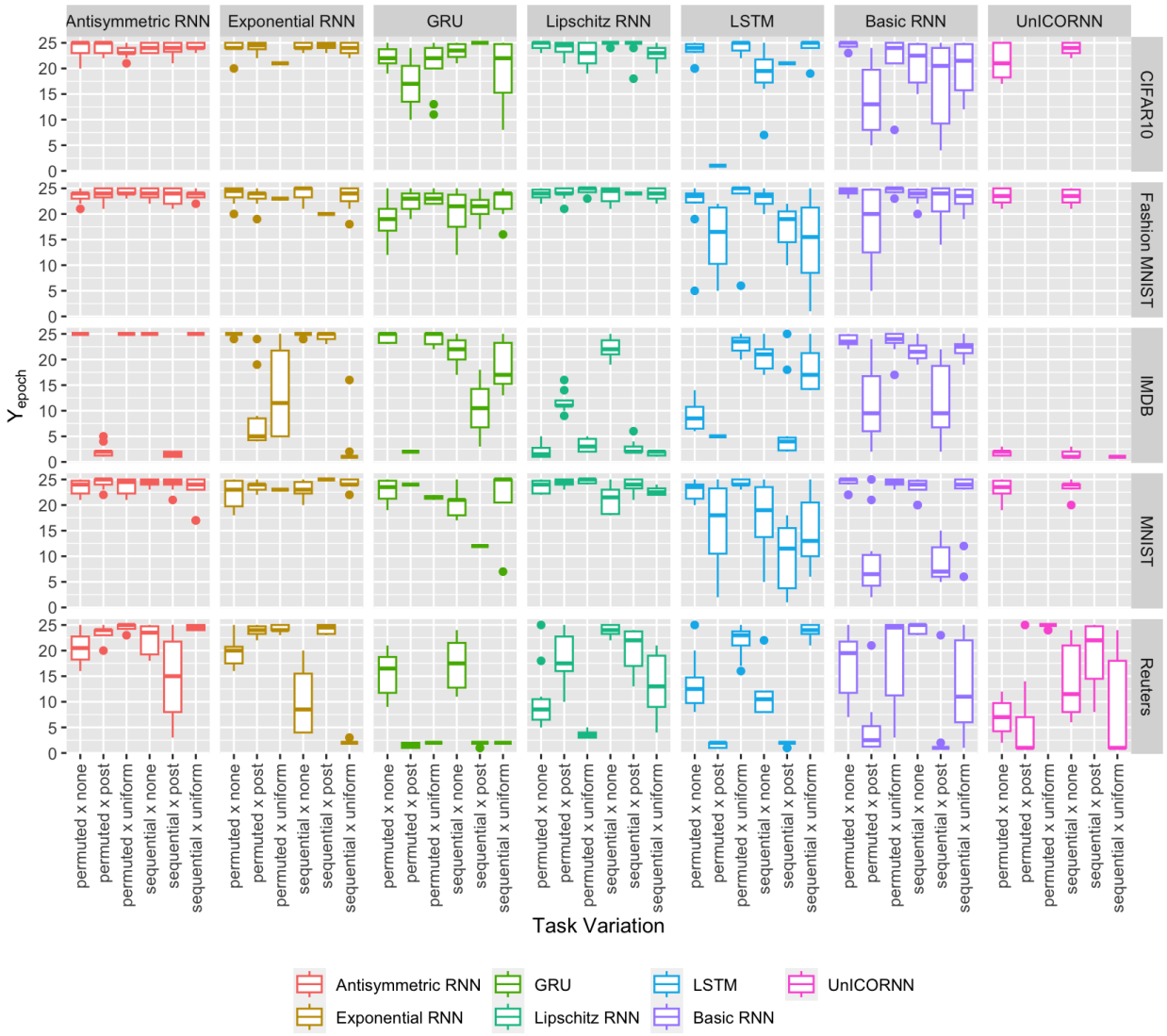


Figure 1.3: Boxplots of $Y_{\texttt{epoch}}$ for each task variation (horizontal axis) across architectures (column facets) and datasets (row facets). For the UnICORNN architecture, only a subset of tasks were able to be completed owing to extremely long training times.

1.5.2 Second-order Behaviors

Average Task Accuracy

We extend our investigation of the behavior *average task accuracy* to a second-order behavior by studying it as a function of the entire hyperparameter space, $\mathcal{H}$. We reflect this change in the linear estimator formula,

$$\hat{Y}^{(t)}_{\text{accuracy}} = \hat{\beta}^{(t)}_0 + \hat{\beta}^{(t)}_{\mathcal{A}} \times \mathcal{A} + \hat{\beta}^{(t)}_{\mathcal{H}} \times \mathcal{H} + \hat{\beta}^{(t)}_{\mathcal{A},\mathcal{H}} \times [\mathcal{A} * \mathcal{H}] + \epsilon^{(t)} \tag{1.10}$$

where $\hat{\beta}_{\mathcal{H}}$ denotes regression coefficients associated with the hyperparameters: learning rate, second-order learning rate, recurrent dimension and training length; and $\hat{\beta}_{\mathcal{A},\mathcal{H}}$ denotes the regression coefficients of the interactions between: architecture and learning rate, and architecture and second-order learning rate. Hence, we can write the linear model formula in the context of our experimental factors as:

$$\begin{aligned}\hat{Y}_{acc}^{(t_i)} =& \hat{\beta}_0^{(t_i)} + \hat{\beta}_{arch}^{(t_i)} \times \text{architecture} + \hat{\beta}_1^{(t_i)} \times \log(\text{learning rate}) \\ &+ \hat{\beta}_2^{(t_i)} \times \log(\text{learning rate})^2 + \hat{\beta_3}^{(t_i)} \times \text{rec. dimension} \\ &+ \hat{\beta}_4^{(t_i)} \times \text{training length} + \hat{\tau}_{arch}^{(t_i)} \times (\text{architecture} \times \log(\text{learning rate})) \\ &+ \hat{\gamma}_{arch}^{(t_i)} \times (\text{architecture} \times \log(\text{learning rate})^2) + \epsilon^{(t_i)}\end{aligned} \tag{1.11}$$

where t_i indexes the thirty experimental tasks of $\mathcal{T}_f$. The model formula was chosen based on quality. We summarize each of the thirty task-specific models in Table A3 of the appendix, and note here that nearly all estimated models are able to capture more than 70% of the variation in task accuracy.

We test each regression coefficient of (1.11) for task generalization following the procedure described in Section 1.3. Similar to the first-order behavior, no estimated effect ($\hat{\beta}_{\mathcal{A}}$, $\hat{\beta}_{\mathcal{H}}$ or $\hat{\beta}_{\mathcal{A},\mathcal{H}}$) successfully generalized across tasks. In Fig. 1.4 we display the proportion of pairwise tests that rejected H_0 for each of the respective regression coefficients in (1.11).

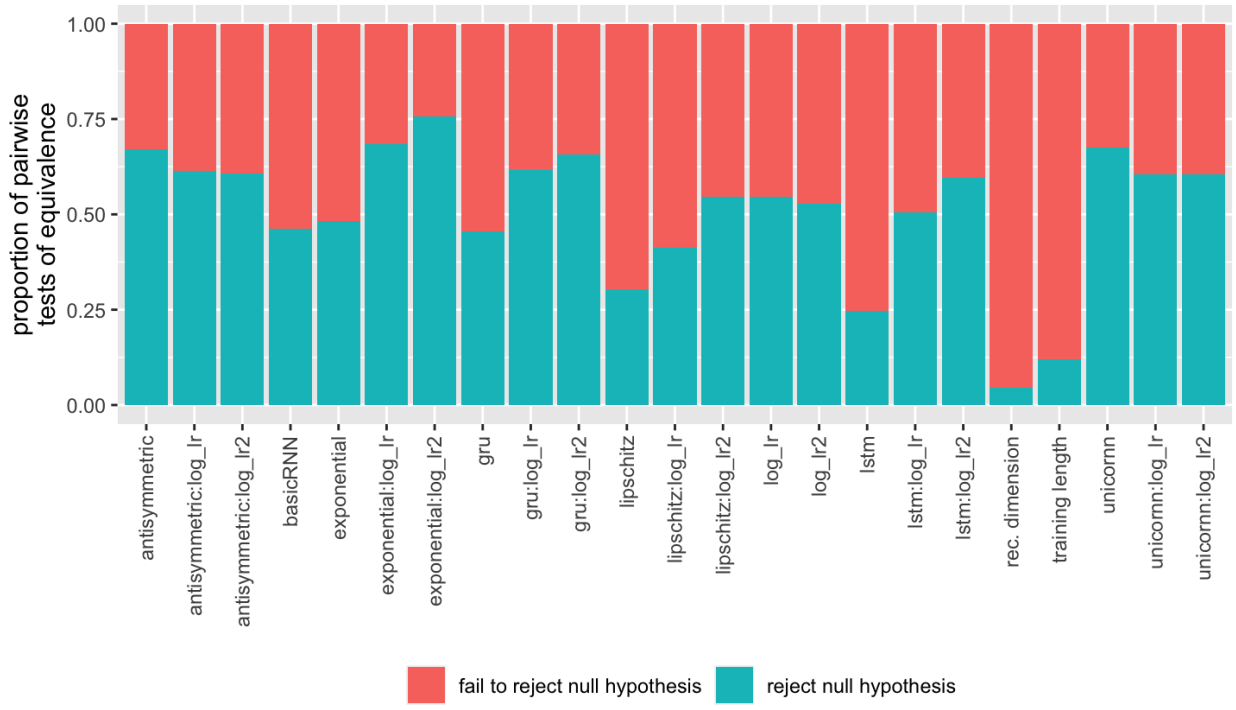


Figure 1.4: The proportion of rejected hypotheses, $H_0 : \beta^{(i)}_{coef} = \beta^{(j)}_{coef}$, at a significance level of $\alpha = 0.05$ for each predictor of the estimated linear model, (1.11).

Recall from Section 1.2, that there are numerous experimental variables that commonly are uncontrolled during experimentation, like hyperparameter selection, that will impact an architecture's task performance. This scenario is corroborated in our experiment, where we observe that all of the interaction terms encoded in the multivariate linear model, (1.11), statistically varied across data tasks. This feature captures the idea that interactions between an architecture and the selected hyperparameters are highly task-dependent, and do not generalize to larger task families with a degree of certainty.

While these results are perhaps not surprising, it is informative in explaining why we observe the tendency for incredibly involved training procedures of modern models. Namely, there is an immense amount of tinkering and familiarity with an architecture and dataset needed in order to optimize modern models. Furthermore, this tailoring appears to be unavoidable when the objective sought is maximizing task performance (e.g., maximizing classification accuracy or minimizing test error). Hence, if an architecture's effectiveness is to be determined by its task capacity, then there needs to be a much larger emphasis placed on the detail of the training procedure used, as well as the deviation in the model's performance when alternative training strategies are used. This is particularly crucial in the current environment of machine learning where training resource availability substantially differs between research labs and industry leaders. Without this level of

detail, resource allocation cannot be appropriately made, and as a consequence, limits the practical usability of new models.

Training Error Rate

One of the main difficulties with using neural networks is the difficulty in training them. This problem is particularly troublesome for very deep neural networks (which most modern networks are) as well as recurrent networks such as our experimental architectures [80, 48]. We test this with the behavior *training error rate* as the proportion of training attempts that produced numerical error during training. Computing error rate for each architecture and task revealed this behavior to be *architecture dependent*, where three architectures — Exponential RNN, Antisymmetric RNN and UnICORNN — displayed a robust relationship with the training process (less than 1% error rate), while the other four architectures consistently encountered numerical errors during training, dependent on the task and hyperparameter configuration applied.

	Error Rate
Antisymmetric RNN	0.0
Exponential RNN	< 0.01
GRU	0.43
Lipschitz RNN	0.32
LSTM	0.28
Basic RNN	0.12
UnICORNN	0.0

Table 1.7: Proportion of training instances that produced numerical errors during training for each RNN architecture.

1.6 Conclusion

In this chapter, we highlighted a misconception that plagues the scientific study of deep learning, namely, the eagerness to accept a specific task instance (i.e., benchmark performance) as an inferential tool for understanding a model's capabilities on other related tasks. We investigated this misconception empirically by introducing the concepts of model behavior and task generalization using a large factorial experiment as a case study where more than 40,000 models were trained on thirty related tasks. From this experiment, we found that the only model behavior that was reproducible was that of training completion rate, which we defined as the proportion of training attempts that did not encounter numerical error during training for a given model and task, and this characteristic was

architecture dependent. More importantly, we found that task accuracy was not a reliable behavior that could be attained on one task and inferred on another related task. Additionally, this behavior did not generalize across tasks when considering models trained at their maximizing hyperparameter configuration (as typically reported in the literature) or when considering task accuracy as a linear function of an architecture and training hyperparameters. While such results are hinted to in the literature, we provide evidence that benchmarks are only representative of themselves, and not a statement of model prowess on a larger family of tasks. Accordingly, we have two recommendations. We encourage engineers to invest in models with which they are familiar, and to carefully tune these models to their particular applications and computing environments. Furthermore, we encourage researchers creating new models or training methods to use our data analysis approach developed in Section 1.3 to provide a more holistic characterization of how their new methods behave across tasks.

2 CONTROLLING GRADIENT DYNAMICS IN RECURRENT NEURAL NETWORKS

In this chapter we study a class of deep learning models designed for learning sequential data patterns, the recurrent neural network (RNN). Although RNNs are standard models for learning with sequential data, RNNs are challenging to implement owing to the difficulty of managing gradient behavior during training. While several approaches have been proposed to manage gradient behavior (e.g., gated architectures, orthogonal recurrence constraint/penalty, reservoir computing), these approaches either reduce the modeling power of RNNs or require expensive training approaches. In this chapter, we propose two methods for addressing these issues. The first method we propose is a novel penalty on the intermediate partial derivatives computed during backpropagation through time. We show that our approach generalizes the orthogonal recurrence penalty; it is easy to compute and differentiate; and it readily applies to nonlinear activation functions. The second method we propose is a simple architecture modification that appears to improve learning distant time-dependent structure. We demonstrate our approach's effectiveness on modifications of benchmark training tasks that allow us to probe the impact of gradient behavior during training.

2.1 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a class of architectures designed for learning sequential patterns. The defining feature of RNNs is their use of feedback connections. This connectivity allows the model to dynamically acquire a memory over the input sequence. This feature makes RNNs an attractive modeling choice when learning sequential patterns. Accordingly, these models have been used in a variety of application areas, including: natural language understanding, financial forecasting, weather prediction, music and video generation, and robotics [71, 92, 116, 38, 29].

In Fig. 3.1 we illustrate the connectivity of a basic RNN. In general, an RNN can also be used to produce a prediction at each time point from $t = 1, \ldots, T$, but we will focus on the case where a prediction is made at the final time T (as illustrated in Figure 3.1). Moreover, while the diagram refers to the basic RNN, all RNN architectures produce a sequence of memory states, $\{x_t\}$, that are propagated forward in time and then used to produce a prediction.[1]

In comparison to the classical feed-forward network (FFN), the recurrent connectivity of RNNs offers certain advantages that the FFN does not. For example, RNNs can process sequences of input features, and these sequences can vary in length. As evidenced by the

Disclaimer: Portions of this chapter are based on a published paper that I co-authored with Vivak Patel (see [48]).

[1] While many RNN architectures can be visualized using Figure 3.1, there are some RNN architectures that use multiple memory states from previous iterates as inputs [93], but analogous diagrams can be produced.

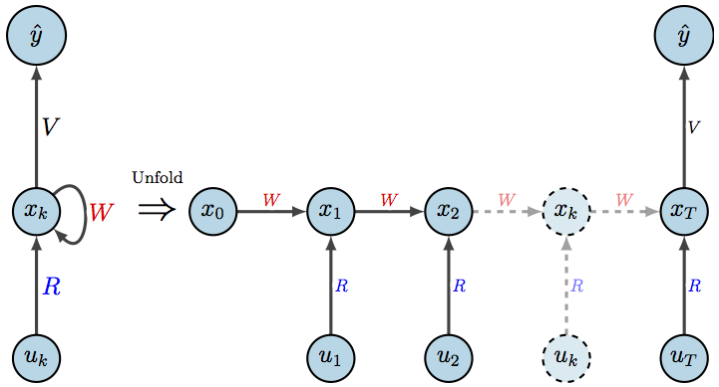


Figure 2.1: A visualization of the forward dynamics of the basic RNN (Definition 2.1).

numerous applications that use these models, this flexibility in processing makes RNNs amenable to a variety of tasks and applications.

Although RNNs have been successfully deployed in a variety of applications, these models suffer from two issues that can make them far less effective in practice. First, the sequential processing of RNNs makes parallelizing computation far less effective. As a result, these models can often be expensive to train. Second, optimizing these models such that they are able to maintain memory over long time horizons is extremely difficult. Owing to the repeated application of the transfer function (i.e., W), gradient based methods often produce error gradients that become vanishingly small or explode in size as they are propagated throughout the network. Both scenarios can lead to convergence issues during training, and difficulty contextualizing information over long time periods. This latter issue is known as the *vanishing and exploding gradient problem* (VEG) and will be the focal point of this chapter.

2.2 The Optimization Problem

Here, we formalize the *basic RNN* architecture and its corresponding (supervised) optimization problem.

Definition 2.1 (Basic Recurrent Neural Network)**.** *For a single input sequence* $\{u_1, u_2, \dots, u_T\} \subset \mathbb{R}^p$ *and arbitrary initial state* $x_0 \in \mathbb{R}^d$*, an activation function* $\sigma : \mathbb{R} \to \mathbb{R}$ *(applied component-wise), and an output function* $\phi : \mathbb{R} \to \mathbb{R}$ *(applied component-wise), the Basic Recurrent Neural Network (basic RNN) is defined by*

$$\begin{aligned} x_t &= \sigma(Wx_{t-1} + Ru_t + b_1) \qquad t = 1, \dots, T \\ \hat{y} &= \phi(Vx_T + b_2) \end{aligned} \tag{2.1}$$

where $W \in \mathbb{R}^{d \times d}$ *is the recurrent weight matrix;* d *is the recurrent dimension;* $R \in \mathbb{R}^{d \times p}$*;* $b_1 \in \mathbb{R}^d$*;* $V \in \mathbb{R}^{l \times d}$*;* l *is the output dimension;* $b_2 \in \mathbb{R}^l$*; for any* $t \in \{0, \ldots, T\}$*,* $x_t \in \mathbb{R}^d$ *is the memory state at time* t*; and* $\hat{y} \in \mathbb{R}^l$ *is the prediction.*

The optimization problem associated with this model can be formulated following the empirical risk minimization procedure where a loss function, F, is minimized over a set of N input-output pairs, $\{(u_1^i, u_2^i, \ldots, u_T^i, y^i) : i = 1, 2, \ldots, N\}$, subject to the architecture dynamics (i.e., (2.1)).

$$\begin{aligned} \arg\min_{\theta \in \Theta} \quad & \frac{1}{N} \sum_{i=1}^{N} F(y^i, \hat{y}^i) \\ \text{subject to} \quad & x_t^i = \sigma(W x_{t-1}^i + R u_t^i + b_1) \qquad t = 1, \ldots, T \\ & \hat{y}^i = \phi(V x_T^i + b_2), \end{aligned} \tag{2.2}$$

where the initial state, x_0, is typically initialized to the zero-vector and learned as an additional parameter during training.

Typically, solving (2.2) involves computing the partial derivative of the cost with respect to each of the parameters. For recurrent parameters, this derivative will be a summation over gradient information collected at each time step. For example, to compute the gradient with respect to W for a single observation pair, we use the chain rule to derive

$$\frac{\partial F}{\partial W} = \sum_{t=1}^{T} \left(\frac{\partial x_t}{\partial W}\right)' \left(\frac{\partial x_{t+1}}{\partial x_t}\right)' \cdots \left(\frac{\partial \hat{y}}{\partial x_T}\right)' \frac{\partial F(y, \hat{y})}{\partial \hat{y}}, \tag{2.3}$$

where T refers to the time horizon, and the apostrophe operator denotes the transpose of a matrix. Unfortunately, as seen in (2.3), computing the gradient requires the computation and product of a sequence of Jacobian matrices, which can become prohibitively expensive as the dimension of the states or parameters grow.

To avoid this, the adjoint method can be used. The adjoint method efficiently addresses this expense by proceeding by duality, formulating a so-called adjoint system or "backward" discrete-time system, and leveraging the solution vectors—called the adjoint states or adjoint variables—of the adjoint system to compute derivatives using only matrix-vector products [58].

The adjoint system can be derived through the Lagrangian. This derivation considers the state dynamics as additional constraints under study and introduces a set of adjoint variables (Lagrange multipliers), $\lambda_1, \ldots, \lambda_T$, corresponding to the state equation at each

position in the state trajectory. Under this framework, the Lagrangian is defined as

$$\mathcal{L} = F(y, \hat{y}) + \lambda_T'(\hat{y} - \phi(Vx_T + b_2)) + \sum_{t=0}^{T-1} \lambda_t' \Big(x_{t+1} - \sigma(Wx_t + Ru_{t+1} + b_1) \Big). \tag{2.4}$$

The Lagrangian's first term is interpreted as the cost incurred from predicting $\hat{y}$, and the Lagrangian's summation term is interpreted as the cost of abiding by the forward system dynamics. Note, the Lagrangian's partial derivatives with respect to the adjoint state are zero (i.e., $\frac{\partial \mathcal{L}}{\partial \lambda_i} = 0$) if and only if the forward, discrete-time dynamics are satisfied. Analogously, the Lagrangian's partial derivatives with respect to the state variables,

$$\frac{\partial \mathcal{L}}{\partial x_t} = 0 \in \mathbb{R}^d, \tag{2.5}$$

generate the backward or adjoint system, which, when satisfied, can be used to compute the derivative of the Lagrangian with respect to the parameters.

The Adjoint System of Backpropagation. Given an RNN, an example $(\{u_1, \ldots, u_T\}, y)$, and a loss function $F(y, \hat{y})$, the usual backpropagation calculation produces the adjoints $\{\lambda_t : t = 1, \ldots, T\}$, defined by

$$\lambda_t = \begin{cases} -V'\phi^{(1)}(Vx_t + b_2)' \frac{\partial F}{\partial \hat{y}} & t = T \\ W'\sigma^{(1)}(Wx_t + Ru_{t+1} + b_1)'\lambda_{t+1} & t < T, \end{cases} \tag{2.6}$$

where $\phi^{(1)}$ and $\sigma^{(1)}$ represent the Jacobians of ϕ and σ, respectively.

Using these calculations, we can express the classical backpropagation algorithm.

Algorithm 1: Backpropagation (Adjoint Method) Algorithm

Data: Training Examples; Loss Function F; Initial Parameter Set, Θ; Activation Function, σ; Output Function ϕ; (Stochastic) Gradient-based Training Algorithm; User-Specified Stopping Condition;

Result: Backpropagation (adjoint method) Trained Recurrent Neural Network

while *User-Specified Stopping Condition* **do**
  if *Stochastic Algorithm* **then**
    Sample mini-batch from training examples in update set;
  else
    Use all training examples in update set;
  end
  for *Each example in Update Set* **do**
    Compute $\{x_t\}$ and $\hat{y}$ using (2.1);
    Compute $\{\lambda_t\}$ using (2.6);
    Compute gradient of F with respect to the parameters;
  end
  Using the training algorithm and mean gradient, update the parameters;
end

2.3 The Vanishing and Exploding Gradient Problem

The vanishing and exploding gradient problem (VEG) is characterized by the rapid (exponential) change in magnitude of the error gradient as it is propagated back throughout the network. Specifically, consider a T-horizon sequential task, where x_t is an intermediate hidden state, and F the objective being minimized. Then the VEG problem is commonly characterized by the rapid decay or growth of $\frac{\partial F}{\partial x_t}$ as the number of layers (i.e., recurrent steps), T, grows. In terms of our derivation of the backpropagation algorithm, this characterization of VEG can be analogously stated in terms of the adjoint states, λ_t. Namely, the vanishing case is described by the rapid decay in size of $||\lambda_t||_2$ as T grows, and the exploding case by the rapid increase in size of $||\lambda_t||_2$ as T grows.

To understand the impact of VEG on the training process, consider computing the recurrent gradient using algorithm 1 (i.e., backpropagation).

$$\nabla_W F(y, \hat{y}) = -\sum_{t=1}^{T} \sigma^{(1)}(Wx_{t-1} + Ru_t + b_1)' \lambda_t x'_{t-1}. \tag{2.7}$$

First, notice that in (2.7), each time point contributes equally to computing the gradients for the recurrent weight matrix. Thus, at the beginning of training, each time point can influence the prediction quality of the RNN equally. Given that each time point corresponds to an input vector u_t, then each input vector can equally influence the prediction quality of the RNN at the beginning of training, and this would only be modified if the minimization induces the parameters to change the impact of each input.

However, owing to the nonlinearity of σ and the matrix-vector products in (2.6), the adjoint variables, λ_t, tend to zero as $t \downarrow 1$ (note, they can also explode towards infinity, but this is observed less frequently in practice when suitable initializations are used). As a result, the early inputs (i.e., those observed closest to $t = 1$) have a substantially diminished or negligible influence on the choice of parameters, and, consequently, have a negligible influence on the predictions that are being made. If there is no prior knowledge about the relative importance of the inputs, then such a phenomenon would bias the RNN and could result in poorer prediction power.

2.4 Proposed Methods for Mitigating VEG

In practice, if a model experiences VEG during optimization, then gradient credit assignment becomes biased, and the RNN fails to learn or does so poorly. Accordingly, there have been numerous techniques proposed to address this problem. Below, we discuss the main mechanisms that these methodologies exploit to address VEG.

2.4.1 Gating Mechanisms

One popular strategy for improving gradient flow in RNNs is the use of gating mechanisms. This strategy is the basis for the popular long short-term memory (LSTM) architecture [42]. While there have been a variety of proposed architectures that modify the original LSTM (e.g., see [21, 35, 20]), we will focus on the LSTM to explain the strategy that gated architectures exploit to better manage gradient flow in RNNs.

The LSTM RNN is composed of a cell state, and three gating mechanisms: an input gate, an output gate and a forget gate.[2]

[2]The number of gates can vary based on the specific architecture, but for the classical LSTM, there are three gating mechanisms.

Definition 2.2 (Long short-term memory)**.** *The long short-term memory* (*LSTM*) *cell is equipped with parameterized gating mechanisms,*

$$\begin{aligned} f_t =& \sigma_g(W_f x_{t-1} + R_f u_t + b_f) \qquad (\textit{forget gate}) \\ i_t =& \sigma_g(W_i x_{t-1} + R_i u_t + b_i) \qquad (\textit{input gate}) \\ o_t =& \sigma_g(W_o x_{t-1} + R_o u_t + b_o) \qquad (\textit{output gate}) \end{aligned} \tag{2.8}$$

where σ_g *is the sigmoid activation that maps to* $[0, 1]$*. Along with these gating units, the LSTM cell develops an input state,* $\tilde{c}_t$*, as,*

$$\tilde{c}_t = \sigma_h(W_c x_{t-1} + R_c u_t + b_c) \tag{2.9}$$

where σ_h *is the hyperbolic tangent* (*element-wise*) *activation that maps to* $[-1, 1]$*. Using these components, the cell state* (*memory*)*,* c_t*, and recurrent state,* x_t*, are developed as,*

$$\begin{aligned} c_t =& f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\ x_t =& o_t \odot \sigma_h(c_t) \end{aligned} \tag{2.10}$$

where $\odot$ *denotes the element-wise Hadamard product.*

As Def. 2.2 illustrates, the dynamics of the LSTM cell are significantly more complex than the basic RNN, however, each component can be explained in a simple manner by considering the LSTM cell as a computer equipped with a memory, and delete, write and read operations. Namely, c_t, is the system's memory; f_t is a learned weighting that controls what components of memory should be deleted at each computational step (i.e., $f_t \odot c_{t-1}$); i_t is a learned weighting that controls what components of the input should be written to memory (i.e., $i_t \odot \tilde{c}_t$); and o_t is a learned weighting that controls which memory components should be read at each computational step (i.e., $o_t \odot \sigma_h(c_t)$).

The key component of this design that allows for better training dynamics over long time periods is the linearity of the memory cell, i.e.,

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t. \tag{2.11}$$

This linearity is advantageous when considering the gradient of the memory cell through time is then just the forget gate weighting,

$$\frac{\partial c_t}{\partial c_{t-1}} = \frac{\partial}{\partial c_{t-1}} f_t \odot c_{t-1} = f_t. \tag{2.12}$$

In the case were $f_t = \mathbf{1}$, no memory is deleted over time and the error gradient remains constant through time. As a result, information written to c_t will remain in memory indefinitely, and will only be altered if f_t induces this change.

In many cases, LSTMs (and more generally, gated RNNs) have demonstrated an ability to better mitigate the gradient flow issue in RNN optimization — especially with initializations designed for the task's temporal horizon [97]. However, almost paradoxically, gated architectures lose memory capacity in comparison to the basic RNNs [22], and are susceptible to gradient explosion and gradient decay over long time horizons [3, 104]. Taking these observations together, gated architectures appear to be more amenable to current training techniques (up to a point), despite having a smaller learning capacity in comparison to the basic RNN. Additionally, the gating mechanisms of these architectures requires additional parameterization in comparison to the basic RNN, further exacerbating the already costly training time of these models, and further contributes to the environmental costs of machine learning [47, 96].

2.4.2 Orthogonal and Unitary Recurrent Matrices

Another popular strategy for addressing VEG in RNNs is by using orthogonal (unitary) matrices to either initialize or parameterize the recurrent (hidden-to-hidden) transition matrix. The motivation for these strategies can be seen through the adjoint system derived in the previous section (see 2.6), and reproduced here for readability,

$$\lambda_t = \begin{cases} -V'\phi^{(1)}(Vx_t + b_2)'\frac{\partial E}{\partial \hat{y}} & t = T \\ W'\sigma^{(1)}(Wx_t + Ru_{t+1} + b_1)'\lambda_{t+1} & t < T. \end{cases} \tag{2.13}$$

Recall, that VEG can be characterized by the exponential vanishing or exploding of $\|\lambda_t\|_2$ through time. Hence, the size of the recurrent matrix, W, plays a critical role in determining whether the gradient dynamics will explode or vanish in time. Namely, if $\|W\|_2 < 1$, then $\|\lambda_t\|_2 \to 0$ as $T \downarrow 1$ (vanishing gradient); and if $\|W\|_2 > 1$, then $\|\lambda_t\|_2 \to \infty$ as $T \downarrow 1$ (exploding gradient). Setting W to be orthogonal (i.e., $W'W = WW' = I$) takes advantage of the norm preserving properties of orthogonal matrices (i.e., $\|Wx\|_2 = \|x\|_2$), and removes the exponential impact of W on the adjoint system, (2.13).

The exploration of this strategy has included a variety of methods. In particular, initializing W as the identity matrix was studied in [60], and initializing W as an orthogonal matrix in [40]. Empirically, both of these strategies seem to improve training convergence in RNNs. Although these initializations can successfully start the model at a place where the singular values of the Jacobian are near one, it does not guarantee that these dynamics are

maintained during training. Thus, while orthogonal initialization is the default in modern deep learning frameworks (e.g., TensorFlow, Theano, PyTorch), the initialization alone is not sufficient for preventing vanishing and exploding gradients throughout training. This shortcoming of initialization schemes has lead to the development of methods that maintain orthogonal/unitary recurrent matrix structure throughout training using either reparameterization of the recurrent matrix [3, 49, 115], or gradient descent on the Stiefel manifold [111, 105].

Under certain conditions, enforcing an orthogonal recurrent matrix can provably avoid the exploding gradient case as we elucidate in Theorem 2.1 below.[3]

Theorem 2.1. *A basic RNN (defined in Definition 2.1) with an orthogonal recurrent matrix W and activation function $\sigma = tanh$ will avoid exploding gradients.*

Recall that VEG can be characterized by the sensitivities of the recurrent states, λ_t. Thus, we can show that this architecture will avoid exploding gradients by bounding $||\lambda_t||_2$.

$$
\begin{aligned}
||\lambda_t||_2 &= \left|\left| \left(\prod_{i=t}^{T-1} W' \sigma^{(1)}(Wx_t + Ru_{t+1} + b_1)' \right) \lambda_T \right|\right|_2 \\
&\leqslant ||\lambda_T||_2 \prod_{i=t}^{T-1} || \, W' \sigma^{(1)}(Wx_t + Ru_{t+1} + b_1)' \, ||_2 \\
&= ||\lambda_T||_2 \prod_{i=t}^{T-1} || \, \sigma^{(1)}(Wx_t + Ru_{t+1} + b_1) \, ||_2 \\
&\leqslant ||\lambda_T||_2
\end{aligned}
\tag{2.14}
$$

where we use the triangle inequality in line 2, the norm preserving quality of orthogonal maps (i.e., $||Wx||_2 = ||x||_2$) in line 3, and the fact that $0 \leqslant \sigma^{(1)} \leqslant 1$ for the hyperbolic tangent activation in line 4.

Unfortunately, orthogonal parameterization does not provably prevent vanishing gradients as the magnitude of the backpropagated gradients is also influenced by $\sigma^{(1)}(x) \in [0, 1]$. Hence, even for parameterizations such that $||W||_2 = 1$, the adjoint dynamics will often exhibit decaying behavior. Nevertheless, orthogonal (unitary) architectures appear to better mitigate the vanishing case as well in comparison to the basic RNN.

[3] Theorem 2.1 is taken from [3] and modified to the notation of the adjoint states and basic RNN dynamics.

2.4.3 Dynamical Systems RNNs

More recently, architectures inspired from dynamical systems have been applied to learning long-term dependencies. These architectures draw a connection between the stability of an ordinary differential equation (ODE) and the trainability of an RNN. Namely, in [17], the authors note that the solution of an ODE is stable if the real parts of the eigenvalues of the Jacobian matrix are less than or equal to zero. However, a stable solution does not guarantee the system will be able to maintain memory over a long time-horizon. Namely, when the real parts of the eigenvalues of the Jacobian matrix are less than one, then the system will converge to a fixed point, resulting in a lossy system [37]. Hence, the construction proposed aims to keep the real parts of the eigenvalues of the Jacobian matrix near zero. To accomplish this objective, methods have proposed parameterizing the recurrent matrix with an antisymmetric (skew-symmetric) matrix, $M \in \mathbb{R}^{d\times d}$, leveraging the fact that the eigenvalues of M are all imaginary. In particular, antisymmetric matrices are used in [17, 74, 65], and a mixture of antisymmetric matrices in [32]. These models benefit from being simple to implement, and empirically have shown improvement over both the basic RNN and LSTM in tasks that require maintaining memory of very long time horizons. However, owing to the nonlinearity of the activation function, these architectures can still succumb to vanishing gradients, and also require the additional tuning of a task-specific diffusion parameter in order to ensure the stability of the ODE's discretization (e.g., forward Euler discretization).

2.4.4 Regularization Methods

One less explored strategy for combatting VEG in RNNs is through regularization. In [56] the authors penalize the squared distance between successive hidden state norms,

$$\beta \frac{1}{T} \sum_{t=1}^{T} (||x_t||_2 - ||x_{t-1}||_2)^2, \tag{2.15}$$

where β is a user-specified penalty weight. The idea of this regularizer is that training in RNNs can be improved by regularizing the forward computational path of the memory state, by encouraging the memory state towards uniform size through time. This method is reversed in [80], where they regularize the backward dynamics (i.e., adjoint states).

The regularizer introduced in [80] can be seen as a specific case of a more general penalized optimization method that we develop in Section 2.6. In the next section, we use linear RNNs to motivate our optimization procedure.

2.5 Linear RNN Learning

In this section, we use linear RNNs – that is, a Recurrent Neural Network (RNN) with identity activation functions – to motivate the use of a novel regularizer that facilitates gradient flow in RNNs. Specifically, as we now formalize, we focus on the case where gradient flow is most challenging: when the predicted label is produced at the terminal layer.

Definition 2.3. *For inputs* $\{u_1, \dots, u_T\} \subset \mathbb{R}^p$ *and arbitrary initial state* $x_0 \in \mathbb{R}^d$*, a Linear Recurrent Neural Network (LRNN) is defined by*

$$\begin{aligned} x_t &= Wx_{t-1} + Ru_t + b_1 \qquad t = 1, \dots, T \\ \hat{y} &= Vx_T + b_o \end{aligned} \tag{2.16}$$

where $W \in \mathbb{R}^{d \times d}$, $R \in \mathbb{R}^{d \times p}$, $b_1 \in \mathbb{R}^d$, $V \in \mathbb{R}^{\ell \times d}$ *and* $b_o \in \mathbb{R}^\ell$. □

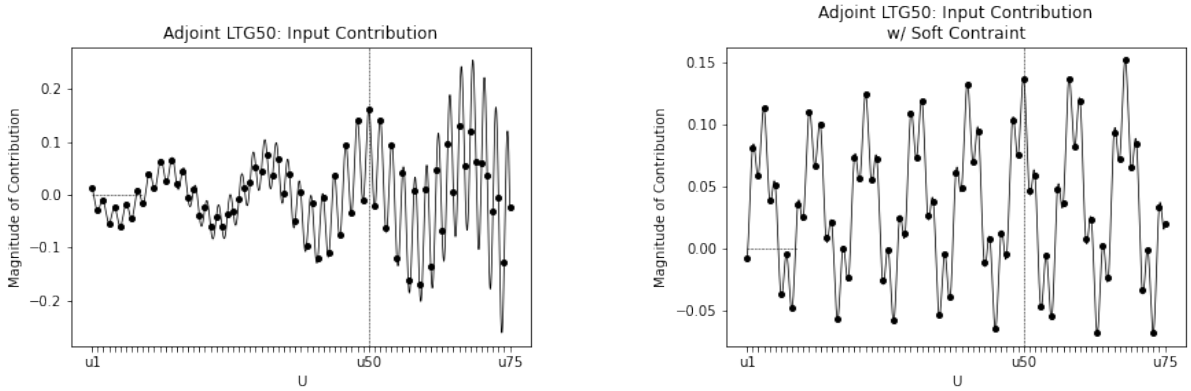
In fact, we can summarize the relationship between the prediction, $\hat{y}$, and inputs, $\{u_1, \dots, u_T\}$, by

$$\hat{y} = \sum_{t=1}^{N} VW^{T-t}Ru_t + b, \tag{2.17}$$

where $b \in \mathbb{R}^\ell$ depends on the initial state, weights and biases specified in Definition 2.3. As (2.17) demonstrates, the interaction between V and R with the generalized eigenspaces of W determines which inputs are more heavily weighted and which inputs are less weighted.

For example, suppose the rows of V and the columns of R belong to the same generalized eigenspace of W with a corresponding eigenvalue that has a magnitude greater than one. Then, increasing powers of W will correspond to increasing powers of this eigenvalue's magnitude, and would cause a greater emphasis on earlier inputs in the sequence (i.e., u_1) on the prediction. Moreover, if this choice of W is dependent on the initialization or training process rather than inherent patterns in the data, we have gradient explosion. Analogously, suppose the columns of V and R belong to the same generalized eigenspace of W with a corresponding eigenvalue that has a magnitude less than one. Then, increasing powers of W will correspond to reducing the emphasis of earlier inputs on the prediction. Again, if this is not data-driven, then we would have a vanishing gradient.

However, these are not the only two cases that can occur. Often, V and R will span several generalized eigenspaces of W, which will result in interactions between the eigenvalues corresponding to these generalized eigenspaces. As a result of these interactions between the eigenvalues of W, somewhat counterintuitively (yet well known), a LRNN can emphasize different values of the input sequences *in a nonlinear manner, even with small*



(a) Basic RNN (b) Basic RNN w/ orthogonality constraint.

Figure 2.2: We train an LRNN with internal memory dimension $d = 4$ to solve the LTG task with $T = 75$ inputs and labels determined by $\text{sign}(u_{50})$. In (a), we train the problem on the empirical risk, while, in (b), we do the same training and add a penalty to favor W to be an orthogonal matrix.

memory dimension, d, *in comparison to the time horizon,* T, as we now illustrate using a simple synthetic task.

Experiment 2.2 (Long Time Gap)**.** *The Long Time Gap (LTG) experiment is a simple synthetic binary classification task that requires the model to latch a single signal amongst a sequence of noise. The input sequence,* $\mathcal{U} = \{u_t\}_{t=1}^T$*, is a Rademacher series of length* T *– i.e., each* u_t *is an i.i.d. Rademacher random variable taking value of either* $+1$ *or* -1 *with probability* 0.5 *– and the label corresponds to sign*(u_{t^*}) *for a pre-specified sequence location* t^**. In order to successfully learn, the model must correctly distinguish the sign of* u_{t^*} *while learning to ignore the other* $T - 1$ *input elements.* □

Remark 2.1. *We cede that this task is simple in nature, but offers a controlled environment to examine the learned dynamics of RNNs.*

Using standard training routines in TensorFlow [1], we train LRNNs with internal memory dimension $d = 4$ to address LTG for $t^* = 50$ and $T = 75$. In Figure 2.2, we plot example cases of the behavior of the learned values of $VW^{T-t}R$ for $t \in [0, T-1]$ and emphasize the values in $[0, T-1] \cap \mathbb{T}$. In Figure 2.2a, we see the nonlinear behavior that can be induced even for such a small memory dimension of $d = 4$: as we would want for this LTG task, the input at u_{50} is emphasized relative to other inputs, which represents some degree of learning.

Interestingly, we find a mixed bag of performance when we induce W to be orthogonal through a soft penalty, as shown in Figure 2.2b for the same task as in Figure 2.2a (see Section A3.1 in appendix for more examples). Indeed, we should expect this type of

behavior when W is orthogonal: since the magnitude of W's eigenvalues are all one, the powers of W in (2.17) would create a periodic behavior or ergodic mixing; in turn, this periodicity or mixing would not only emphasize the t^{th} input, but others as well, as we observe in Figure 2.2b; the result is a network with less accurate predictions. We note that allowing for a soft penalty instead of a hard constraint should ease this behavior in LRNNs, but will not consistently offer improved performance when we do not have such a penalty term (see Section A3.1). Thus, inducing W to be orthogonal may address gradient flow, but at the cost of performance.

However, we can use the reasoning behind inducing W to be orthogonal to come up with a procedure that allows for a broader choice of parameters while also readily generalizing to nonlinear activation functions. Suppose we compute the gradient of the parameters, W and R, for a single example, $(\{u_1, \ldots, u_T\}, y)$, with respect to a loss function $F(y, \hat{y})$. The gradients of the parameters are given by

$$\begin{aligned} \nabla_W F(y, \hat{y}) &= -\sum_{t=1}^{T} \lambda_t x_{t-1}', \text{ and} \\ \nabla_R F(y, \hat{y}) &= -\sum_{t=1}^{T} \lambda_t u_t', \end{aligned} \tag{2.18}$$

where

$$\lambda_t = \begin{cases} -V' \frac{\partial F}{\partial \hat{y}} & t = T \\ W' \lambda_{t+1} & t < T. \end{cases} \tag{2.19}$$

Requiring W to be orthogonal induces $\|\lambda_t\|_2 = \|\lambda_{t+1}\|_2$ for $t < T$ in (2.19), which, in turn, induces each intermediate output of the RNN to weigh equally in (2.18). Thus, requiring W to be orthogonal controls gradient flow by requiring $\{\lambda_t\}$, called the adjoints in dynamical systems language, to have equal norm.

We can induce this effect directly by adding a penalty $G(\{\lambda_t\})$ to $F(y, \hat{y})$, such as penalizing the scaled variance of the adjoints' norms:

$$G(\{\lambda_t\}) = \sum_{t=1}^{T} \left[\|\lambda_t\|_2 - \left(\frac{1}{T} \sum_{i=1}^{T} \|\lambda_i\|_2 \right) \right]^2. \tag{2.20}$$

Clearly, for a LRNN, a choice of orthogonal W would certainly set the example $G(\{\lambda_t\})$ in (2.20) to zero. What is more, an alternative choice of W that preserves the norm along the row space of V would also set the example $G(\{\lambda_t\})$ in (2.20) to zero. Thus, the example penalty in (2.20) allows for a much broader set of learned parameters, even for the linear

case, in comparison to inducing W to be orthogonal.

In general, not only do penalties on the adjoints allow for a greater set of learned parameters, but they also allow for greater flexibility by changing the form of the penalties: we can make use of different norms; we can use different measures of variability; and we can add weights to the penalty. Moreover, as we shown next, we can readily extend this formulation to nonlinear activation functions, as the adjoints are always well defined so long as the activation functions are differentiable. Finally, as we also show next, penalties on the adjoints are inexpensive to compute and differentiate, do not require propagating matrix-valued or tensor-valued quantities, and can be directly integrated into popular automatic differentiation software, such as TensorFlow.

2.6 The Coadjoint Algorithm

Here, we present our procedure for addressing gradient flow problems in training Recurrent Neural Networks (RNNs). While we focus on RNNs, we note that the procedure is general and can be derived for arbitrary networks (e.g., gated recurrent networks, feed forward networks, convolution networks, transformer networks, etc.) so long as the loss function and the activation functions are twice differentiable (excepting a measure zero set). Furthermore, we can modify the choice of penalty to induce other interesting properties in the network. We will leave demonstrating these generalizations as future work.

We will present our procedure by first specifying general recurrent neural networks *for which gradient flow is particularly challenging*; then, we will derive the usual backpropagation calculation to define the adjoints; using the adjoints, we will derive our procedure; and, finally, we will state an algorithm for (stochastic) gradient-based training algorithms. Note that the definition of the general RNN and derivation of the adjoint system is also presented in Section 2.2, but we repeat this information here for better readability.

Defining a Recurrent Neural Network. We begin by recalling the definition of the basic RNN (i.e., Def. 2.1).

Definition 2.1 (Basic Recurrent Neural Network)**.** *For a single input sequence* $\{u_1, u_2, \dots, u_T\} \subset \mathbb{R}^p$ *and arbitrary initial state* $x_0 \in \mathbb{R}^d$*, an activation function* $\sigma : \mathbb{R} \to \mathbb{R}$ (*applied component-wise*)*, and an output function* $\phi : \mathbb{R} \to \mathbb{R}$ (*applied component-wise*)*, the Basic Recurrent Neural Network* (*basic RNN*) *is defined by*

$$\begin{aligned} x_t &= \sigma(W x_{t-1} + R u_t + b_1) \qquad t = 1, \dots, T \\ \hat{y} &= \phi(V x_T + b_2) \end{aligned} \tag{2.1}$$

where $W \in \mathbb{R}^{d \times d}$ *is the recurrent weight matrix;* d *is the recurrent dimension;* $R \in \mathbb{R}^{d \times p}$*;* $b_1 \in \mathbb{R}^d$*;* $V \in \mathbb{R}^{l \times d}$*;* l *is the output dimension;* $b_2 \in \mathbb{R}^l$*; for any* $t \in \{0, \dots, T\}$*,* $x_t \in \mathbb{R}^d$ *is the memory state at time* t*; and* $\hat{y} \in \mathbb{R}^l$ *is the prediction.*

As per usual, we can let T vary from example to example and still maintain the same RNN.

The Adjoints of Backpropagation. Given an RNN, an example $(\{u_1, \dots, u_T\}, y)$, and a loss function $F(y, \hat{y})$, the usual backpropagation calculation produces the adjoints $\{\lambda_t : t = 1, \dots, T\}$, defined by

$$\lambda_t = \begin{cases} -V'\phi^{(1)}(Vx_t + b_0)' \frac{\partial F}{\partial \hat{y}} & t = T \\ W'\sigma^{(1)}(Wx_t + Ru_{t+1} + b_1)'\lambda_{t+1} & t < T, \end{cases} \tag{2.21}$$

where $\phi^{(1)}$ and $\sigma^{(1)}$ represent the Jacobians of ϕ and σ, respectively. As is well known, the adjoints are then used to compute the gradients with respect to the parameters (i.e., W, R, b_1, V, b_0) in a manner analogous to (2.18); for example,

$$\nabla_W F(y, \hat{y}) = -\sum_{t=1}^{T} \sigma^{(1)}(Wx_{t-1} + Ru_t + b_1)'\lambda_t x_{t-1}'. \tag{2.22}$$

Deriving the Coadjoints from Penalties on the Adjoints. Now, given an RNN, an example $(\{u_1, \dots, u_T\}, y)$, a loss function $F(y, \hat{y})$, and an adjoint penalty $G(\lambda_1, \dots, \lambda_T)$, we can apply backpropagation to the sum of F and G following Lagrangian formalism [62] (see Section A2 of the appendix for details). Applying backpropagation produces, what we term, the *forward coadjoints*, $\{\gamma_t\}$, and the *backward coadjoints*, $\{\alpha_t\}$, defined by

$$\gamma_t = \begin{cases} -\frac{\partial G}{\partial \lambda_1} & t = 1 \\ \sigma^{(1)}(Wx_{t-1} + Ru_t + b_1)W\gamma_{t-1} - \frac{\partial G}{\partial \lambda_t} & t > 1, \end{cases} \tag{2.23}$$

and

$$\alpha_t = \begin{cases} V'\phi^{(1)}(Vx_T + b_0)\dfrac{\partial F}{\partial \hat{y}} \\ \quad + V'\phi^{(1)}(Vx_T + b_0)'\dfrac{\partial^2 F}{\partial^2 \hat{y}}\phi^{(1)}(Vx_T + b_0)V\gamma_T & t = T \\ \quad - V'\left(\displaystyle\sum_{k=1}^{\ell}\frac{\partial F}{\partial \hat{y}_{[k]}}\phi^{(2)}_{[k]}(Vx_T + b_0)'\right)V\gamma_T \\ W'\sigma^{(1)}(Wx_t + Ru_{t+1} + b_1)'\alpha_{t+1} \\ \quad + W'\left(\displaystyle\sum_{k=1}^{d}\lambda_{t+1}[k]\sigma^{(2)}_{[k]}(Wx_t + Ru_{t+1} + b_1)'\right)W\gamma_t & t < T, \end{cases} \tag{2.24}$$

where the $[k]$ represents the k^{th} component of a given quantity; and $\phi^{(2)}_{[k]}$ and $\sigma^{(2)}_{[k]}$ represent the Hessians of $\phi_{[k]}$ and $\sigma_{[k]}$, respectively. Now, we can use the coadjoints to compute the gradient of the $F + G$ with respect to the parameters; for example, the gradient of $F + G$ with respect to W is

$$\begin{aligned} -\sum_{t=1}^{T} & \sigma^{(1)}(Wx_{t-1} + Ru_t + b_1)'\alpha_t x'_{t-1} \\ & + \gamma_t\lambda'_{t+1}\sigma^{(1)}(Wx_t + Ru_{t+1} + b_1) \\ & + W\gamma_t x'_t\left(\sum_{k=1}^{d}\lambda_{t+1}[k]\sigma^{(2)}_{[k]}(Wx_t + Ru_{t+1} + b_1)\right). \end{aligned} \tag{2.25}$$

Remark 2.2. *When ϕ and σ are applied component-wise, as is typically the case, the equations for forward and backward coadjoints simplify greatly. Even when ϕ and σ are applied to the entire argument, we can simplify the notation using tensor notation, but we have opted to use matrix notation for clarity.*

We note that in the above formulation, we require F, ϕ, and σ to be twice differentiable and we need G to be once differentiable. Generally, this is not a problem for most common loss functions, activation functions and our anticipated choices of G, at least off of a set of measure zero.

The Coadjoint Algorithm. Using the above calculations, we can state a (stochastic) gradient-based training algorithm, Algorithm 2. We now state several properties of this procedure. First, Algorithm 2 elucidates the naming of the forward and backward coadjoints: the forward coadjoints are analogous to evaluating the RNN forward in time, while

Algorithm 2: Coadjoint Algorithm

Data: Training Examples; Loss Function F; Adjoint Penalty G; Initial Parameter Set, Θ; Activation Function, σ; Output Function ϕ; (Stochastic) Gradient-based Training Algorithm; User-Specified Stopping Condition;
Result: Coadjoint Trained Recurrent Neural Network
while *User-Specified Stopping Condition* **do**
  if *Stochastic Algorithm* **then**
    Sample mini-batch from training examples in update set;
  else
    Use all training examples in update set;
  end
  for *Each example in Update Set* **do**
    Compute $\{x_t\}$ and $\hat{y}$ using (**??**);
    Compute $\{\lambda_t\}$ using (2.21);
    Compute $\{\gamma_t\}$ using (2.23);
    Compute $\{\alpha_t\}$ using (2.24);
    Compute gradient of $F + G$ with respect to the parameters;
  end
  Using the training algorithm and mean gradient, update the parameters;
end

the backward coadjoints are analogous to the adjoints, which are calculated backward in time. Second, in terms of memory costs, Algorithm 2 requires twice as much storage as the regular backpropagation algorithm, as we must now also compute $\{\gamma_t\}$ and $\{\alpha_t\}$; this additional storage cost may be high for RNNs with large d and T, but can be ameliorated using checkpoints [36]. Third, in terms of operations, Algorithm 2 requires twice as much computation as the standard backpropagation algorithm. Overall, Algorithm 2 ought to be a few times slower than standard backpropagation, given an efficient implementation.

Addressing the Vanishing Gradient RNN training difficulty primarily stems from unequal and biased credit assignment of terms, $\nabla_W F^{(t)}$, in (2.22). In particular, the vanishing gradient problem can be summarized by

$$\left\|\nabla_W F^{(t)}\right\| \approx 0. \tag{2.26}$$

As a result the t^{th} sequence position becomes negligible with respect to the optimization path. As we've demonstrated, the adjoint state λ_t characterizes the magnitude of each term in (2.22). In an analogous manner, the coadjoint procedure would incur the same loss of signal if $\left\|\nabla_W (F + G)^{(t)}\right\| \approx 0$. We argue, albeit informally, that this will not occur if

optimization suffers from gradient decay. If adjoints have become vanishingly small at time t, then $\nabla_W (F+G)^{(t)} \approx \nabla_W G^{(t)}$. Adopting the scaled variance adjoint penalty, $\nabla_W G^{(t)}$ is zero only if all adjoints are equal in size. In the decayed setting, this cannot be true as $\|\lambda_t\| \approx 0$ implying $\left\|\nabla_W (F+G)^{(t)}\right\| > 0$.

2.6.1 Visualizing the Coadjoint Regularizer

In the previous section, we saw that LRNNs learned the LTG task by emphasizing label relevant input tokens while down-weighting noise tokens. Furthermore, when we encouraged the recurrent matrix towards orthogonality through a soft constraint we observed that the dynamics induced from this constraint resulted in a more uniform emphasis applied to all input tokens. The idea here being that the orthogonality constraint of the recurrent matrix prevents the gradient envelope from decaying or exploding through time. Unfortunately, in order to obtain this more uniform weighting of the gradient field, the forward dynamics are reduced in their expressivity (see Fig. 2.2).

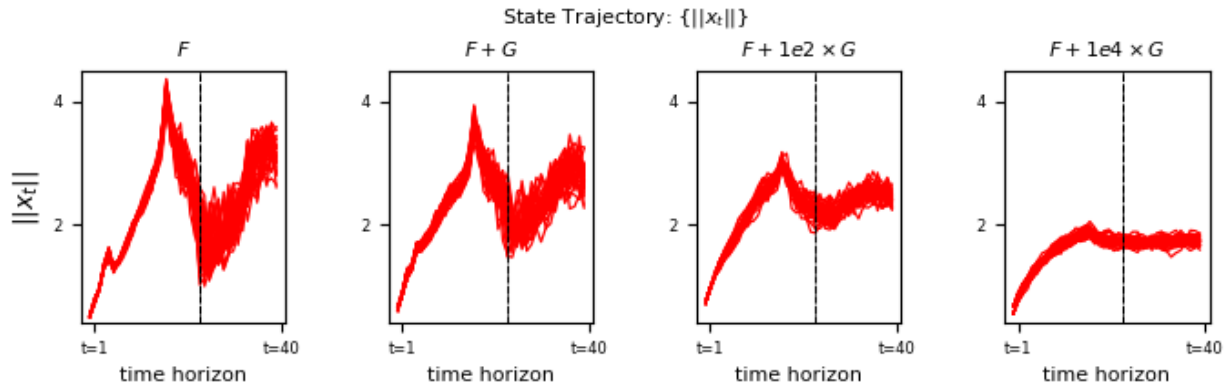
In comparison, the coadjoint algorithm does not make any explicit restriction on the forward dynamics or parameterization of the recurrent matrix. Instead, the scaled variance penalty in (2.20) regularizes the gradients of the model towards dynamics that are uniform in time. We illustrate this using the LTG task in Fig. 2.3 where we train RNNs (nonlinear) with varying regularization weights.

In Fig. 2.3 we process a single batch of training samples and plot $||x_t||_2$ from $t = 1, \ldots, T$ to visualize the learned state trajectory of the model (top red panels). The middle panels of Fig. 2.3 correspond to the adjoint states, $||\lambda_t||_2$; and the bottom panel corresponds to the adjoint states after being normalized by $\frac{||\lambda_t||}{||\Lambda||}$ where $||\Lambda|| = \max_{t\in[1,T]}\{||\lambda_t||\}$.

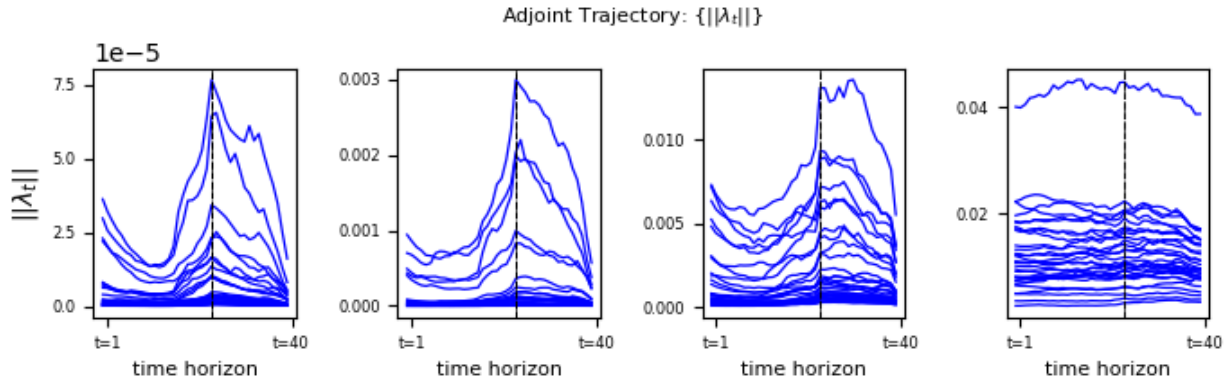
Each of the trained models in Fig. 2.3 are able to achieve perfect accuracy on the LTG task, but do so by learning very different dynamics. As the coadjoint penalty weight is increased from $0 \rightarrow 10^4$ (left-to-right in Fig. 2.3), the adjoint states become more uniformly sized over time indices. This behavior suggests that it may be useful in controlling gradient dynamics through time, and mitigating the impact of gradient decay/explosion. We experimentally test this hypothesis in the next section using a handful of benchmark tasks used to evaluate the ability of RNNs to learn long-term time dependencies.

2.7 Experimental Results on the Coadjoint Method

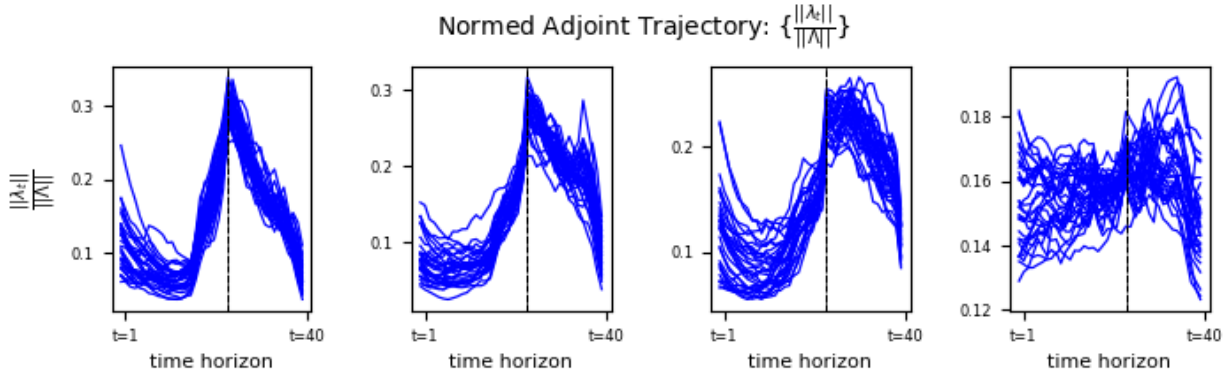
Fundamentally, we wish to evaluate whether the coadjoint method can improve RNN learning relative to the aforementioned approaches when gradient flow is a problem. To



(a) Recurrent State Trajectory



(b) Adjoint State Trajectory



(c) Normed Adjoint State Trajectory

Figure 2.3: Basic RNNs trained on the LTG task (T = 40) with varying levels of the coadjoint regularizer applied. The panels from left-to-right indicate the level of regularizer applied to the training process (left panel is no penalty (i.e., backpropagation) and the left is largest (10^4 coefficient applied to G). The top panel illustrates the state trajectory for each model, the middle panel depicts the adjoint trajectory and the bottom illustrates the normalized adjoint trajectory.

this end, we modify two benchmark tasks into a family of tasks that will exacerbate a common gradient flow problem, namely gradient decay. As an overview, we modify the MNIST and Fashion MNIST classification tasks to (i) turn them into a temporal problem by feeding in one row of the image to the RNN per time step (resulting in 28 inputs), and (ii) appending the data sequence (i.e., the rows of the images) with varying numbers of rows composed of independent standard Gaussian noise to extend the distance between the output layer and the information in the image. Note, this second modification is essential: when one of the aforementioned approaches for addressing gradient flow is still able to learn as we increase the amount of noise appended to the sequence, this is an indication that the approach is successfully able to address the gradient decay in the tasks.

Experiment 2.3 (Noise Padded MNIST and Fashion MNIST)**.** *Inspired by the padded experiments of* [17]*, we extend the temporal horizon of the input sequences by appending noise to the input. We process the original MNIST and Fashion MNIST images* (*which are* 28×28) row-by-row *and append iid* $\mathtt{Unif}(0,1)$ *noise such that the total time horizon,* T*, is a multiple of the original image's number of rows. For example a Pad2x experiment would have a total time horizon of* $\mathsf{T} = 56$ *where the first 28 elements are the rows of the original image and the latter 28 elements are vectors in* $\mathbb{R}^{28}$ *where each coordinate is an i.i.d. standard uniform random variable. For each dataset we perform 0x* (*baseline with no padding*)*, 4x, 8x and 12x padded experiments. We admit that* (*in theory*) *these time horizons are not prohibitively long for a basic RNN to capture. However, by "hiding" the original image in the beginning of the input sequence the network must learn to ignore the noise and retain the signal over the entire horizon, making the task much more difficult in comparison to tasks where there is signal presented at each time point.*

We will briefly mention several salient points about the experimental design. First, in modifying the tasks, the amount of noise that is appended is a multiple of the number of rows of the original image: so a 4x padded task is appended with 112 (4×28) rows of noise, while an 8x padded task is appended with 224 (8×28) rows of noise. In total, we look at 0x (no noise appended), 4x, 8x, and 12x padded variants of the MNIST and Fashion MNIST tasks. Second, in terms of the learning approaches for addressing gradient flow, we consider the five methods described in Table 2.1. Finally, in terms of the recurrent dimension, we experiment with recurrent dimensions in the set $\mathtt{d} = \{20, 40, 60, 80, 100\}$. We repeat each combination of these experimental factors ten times, and record the median and median absolute deviation (MAD) of test accuracy for each architecture, recurrent dimension and padded task in Table 2.2, and use Fig. 2.4 and Fig. 2.5 to visualize the results.

Method	Details
Adjoint	RNN trained with the adjoint method, i.e., backpropagation
Coadjoint	RNN trained with algorithm 2 and adjoint penalty (2.20)
Constrained	RNN with soft penalty $\|W'W - \mathrm{I}\|_2^2 + \|WW' - \mathrm{I}\|_2^2$
LSTM	Long short-term memory network
RC	RNN trained in the reservoir computer framework

Table 2.1: Recurrent models used in experiments.

For the Pad0x experiments (no additional padding), gradient decay was not observed which resulted in approximately similar performance between RNN methods. LSTM was able to consistently outperform the RNN methods in this setting. However, for the padded experiments LSTM was the least effective method. Interestingly, we observed that LSTM would periodically minimize training loss while failing to make any improvement on the validation/test sets. This occurrence could be due to LSTM learning a pattern inherent to the noise and not the true signal; implying that LSTM is directionally learning in that it first learns the latter sequence elements and then as training progresses slowly incorporates information from the earlier portions of the input sequence. This aspect lends favor to the notion that LSTM training could be improved by the coadjoint algorithm and will be the focus of subsequent work.

For smaller dimensions ($d \leqslant 40$) all of the methods had trouble consistently learning in the padded settings. In particular, the coadjoint trained RNN periodically would fail during training due to numerical error. This problem was mitigated to an extent with increasing of dimension $d > 60$ and decreasing learning rate. Furthermore, these numerical issues were not consistent in that coadjoint RNN learning was sensitive to the initialization.

In the Pad8x regime, the two datasets proved to be very different in the ease of learning. For Fashion MNIST Pad8x, all RNN methods were able to learn and performed relatively similarly given that the recurrent dimension was sufficiently large ($d \geqslant 80$). Suggesting that gradient decay was not as detrimental for the Fashion MNIST dataset. On the other hand, MNIST Pad8x proved to be a difficult task; only the coadjoint RNN was able to generate a median accuracy greater than baseline, regardless of recurrent dimension.

For the Pad12x experiments, only the coadjoint RNN was able to achieve a median accuracy better than baseline — although these models needed additional initializations due to numerical difficulties. Of note, the constrained RNN was able to generate a few initializations that learned to the degree of the coadjoint RNN but was not consistent in that the vast majority of initializations did no better than the baseline accuracy.

	Method	MNIST d = 20	MNIST d = 40	MNIST d = 60	MNIST d = 80	MNIST d = 100	Fashion MNIST d = 20	Fashion MNIST d = 40	Fashion MNIST d = 60	Fashion MNIST d = 80	Fashion MNIST d = 100
Pad0x	RNN adjoint	84.5 ± 0.425	90.3 ± 0.17	0.91.4 ± 0.10	91.7 ± 0.22	91.8 ± 0.10	77.8 ± 0.71	81.8 ± 0.18	82.9 ± 0.17	83.5 ± 0.14	83.6 ± 0.15
	RNN coadjoint	84.8 ± 1.03	90.3 ± 0.12	91.4 ± 0.15	91.7 ± 0.10	91.8 ± 0.17	77.8 ± 0.29	81.8 ± 0.25	82.9 ± 0.20	83.5 ± 0.23	83.6 ± 0.15
	RNN constrained	84.8 ± 0.43	90.3 ± 0.21	91.4 ± 0.11	91.8 ± 0.12	91.8 ± 0.12	77.9 ± 0.89	81.7 ± 0.26	82.9 ± 0.22	83.4 ± 0.13	83.6 ± 0.13
	LSTM	**96.7 ± 0.23**	**98.0 ± 0.12**	**98.4 ± 0.10**	**98.5 ± 0.10**	**98.6 ± 0.10**	**85.6 ± 0.23**	**87.5 ± 0.13**	**88.5 ± 0.10**	**88.7 ± 0.08**	**89.0 ± 0.15**
	RNN RC	84.7 ± 0.71	90.1 ± 0.24	91.4 ± 0.10	91.6 ± 0.11	91.6 ± 0.11	77.8.0 ± 0.77	81.9 ± 0.30	82.9 ± 0.31	83.5 ± 0.16	83.8 ± 0.07
Pad4x	RNN adjoint	10.8 ± 0.14	10.8 ± 0.20	45.7 ± 22.54	70.0 ± 5.24	77.8 ± 1.74	10.2 ± 0.25	45.2 ± 10.3	64.1 ± 3.7	68.1 ± 2.9	71.4 ± 2.0
	RNN coadjoint	9.80 ± 0.0	10.8 ± 0.15	54.4 ± 12.73	**72.7 ± 10.57**	**81.5 ± 1.52**	10.1 ± 0.13	**53.3 ± 12.1**	64.8 ± 3.1	**70.5 ± 3.2**	**72.6 ± 1.4**
	RNN constrained	10.9 ± 0.14	10.6 ± 0.18	42.3 ± 31.68	68.8 ± 5.94	81.1 ± 0.94	10.0 ± 0.23	50.1 ± 10.1	58.7 ± 7.8	67.6 ± 1.9	70.8 ± 1.7
	LSTM	10.3 ± 0.23	10.3 ± 0.08	9.97 ± 0.12	10.1 ± 0.15	9.97 ± 0.19	10.1 ± 0.25	10.1 ± 0.28	10.2 ± 0.19	18.1 ± 8.1	70.9 ± 8.1
	RNN RC	10.5 ± 0.17	10.7 ± 0.18	**58.4 ± 4.68**	71.3 ± 5.68	78.8 ± 3.47	9.92 ± 0.25	43.4 ± 23.2	**66.8 ± 0.97**	68.1 ± 1.8	72.0 ± 1.4
Pad8x	RNN adjoint	10.8 ± 0.24	10.4 ± 0.16	10.2 ± 0.10	10.4 ± 0.30	10.1 ± 19.4	9.9 ± 0.21	9.9 ± 0.19	9.8 ± 0.19	30.1 ± 20.4	51.0 ± 3.00
	RNN coadjoint	9.80 ± 0.0	9.80 ± 0.0	**36.2 ± 12.1**	**47.2 ± 2.9**	**50.7 ± 6.49**	10.0 ± 0.00	10.0 ± 0.00	**33.9 ± 20.8**	**52.3 ± 5.6**	**61.0 ± 5.3**
	RNN constrained	10.6 ± 0.15	10.3 ± 0.26	10.2 ± 0.21	10.2 ± 0.11	10.1 ± 0.23	10.0.0 ± 0.15	10.1 ± 0.35	10.2 ± 0.69	50.6 ± 7.4	56.8 ± 6.5
	LSTM	10.6 ± 0.08	10.4 ± 0.17	10.1 ± 0.15	10.1 ± 0.20	10.1 ± 0.20	10.3 ± 0.10	10.2 ± 0.09	10.0 ± 0.13	10.1 ± 0.21	10.0 ± 0.25
	RNN RC	10.7 ± 0.06	10.5 ± 0.10	10.2 ± 0.12	9.93 ± 0.11	10.4 ± 0.42	9.9 ± 0.21	9.7 ± 0.36	32.4 ± 21.5	26.6 ± 17.21	55.6 ± 9.6
Pad12x	RNN adjoint	-	-	-	9.9 ± 0.01	10.0 ± 0.19	-	-	-	10.0 ± 0.24	10.2 ± 0.16
	RNN coadjoint	-	-	-	**29.0 ± 4.96**	**42.2 ± 8.63**	-	-	-	**32.5 ± 17.1**	**47.7 ± 1.52**
	RNN constrained	-	-	-	10.1 ± 0.10	10.1 ± 0.12	-	-	-	10.2 ± 0.18	9.9 ± 0.16
	LSTM	-	-	-	10.1 ± 0.31	10.1 ± 0.15	-	-	-	10.0 ± 0.19	10.1 ± 0.17
	RNN RC	-	-	-	10.0 ± 0.16	10.1 ± 0.13	-	-	-	9.9 ± 0.08	10.1 ± 0.12

Table 2.2: MNIST and Fashion MNIST Pad0x, Pad4x, Pad8x and Pad12x median classification accuracy and MAD over ten initializations for five RNN methods.

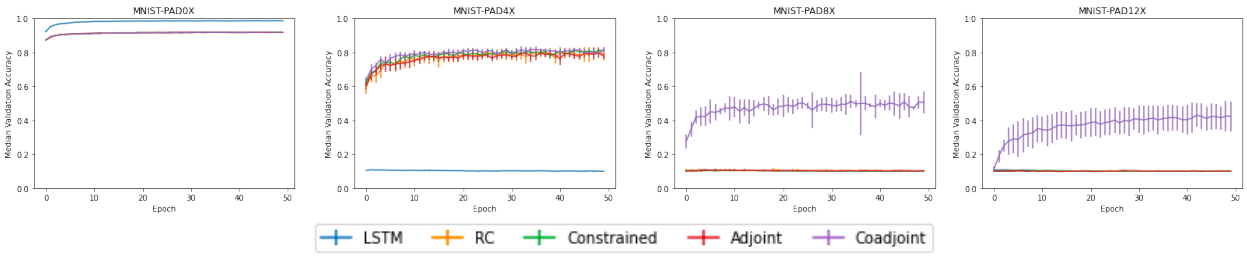


Figure 2.4: MNIST: Median validation accuracy for recurrent architectures with hidden dimension d = 100 for padded experiments 0X, 4X, 8X and 12X, respectivley.

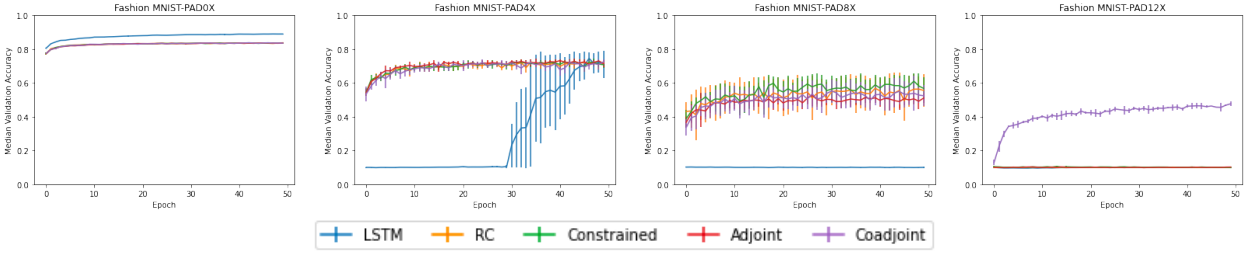


Figure 2.5: Fashion MNIST: Median validation accuracy for recurrent architectures with hidden dimension d = 100 for padded experiments 0X, 4X, 8X and 12X, respectively.

2.8 Observations on Gradient Control and Learning

Recall, we hypothesized that, when learning fails to occur in an RNN due to gradient variability, the variability of the adjoints ($\{\lambda_t\}$) is the culprit, and by inducing control via a penalty, $G(\{\lambda_t\})$, will address this issue and allow an RNN to learn. To evaluate the first part of this claim — the relationship between adjoint values and learning —, we compare the value $\frac{\|\lambda_1\|_2}{\|\lambda_T\|_2}$ (i.e., the ratio between the initial and terminal adjoints) and the classification accuracy. We recall that $\|\lambda_1\|_2 \to 0$ (gradient decay) and $\|\lambda_1\|_2 \to \infty$ (gradient explosion) is what we view as the primary culprit in the loss of learning. Thus, an adjoint ratio of one implies that the initial and terminal sequence positions are equally influential in gradient credit assignment. In Figure 2.6, we plot the values of the adjoint ratio (i.e. $\frac{\|\lambda_1\|_2}{\|\lambda_T\|_2}$) against the classification accuracy for networks trained by backpropagation and the coadjoint procedure.

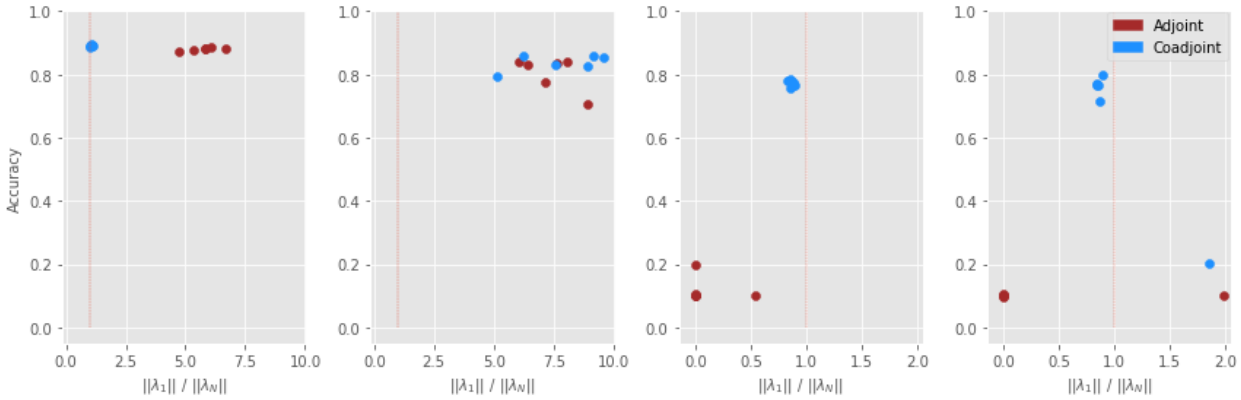


Figure 2.6: (left-to-right: No padding, 2x, 3x and 4x padding) Test accuracy and the ratio between initial and terminal adjoint sizes (i.e. $\frac{||\lambda_1||_2}{||\lambda_T||_2}$) for the Fashion MNIST noise padded tasks at training completion. We display six basic RNNs trained with either: backpropagation algorithm (brown), or the coadjoint algorithm (blue).

For the no padding and Pad 2x tasks, we see that there is really no relationship between the adjoint ratio and classification accuracy: this is expected as there are not prohibitively long temporal dependencies introduced such that gradient behavior becomes problematic. However, as we increase the padding, we see that the classification accuracy increases as the adjoint ratio approaches one (vertical line in Fig. 2.6). This pattern bolsters our claim that the values of the adjoints play a critical role in determining whether learning will occur, when gradient control is a problem. Furthermore, with the introduction of longer temporal dependencies through additional padding we see that RNNs trained with backpropagation (brown points in Fig. 2.6) tend to zero, exemplifying the regime where gradient decay is problematic. On the other hand, RNNs that were trained with the coadjoint method (blue

points in Fig. 2.6) are able to homogenize the adjoint sizes such that the model is able to utilize the gradient information provided from early sequence elements. Again, we view this pattern as bolstering our position. To summarize, we observe that the adjoint ratio is predictive of the classification accuracy as gradient flow becomes a problem, and, by controlling this value through our coadjoint training, we are able to induce learning.

While the general trend described in Fig. 2.6 suggests that RNNs will learn when the credit assignment is balanced (as measured by $\frac{||\lambda_1||_2}{||\lambda_T||_2}$), we also observe cases where credit assignment is balanced (i.e., $||\lambda_1||_2 \approx ||\lambda_T||_2$), yet task accuracy does not improve (see right panel of Fig. 2.6). This observation is in contrast to the viewpoint that RNN learning is highly correlated with the ability of the model to mitigate VEG during training.

Using this observation as motivation, in Chapter 3 we leverage the data generated from the factorial experiment described in Section 1.4 to study the relationship between RNN learning and VEG.

2.9 LU RNN

In Sections 2.6 and 2.7, we introduced the coadjoint procedure and algorithm for penalizing the intermediate partial derivatives (i.e., adjoint states) generated during backpropagation to improve gradient flow during training. Hence, the coadjoint procedure can be categorized as a technique for inducing gradient flow through regularization. Moreover, recall that regularization methods were one of a handful of main strategies for improving gradient flow and learning long-time dependent structure in RNNs that was discussed in Section 2.4. In this section, we re-examine another one of these strategies — constraining the recurrent matrix to be orthogonal/unitary — to motivate a simple architecture/procedure that empirically improves long-term memory in RNNs.

As described in Section 2.4, constraining the recurrent matrix to be orthogonal is one of the main strategies for controlling gradient flow in RNNs. The motivation for this parameterization can be characterized by the recurrent adjoint system (copied from (2.6) below).

$$\lambda_t = W'\sigma^{(1)}(Wx_t + Ru_{t+1} + b_1)'\lambda_{t+1} \qquad t < T \tag{2.27}$$

Namely, if $||W||_2 < 1$, then $||\lambda_t||_2 \to 0$ as $T \to 1$ and we have vanishing gradients; and if $||W||_2 > 1$, then $||\lambda_t||_2 \to \infty$ as $T \to 1$ and we have exploding gradients. However, if $||W||_2 = 1$, then W is a norm preserving linear map, and $||\lambda_t||_2$ of (2.27) will not be exponentially impacted by the magnitude of W. Thus, this constraint strategy addresses the gradient flow issue in RNNs by controlling the global gradient flow (from time index

$T \downarrow 1$) by controlling the local gradient flow (i.e., the change in adjoint magnitude from $\lambda_{t+1} \to \lambda_t$).

Using this same motivation, we now introduce a simple architecture that is capable of controlling the adjoint system without directly penalizing the recurrent matrix, or the gradients generated during backpropagation, which to our knowledge, is a novel technique for inducing gradient flow in RNNs. Specifically, rather than constraining the recurrent matrix, we simply require it to be invertible (i.e., $WW^{-1} = W^{-1}W = I$). Moreover, the forward dynamics remain identical to the basic RNN (see Def. 2.1), but W and W^{-1} are interchanged every $\tau < T$ time steps. The intuition of this construction is that alternating between W and W^{-1} creates a "pseudo" norm preserving property every 2τ time steps without requiring $\|W\|_2 = 1$.

As illustration, consider $\|W\|_2 = 2$ and the recurrent adjoint system, (2.6), where $\sigma^{(1)}(\cdot) = 1$ for simplicity. Clearly, when $\|W\|_2 = 2$, then $\|\lambda_t\|_2 > \|\lambda_{t+1}\|_2$, and the dynamics become susceptible to encountering the exploding gradient case as T increases. Similarly, if the model was parameterized with the matrix inverse, then $\|W^{-1}\|_2 = \frac{1}{2}$ and the dynamics of (2.27) become susceptible to the vanishing gradient case. However, if we interchange W for W^{-1} every τ time steps, then the adjoint system, (2.27), will alternate between expanding and contracting dynamics depending on whether W or W^{-1} parameterizes the recurrent matrix. As a result, W is free to vary in size, yet the adjoint system should remain relatively uniform in size over arbitrary length time-horizons (for a suitably chosen $\tau < T$). Furthermore, we see this parameterization as a relaxation of the orthogonally constrained RNNs. Namely, by inverting the recurrent matrix, gradient signal can be maintained over the entire computational path, but does so without restricting the recurrent matrix to be norm-preserving at each time point.

Although this simple trick for maintaining gradient strength is numerically sensible, it requires computing the inverse of W. In general, inverting the square matrix, $W \in \mathbb{R}^{d\times d}$, requires $\mathcal{O}(d^3)$ operations (e.g., using Gauss-Jordan elimination), which can become prohibitive if d grows. To reduce this cost, we use the LU decomposition to parameterize W (i..e, $W = LU$ where $L \in \mathbb{R}^{d\times d}$ is a lower triangular matrix, and $U \in \mathbb{R}^{d\times d}$ is an upper triangular matrix). As a result of this decomposition, W^{-1} can be computed in $\mathcal{O}(d^2)$ operations, while storing LU in place of W directly, only marginally increases storage requirements.

Using this parameterization, we now define the *LU RNN*.

Definition 2.4 (LU Recurrent Neural Network)**.** *For a single input sequence* $\{u_1, u_2, \dots, u_T\} \subset \mathbb{R}^p$, *constant* $\tau < T$, *and arbitrary initial state* $x_0 \in \mathbb{R}^d$, *an activation function* $\sigma : \mathbb{R} \to \mathbb{R}$ (*applied component-wise*), *and an output function* $\phi : \mathbb{R} \to \mathbb{R}$ (*applied component-wise*), *the LU Recurrent*

Neural Network is defined by

$$\begin{aligned} x_t &= \sigma(Wx_{t-1} + Ru_t + b_1) \qquad t = 1, \dots, T \\ \hat{y} &= \phi(Vx_T + b_2) \end{aligned} \tag{2.28}$$

where $W = LU$ *is stored as its LU decomposition and is interchanged with its inverse,* W^{-1}*, in the computation graph whenever* $t \mod \tau = 0$*.*

In addition to the LU RNN, we also consider directly parameterizing with W^{-1} (no LU decomposition) and refer to this architecture as the *inverse RNN*.

2.9.1 LU RNN (Preliminary) Experimental Results

To evaluate the LU RNN's and inverse RNN's ability to learn distant time dependencies, we utilize a variation of a common task designed for testing a RNN's ability to learn long-time dependent structure, *the adding problem*.

Experiment 2.4 (Adding Problem)**.** *First introduced in* [43]*, the adding problem was designed to stress the ability of an RNN to store bits of information over extended periods of time. We implement a variation of this task, where input features are formed by concatenating two univariate sequences of length* T*. The first sequence consists of elements* $u_t \in Unif(0, 1)$*); and the second is an indicator vector of length* T*, where two elements are marked. The first marked element is selected uniformly at random from positions* $[0, \frac{T}{2}]$*, and the second marked element from positions* $[\frac{T}{2} + 1, T]$*. The objective of the model is to return the sum of the two elements in the first sequence that have a corresponding marked position. In Fig. 2.7 we illustrate this task for a sample input pair.*

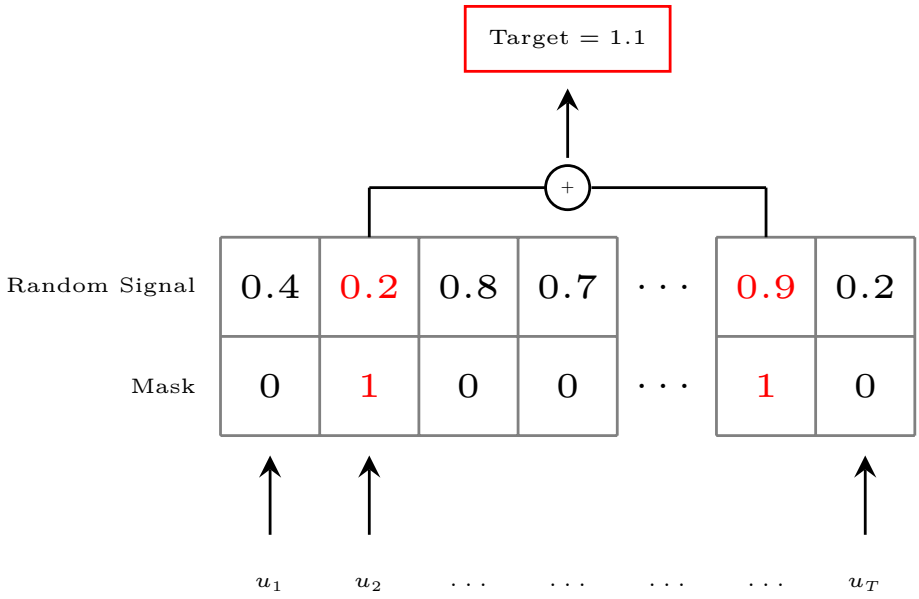


Figure 2.7: An example observation of the T horizon adding problem. In this input observation, the positions $t = \{2, T-1\}$ are marked, indicating that the sum of their corresponding random uniform elements will determine the target value (i.e., $0.2 + 0.9 = 1.1$).

Remark 2.3. *For the adding problem, a naïve strategy of predicting 1 for all output (regardless of the input sequence) results in an expected mean square error (MSE) of 0.167, the variance of the sum of two independent uniform distributions. This is the baseline which each model attempts to improve.*

We train each introduced architecture — LU RNN and inverse RNN — on the adding problem with $T = \{100, 250, 750\}$ and $\tau = 10$. We compare these models to the basic RNN and LSTM. For each model we fix the recurrent dimension to $d = 128$, the batch size to 32, and select the learning rate from $\in [10^{-4}, 10^{-3}]$. In Fig. 2.8 we display the MSE for each of these models and tasks. As depicted in Fig. 2.8, the basic RNN is unable to improve over the naive strategy for any T. The LSTM is able to quickly learn at $T = 100$, but fails to reduce MSE for $T = 250$. Both the LU RNN and inverse RNN are able to reduce MSE to zero for $T = \{100, 250, 750\}$.

Remark 2.4. *Other works have been able to train an LSTM to partially reduce MSE for the adding problem at* $T = 750$ *[3]. However, the discrepancy in results can (possibly) be explained by differences in batch size and optimizer.*

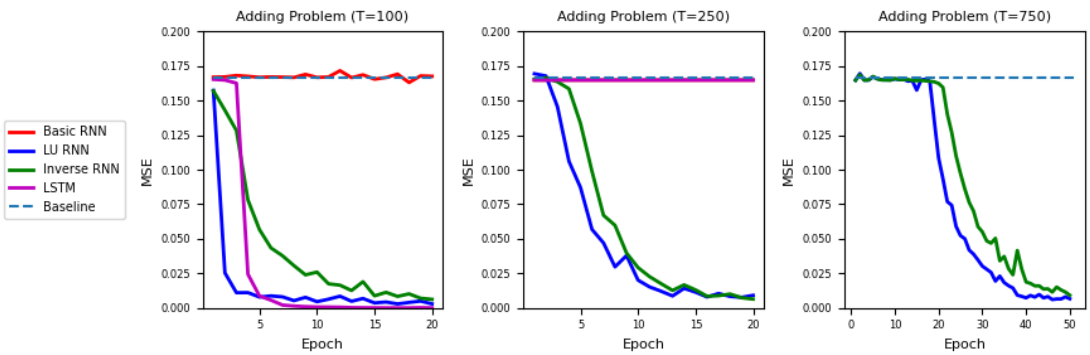


Figure 2.8: Results of the adding problem with $T = \{100, 250, 750\}$ for the basic RNN (red), LU RNN (blue), inverse RNN (red) and LSTM (magenta).

2.10 Conclusion

In this chapter, we studied RNNs from an optimization perspective, emphasizing the vanishing and exploding gradient problem, and the difficulty of learning long-term dependencies with these architectures. Motivated by this problem, we introduced two methods for mitigating VEG and improving long-term memory in RNNs: the coadjoint method and the LU RNN. The coadjoint method introduces a procedure for training RNNs that penalizes the intermediate derivatives generated during backpropagation (i.e., the adjoints) to control gradient flow (e.g., gradient decay), and allow for learning longer range dependencies. We demonstrated that our coadjoint method generalizes the orthogonal recurrence penalty commonly employed to address gradient control in recurrent networks; we discussed how it can be quickly computed through an additional backpropagation; and we pointed out how flexible our coadjoint method is beyond the example that we investigated in this work.

We compared our coadjoint method against state-of-the-art approaches that are designed for addressing gradient flow issues. We found that our coadjoint approach was able to induce learning for basic RNNs when gradient decay is prohibitive to learning and closes the gap between basic RNNs and current state-of-the-art approaches. In summary, our coadjoint approach is a unique training approach that seems to better address the gradient flow problem of training RNNs, and can help improve long-term memory in RNNs.

In addition to the coadjoint procedure, we introduced a simple architecture, the LU RNN, that better learns long-term dependencies than the basic RNN. Furthermore, we discussed the ease in which this architecture can be implemented, and how its computational path can be seen as a relaxation of the orthogonal recurrence constraint; and demonstrated its effectiveness on a commonly used benchmark for learning long-term dependencies.

3 REVISITING THE PROBLEM OF LEARNING LONG-TERM DEPENDENCIES IN RECURRENT NEURAL NETWORKS

Recurrent neural networks (RNNs) are an important class of models for learning sequential behavior. However, training RNNs to learn long-term dependencies is a tremendously difficult task, and this difficulty is widely attributed to the vanishing and exploding gradient (VEG) problem. Since it was first characterized 30 years ago, the belief that if VEG occurs during optimization then RNNs learn long-term dependencies poorly has become a central tenet in the RNN literature and has been steadily cited as motivation for a wide variety of research advancements. In this work, we revisit and interrogate this belief using a large factorial experiment where more than 40,000 RNNs were trained, and provide evidence contradicting this belief. Motivated by these findings, we re-examine the original discussion that analyzed latching behavior in RNNs by way of hyperbolic attractors, and ultimately demonstrate that these dynamics do not fully capture the learned characteristics of RNNs. Our findings suggest that these models are fully capable of learning dynamics that do not correspond to hyperbolic attractors, and that the choice of hyper-parameters, namely learning rate, has a substantial impact on the likelihood of whether an RNN will be able to learn long-term dependencies.

3.1 Introduction

Recurrent neural networks (RNNs) are an important class of models used for addressing tasks with sequential data, and are integral components to state-of-the-art encoder-decoder models for language transduction [4, 81, 54]. However, RNNs have a tremendous difficulty learning, especially when temporal dependencies are distant [10, 9]. An RNN's learning difficulty is attributed to the vanishing and exploding gradient (VEG) phenomenon. Indeed, in the 30 years since identifying VEG as the mechanism of poor RNN learning [10, 9], the belief, *if VEG occurs then an RNN learns long-term dependencies poorly*, has become a central tenet in the RNN literature. Moreover, this tenet is widely cited as motivation by a variety of research advancements to combat or avoid VEG, including specialized initialization schemes, novel empirical risk minimization formulations, novel recurrent neural network architectures, attention mechanisms, and the increasingly popular transformer architecture [102, 106, 23, 47, 111, 46, 65]. In fact, this tenet is so implicitly accepted that research advancements for recurrent networks establish their validity using the tenet's contrapositive: rather than directly showing that VEG is mitigated, these research advancements show an increase in test set accuracy, which, by the tenet's contrapositive, implies that the advancement mitigates VEG better than alternatives (e.g., see Table 1 of [32]). In other words, this tenet—*if VEG occurs then an RNN learns long-term dependencies poorly*—is an accepted fact, even at the forefront of RNN research.

Disclaimer: Portions of this chapter are pulled directly from a co-authored paper written with Vivak Patel, Prasanna Balaprakash and Yumian Cui that is currently under review at the journal *Neural Networks*.

In this chapter, we revisit whether this tenet and its implications are true. Broadly, we interrogate the following questions.

1. Is the tenet—if VEG occurs, then will an RNN learn long-term dependencies poorly—true?

2. Is its contrapositive—if an RNN learns long-term dependencies well, then VEG is mitigated—true?

To interrogate these questions, we use the data generated from the extensive factorial experiment detailed in Section 1.4. Recall that this experiment uses seven RNN architectures with two different recurrent dimensions each; five benchmark tasks with six variations each; two different training horizons; five learning rates; and with all possible combinations replicated ten times.[1] Thus, we explore these questions using over 40,000 training attempts requiring over 14 years of computational time on the Argonne National Laboratory *Theta* supercomputer [33].

Through this extensive experiment, we conclude that the answer to each of the above questions is negative. That is, we provide examples from our training attempts that illustrate the following.

1. Even if VEG occurs, we observe RNNs that still learn long-term dependencies substantially above baseline performance.[2]

2. When an RNN learns long-term dependencies substantially above baseline performance, we observe cases where VEG still occurs.

Importantly, we consider the possibility where the causal relationship is possibly fuzzy and perform an extensive statistical analysis to evaluate these claims under this context. We conclude as follows.

1. VEG has limited ability to explain when RNNs learn long-term dependencies above baseline performance during training or at the model's terminal iterate (marginal $R^2 \approx 0.005$ and $R^2 = 0.25$, respectively), and this explanatory power is further reduced when other covariates which describe the underlying task and training hyperparameters are included (less than 2% increase in R^2 when information measuring VEG is used as a predictor).

[1] For the UnICORNN, the full number of training attempts could not be reached owed to extremely long training times.

[2] We refer to baseline performance as the performance achieved from guessing for a given task.

2. The quality of RNN learning has limited explanatory power in the amount of VEG observed, and this explanatory power is diminished when other covariates are included in the model (quality of learning accounts for less than a 1.5% increase in explanatory power).

Our experimental conclusions seemingly contradict [9], which rigorously proves "that only two conditions can arise when using hyperbolic attractors to latch bits of information. Either the system is very sensitive to noise, or the derivatives of the cost at time t with respect to the system activations converge exponentially to 0 as t increases." In order for both our experimental conclusions and the mathematical results of [9] to be true, we must be in two distinct settings: namely, the standard RNN and more advanced architectures considered in this work are learning dynamics that do not necessarily correspond to hyperbolic attractors, which is the required setting for [9]. While other works have experimentally shown that it is possible for certain RNN architectures to have dynamics that are not hyperbolic attractors [11] or have theoretically constructed specific activations to generate RNNs with strange attractor dynamics [18, 19, 59], our experimental disagreement with [9] raises the question: in practice, do RNN dynamics correspond to hyperbolic attractors? That is, which setting occurs in practice? While our results from the aforementioned experiment already provide an answer — in practical settings, an RNN's dynamics do not always correspond to hyperbolic attractors —, we show that simple RNNs will learn dynamics that do not correspond to hyperbolic attractors using a synthetic latching experiment (described in Section 3.3). Accordingly, our experimental conclusions on the limited relationship between VEG and learning seem applicable in practical settings.

Chapter Contributions To summarize, we show a rather mild relationship between VEG and the ability of RNNs to learn long-term dependencies, which contradicts the pervasive belief that VEG prevents an RNN from learning long-term dependencies. Instead, we observe that it is the selection on hyper-parameters, namely learning rate, that has a substantial impact on whether an RNN will be able to learn long-term dependencies. We also observe that there is no single RNN architecture that seems to dominate tasks with long-term dependencies. In addition, we re-examine the framework of RNNs as hyperbolic attractors and with the use of a simple synthetic experiment provide evidence that illustrates long-term memory can be acquired by RNNs through dynamics that are not hyperbolic. This simple illustration provides a mechanism that can explain why such a mild relationship was observed between VEG and the ability of RNNs to learn long-term dependencies.

Chapter Organization In Section 3.2, we describe VEG and produce a metric that will be used to analyze VEG quantitatively. In Section 3.3, we describe our experimental design. In Section 3.4, we provide examples that address the veracity of the tenet and its contrapositive statement. In Section 3.5, we provide a statistical analysis of the tenet and its contrapositive. In Section 3.6, we re-examine the original framework for analyzing information latching over time. In Section 3.7, we conclude.

Note to Readers This chapter is written such that it can largely be read independently from the first two chapters of this dissertation. Thus, for the reader that chooses to read this dissertation straight through, there may be some redundancy with respect to the first two chapters of this dissertation. In light of this, readers are encouraged to skip Section 3.2 for a more succinct reading experience. Furthermore, although the description of the experimental design is elaborated in Chapter 1 when discussing task generalization, we reiterate the description in Section 3.3 of this chapter, and emphasize different aspects of the design that pertain to our investigation between VEG and RNN learning.

3.2 A Metric for the Vanishing and Exploding Gradient Phenomenon

Herein, we describe the characterization of VEG from [10] using the modern language of backpropagation, and use it to define a metric for VEG. We use the basic RNN (introduced in Def. 2.1 and reproduced below) as a starting point for this endeavor.

Definition 2.1 (Basic Recurrent Neural Network)**.** *For a single input sequence* $\{u_1, u_2, \dots, u_T\} \subset \mathbb{R}^p$ *and arbitrary initial state* $x_0 \in \mathbb{R}^d$*, an activation function* $\sigma : \mathbb{R} \to \mathbb{R}$ (*applied component-wise*)*, and an output function* $\phi : \mathbb{R} \to \mathbb{R}$ (*applied component-wise*)*, the Basic Recurrent Neural Network* (*basic RNN*) *is defined by*

$$\begin{aligned} x_t &= \sigma(Wx_{t-1} + Ru_t + b_1) \qquad t = 1, \dots, T \\ \hat{y} &= \phi(Vx_T + b_2) \end{aligned} \tag{2.1}$$

where $W \in \mathbb{R}^{d \times d}$ *is the recurrent weight matrix;* d *is the recurrent dimension;* $R \in \mathbb{R}^{d \times p}$*;* $b_1 \in \mathbb{R}^d$*;* $V \in \mathbb{R}^{l \times d}$*;* l *is the output dimension;* $b_2 \in \mathbb{R}^l$*; for any* $t \in \{0, \dots, T\}$*,* $x_t \in \mathbb{R}^d$ *is the memory state at time* t*; and* $\hat{y} \in \mathbb{R}^l$ *is the prediction.*

The basic RNN is visualized in Figure 3.1. The basic RNN can also be used to produce a prediction at each time point from $t = 1, \dots, T$, but we will focus on the case where a

prediction is made at the final time T (see Figure 3.1) as this is when VEG is most prominent. Moreover, while the diagram refers to the basic RNN, all RNN architectures produce a sequence of memory states, $\{x_t\}$, that are propagated forward in time and then used to produce a prediction.[3] Hence, all RNN architectures can suffer from VEG given that VEG depends on the behavior of the memory states, as we discuss presently.

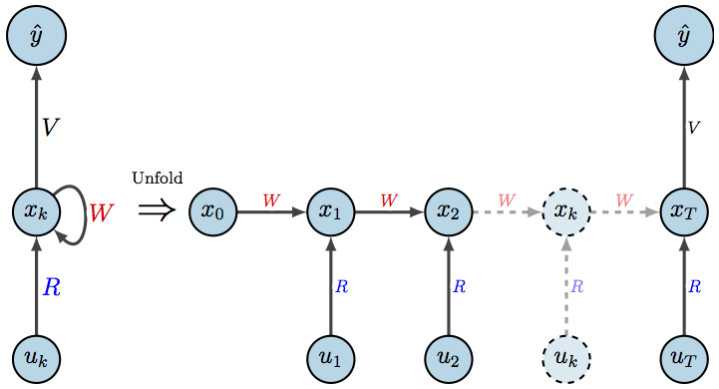


Figure 3.1: A visualization of the forward dynamics of the basic RNN (Definition 2.1).

From [10], an RNN suffers from vanishing gradients if for a given input sequence and parameters, $\|\partial x_\tau / \partial x_t\|_2$ for $t < \tau$ decays to zero as $\tau - t \to \infty$. Analogously, an RNN suffers from exploding gradients if for a given input sequence and parameters, $\|\partial x_\tau / \partial x_t\|_2$ for $t < \tau$ diverges to infinity as $\tau - t \to \infty$. In other words, an RNN suffers from VEG if a marginal change in the memory state at an earlier time point has an insignificant or a substantial impact on the marginal change in the memory state at a future time point.

While this characterization of an RNN's experience of VEG is useful for theory, it has two limitations for developing a metric for VEG in practice. First, this characterization does not apply to tasks with finite time horizons. Second, as VEG impacts the training dynamics, this characterization does not account for possible contributions from the loss function. Therefore, we consider a simple variation on this characterization that allows us to develop a metric for VEG that addresses the above two limitations. In order to describe our simple variation, we need to specify the training problem (i.e., empirical risk minimization). Using the basic RNN as an example, let L be a loss function between a label for an input sequence and the prediction from the RNN. Then, for a set of examples,

[3]While many RNN architectures can be visualized using Figure 3.1, there are some RNN architectures that use multiple memory states from previous iterates as inputs [93], but analogous diagrams can be produced.

$\{(u_1^j, u_2^j, \ldots, u_T^j, y^j) : j = 1, \ldots, N\}$, the training problem is to solve

$$\begin{aligned} \min_{W,R,b_1,V,b_2} \quad & \sum_{j=1}^{N} L(\hat{y}^j, y^j) \\ \text{s.t.} \quad & x_t^j = \sigma(Wx_{t-1}^j + Ru_t^j + b_1) \qquad t = 1, \ldots, T \\ & \hat{y}^j = \phi(Vx_T^j + b_2), \end{aligned} \tag{3.1}$$

where x_0^j is either randomly initialized for each j or set to a fixed value (e.g., zero).

The training problem, (3.1), is typically solved using (stochastic) gradient methods for which gradients of the parameters are computed using backpropagation [85, 61]. Using the rigorous Lagrangian formalism for backpropagation [62], backpropagation can be defined as the *backward* analogue of calculating the memory states; that is, backpropagation generates a sequence of vectors, called adjoint states, starting from time point T and terminating at time point 0, as we now define.

Definition 3.1 (The Adjoint States of Backpropagation)**.** *Given a basic RNN, an example* $(\{u_1, u_2, \ldots, u_T\}, y)$*, and a loss function* $L(\hat{y}, y)$*, the usual backpropagation calculation produces adjoint states* $\{\lambda_t : t = 1, 2, \ldots, T\}$*, defined by*

$$\lambda_t = \begin{cases} -V'\phi^{(1)}(Vx_T + b_2)' \frac{\partial L}{\partial \hat{y}} & t = T \\ W'\sigma^{(1)}(Wx_t + Ru_{t+1} + b_1)'\lambda_{t+1} & t < T, \end{cases} \tag{3.2}$$

where $\phi^{(1)}$ *and* $\sigma^{(1)}$ *(applied component-wise) represent the derivatives of* ϕ *and* σ*, respectively.*

In Figure 3.2, the calculation flow for the adjoint states is represented. Importantly, from Figure 3.2 and by the chain rule, the adjoint state is $\lambda_t = (\partial x_T/\partial x_t)(\partial L/\partial x_T)$. Thus, the adjoint states contain the sensitivities of the memory states to earlier time points, as is done in [10], and contains information about the loss function, which we mentioned was one of the limitations of developing a metric for VEG using the sensitivity of the memory states alone. What is more, as is well known [62], the adjoint states are then leveraged to compute the gradients with respect to the parameters (i.e., W, R, b_1, V, b_2). For example, the adjoint states can be used to compute the gradient of the recurrent weight matrix with respect to an example by

$$\nabla_W L(\hat{y}, y) = -\sum_{t=1}^{T} \sigma^{(1)}(Wx_{t-1} + Ru_t + b_1)'\lambda_t x_{t-1}'. \tag{3.3}$$

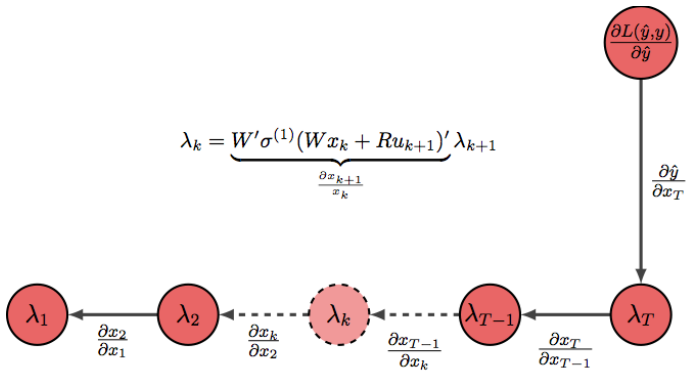


Figure 3.2: A visualization of the backward (adjoint) dynamics of the basic RNN (Definition 2.1).

From (3.3), the adjoint states play a critical role in VEG: roughly, if λ_t's decay rapidly as t approaches zero, then earlier values in the input sequence (i.e., those closer to $t = 0$) will have a small impact on any updates to the parameters; and if λ_t's grow rapidly as t approaches zero, then earlier values in the input sequence will have an out-sized impact on any updates to the parameters. Thus, the adjoint states can be used to quantify VEG by the logarithm of the adjoint ratio,

$$\log_{10} \frac{\|\lambda_1\|_2}{\|\lambda_T\|_2}. \tag{3.4}$$

Specifically, when (3.4) is negative or positive, then we are observing some degree of either vanishing or exploding gradients. Of course, the magnitude of (3.4) indicates the severity of the problem: large absolute values of (3.4) indicate more severe VEG. Hence, (3.4) will be our metric for VEG.

3.3 Experimental Design

3.3.1 Experiments on VEG and Learning

We now describe an experiment to interrogate the relationship between VEG and learning. The experimental design is fully described by the factors and levels presented in Table 3.2 with further details provided below. The combination of each experimental level produces a single treatment, resulting in 4,200 treatments, which are then replicated ten times each independently. Thus, the experiment has 42,000 training attempts (i.e., experimental units), which were excuted over 14 year of computational time at Argonne National Laboratory's *Theta* supercomputer [33].

The experiment factors and levels were designed for the following reasons.

Factor	Levels
Architecture	{Basic RNN [86], LSTM [42], GRU [21], Antisymmetric RNN [16], Lipschitz RNN [32], Exponential RNN [65], UnICORNN [87]}
Recurrent dimension	{128, 256}
Learning rate	$\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$
Training length	{25 epochs, 50 epochs}
Dataset	{CIFAR10 (T = 1024), Fashion MNIST (T = 784), IMDB (T = 500), MNIST (T = 784), Reuters (T = 500)}
Order	{sequential, permuted}
Noise orientation	{none, post, uniform}

Table 3.2: Experimental factors and levels.

1. For the architecture factor, the basic RNN was included as it is the canonical model on which VEG was first described. The other architectures were selected as they have had very strong performance on benchmark tasks (see Table 1 of [32]). Owing to their performance on benchmark tasks and owing to the contrapositive of the tenet, these architectures should not experience VEG as they are able to learn long-term dependencies. Thus, if these architectures do experience VEG, then this can either be attributed to the contrapositive of the tenet being tenuous or to the limited ability of benchmarks to be representative of similar tasks. To discern between these two, the performance of the architectures are always recorded and the tenet is analyzed relative to this performance. That is, the architecture's performance on benchmarks reported in other work is not assumed to apply on similar tasks used here.

2. The recurrent dimension factor is driven by practical considerations. Initial explorations of recurrent dimensions smaller than 128 revealed very little learning could be achieved across the architectures, which would not yield any interesting information. Moreover, recurrent dimensions larger than 256 often produced training attempts that were too expensive to run for certain architectures within our computational budget as the number of parameters can vary substantially across architectures even for the same recurrent dimension (see Table 3.3).

3. The learning rates were selected to contain values that were observed to be successful in the literature the architectures above, and that produced some degree of learning

	Recurrent Dimension			
Architecture	$l = 64$	$l = 128$	$l = 256$	$l = 512$
Basic RNN	4,224	16,640	66,048	263,168
LSTM	16,896	66,560	264,192	1,052,672
GRU	12,672	49,920	198,144	789,504
Antisymmetric RNN	4,224	16,640	66,048	263,168
Lipschitz RNN	8,320	33,024	131,584	525,312
Exponential RNN	4,224	16,640	66,048	263,168
UniCORNN	4,544	17,280	67,328	265,728

Table 3.3: The number of parameters for different architectures by the recurrent dimension where the feature dimension is fixed to $p = 1$.

from preliminary training attempts.

4. The training lengths of 25 and 50 were selected to fit within the computational budget, still produce learning across the majority of training attempts, and to assess the impact of VEG during training.

5. The data sets were selected from standard benchmark tasks for comparing RNN performance in the literature. Each task is a classification task where if no errors occur during prediction, then the model has achieved the maximum possible accuracy of 100%. For each data set, the input sequence is either kept as it is or randomly permuted (the permuted is kept fixed across all examples for a training attempt). Similarly, for each data set, the input sequence (whether the original or permuted) is kept as it is or is padded with 1000 inputs of noise in one of two ways: noise is placed at the end of the sequence (post), or uniformly spread between the inputs (uniform). An illustrative example of the variations of the input sequences are described in Table 1.3 and Fig. 1.1 of Chapter 1. These input sequence variations were selected in order to artificially modify the long-term dependencies that exist in the input sequence of each example, either by breaking them (i.e., permuting input sequences) or by elongating them (i.e., by adding noise). While these additional task variations break the typical structure of the real-world problem, they are commonly employed in evaluating new RNNs as they lengthen or *partially* perturb the temporal horizon of the task making it more difficult to learn [60, 3, 16, 103]. To reiterate, these task variations do not *completely* remove temporal structure from the problem, but instead exacerbate the difficulty in learning the original (sequential) task.

6. In tasks where additional noise was appended to the input sequence, the noise elements were generated uniformly from the range of the original input values. Each

noise sequence was generated once and appended to the original input sequence prior to training. Owing to the randomization in noise generation, it is likely that the added noise tasks will differ in training input. While this allows for the possibility of training set difficulty to vary within task, we replicate each task over ten training attempts, mitigating any differences.

7. The Adam optimizer [55] and a batch size of 32 was used for all training attempts.

In all, the experiment is designed to the best of our ability to interrogate VEG based on the current understanding of RNN learning and performance. Importantly, we have no direct control over whether VEG occurs and to what degree, nor do we have control over the resulting performance of the model trained during a training attempt.

For each of the 42,000 models trained, we record summary statistics after each training epoch. Specifically, at epoch termination, we compute the statistics: loss, accuracy, $||\lambda_1||_2$ and $||\lambda_T||_2$ over the training data and a disjoint test set and record the average over that respective set. Here forward, we will write the initial and terminal adjoint as these quantities, i.e.,

$$||\lambda_1||_2 = \sum_{i=1}^{N} ||\lambda_1^{(i)}||_2 \quad \text{and} \quad ||\lambda_T||_2 = \sum_{i=1}^{N} ||\lambda_T^{(i)}||_2, \tag{3.5}$$

where $i = 1, 2, \ldots, N$ indexes either the training or testing set which for our experimental task ranges from $N \in [2800, 60000]$. Hence, as a proxy for (3.4), we construct the VEG metric from the average statistics recorded,

$$\log_{10} \frac{||\lambda_1||_2}{||\lambda_T||_2} = \log_{10} \frac{\sum_{i=1}^{N} ||\lambda_1^{(i)}||_2}{\sum_{i=1}^{N} ||\lambda_T^{(i)}||_2}. \tag{3.6}$$

where, again, N denotes the number of samples used for computing the average statistics which is dataset dependent and ranges from $N \in [2800, 60000]$ over the experimental tasks.

At first glance, using $||\lambda_1||_2$ and $||\lambda_T||_2$ seems to have the disadvantage that VEG information for individual examples is lost. However, this is not necessarily the case owing to two considerations. First, the values of $||\lambda_T^{(i)}||$ tend to be of orders of magnitude compared to the magnitude of $||\lambda_1||_2$ when either the vanishing or exploding phenomenon is observed (see Figure 3.3). Then, the ratio in (3.6) will be rather similar to the average of the adjoints with the advantage that potential cancellation errors from finite arithmetic are more likely to be avoided. Hence, (3.6) is a reasonable way to capture potential VEG behavior. Second, in terms of resource constraints, using the individual values, $||\lambda_1^{(i)}||$ and $||\lambda_T^{(i)}||$ for $i = 1, \ldots, N$,

becomes prohibitively expensive, as writing $\|\lambda_1^{(i)}\|/\|\lambda_T^{(i)}\|$ over all examples and training attempts requires between 35 GB to 756 GB of data which would have consumed much of our computational budget and would have certainly exceeded our storage capacity. Similarly, storing model weights after each epoch and then computing statistics post training experimental design would have required writing between 126 GB and 1.58 TB of data.[4]

3.3.2 Experiments on Learning Dynamics

In addition to these tasks that make use of real-world data, we utilize a synthetic latching experiment to investigate the dynamics learned in which an RNN can successfully store information over an extended period of time. Recall, we have that if an RNN's learned dynamics have a hyperbolic attractor, then "Either the system is very sensitive to noise, or the derivatives of the cost at time t with respect to the system activations converge exponentially to 0 as t increases" [9]. Thus, if a trained RNN is not sensitive to input noise nor if a trained RNN's sensitivities converges exponentially to 0 as t increases, then the RNN's learned dynamics *are not* generating a hyperbolic attractor.

To assess the sensitivity to noise or decay of derivatives, we consider a family of Long Time Gap (LTG) tasks (introduced in Section 2.5), which is a variation of the latching experiment designed in [9]. Recall from Section 2.5 in Chapter 2 that the LTG task is a simple synthetic binary classification task that requires the model to latch a single signal amongst a sequence of noise. The input sequence, $\mathcal{U} = \{u_j\}_{j=1}^T$, is a Rademacher series of length T — i.e., each u_j is an i.i.d. Rademacher random variable taking value of either $+1$ or -1 with probability 0.5 — and for our current exploration we will fix the label such that it corresponds to $\text{sign}(u_{23})$ for a pre-specified sequence location $t = 23$.

The idea of this experiment is simple in nature: the model must learn to latch to a specific input position (i.e., token position $t = 23$), and successfully maintain this information in memory when exposed to subsequent noise tokens (i.e., input tokens $t = 23, 24, \ldots, T$). The ability to identify the label relevant position is referred to as *latching*, and the ability to resist deleting/writing over this stored information when exposed to subsequent noise as *robust latching*. Hence, an RNN will only be able to successfully complete this task if it can learn to robustly latch to the label relevant token.

Now, we can train a basic RNN of fixed recurrent dimension, $d = 4$, on a variety of time horizons $T \in \{40, 80, 160, 400, 800\}$. If we are able to successfully have the basic RNN learn, then we have ruled out sensitivity to noise. However, at each training, we would

[4]Our smallest model can be stored with 80 KB and the largest with 1 MB. Computing the memory requirement for the smallest model evaluated we have 80 KB$\times$25 epochs$\times$21000 models$+$80 KB$\times$50 epochs$\times$ 21000 models $=$ 126 GB.

change the recurrent weight which would prevent us from analyzing what would happen to the derivatives as T increases. Instead, we will train a basic RNN at the smallest time horizon, we fix the recurrent weight, and then, for all larger time horizons, only train the output layer (i.e., reservoir computing [98]). Thus, if we are able to successfully learn and if our adjoint states do not decay for increasing T, then our basic RNN's dynamics do not correspond to a hyperbolic attractor.

To summarize the experiment, we fix the recurrent dimension of a basic RNN to $d = 4$, and vary the temporal horizon of the task $T = \{40, 80, 160, 400, 800\}$. We denote each of these tasks by D_T. We train the basic RNN on D_{40} to an error of less than 1% and denote this model by M_{40}. We then train four additional RNNs on the remaining four time horizons with the same recurrent weigth as M_{40} (i.e., we only train the output layer). Note, we are able to train these four additional RNNs to an error of less than 1% as well. We denote these models as M_{80}, M_{160}, M_{400}, and M_{800}, where the subscript corresponds to the time horizon.

For each of the trained models, we record the norm of the forward (recurrent state) and backward (adjoint state) dynamics.

3.4 Counter Examples

In the previous section we used a synthetic experiment to demonstrate that RNNs are capable of encoding long-term memory using dynamics that are not hyperbolic. We now further this discussion by examining each question posed in Section 3.1 using our extensive factorial experiment whose design is explained in the previous section to extend our discussion to real-world data tasks. Below, we restate each claim and show examples from our experiment that are counterexamples to each claim.

If VEG occurs, then will an RNN learn long-term dependencies poorly To disprove this statement, we need to find training attempts for which VEG occurs and the RNN is able to learn long-term dependencies above baseline. Here, we need to be more specific about what we mean by VEG and its occurrence, and we need to be specific about what learning above baseline means. While we have a metric for VEG, (3.6), it will be qualitatively useful to specify three distinct regimes using the metric,

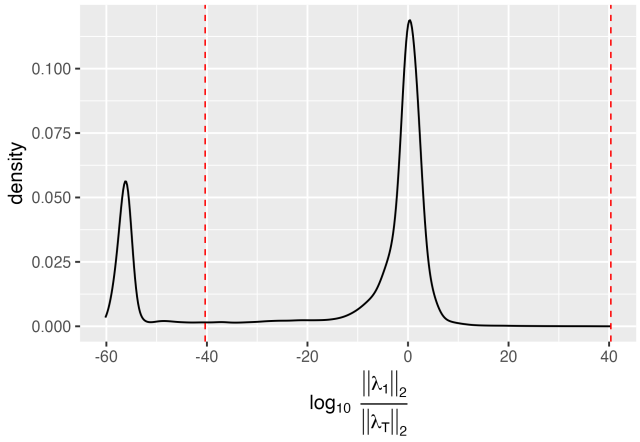


Figure 3.3: Histogram of adjoint ratios for all training attempts and epochs. The red dashed lines are drawn at $x = 40.35$ and $x = -40.35$, which correspond to the maximum recorded log adjoint ratio (i.e., 40.35) and its reflection over the origin and the endpoints of the adjoint regimes.

$$\text{Regime} = \begin{cases} \text{stable} & -40.35 < \log_{10} \frac{||\lambda_1||_2}{||\lambda_T||_2} < 40.35 \\ \text{vanishing} & \log_{10} \frac{||\lambda_1||_2}{||\lambda_T||_2} \leqslant -40.35 \\ \text{exploding} & \log_{10} \frac{||\lambda_1||_2}{||\lambda_T||_2} \geqslant 40.35 \end{cases} \tag{3.7}$$

The regime endpoints were determined from the distribution of collected adjoint ratios (see Figure 3.3) where 40.35 was the largest observed log adjoint ratio across all training attempts that did not incur numerical error. Training attempts that experienced exploding gradients were unable to be recorded owing to numerical failure during training. Although we can not rigorously address the exploding case because of this, we include a short summary of the training attempts that incurred numerical error in the appendix. Lastly, the lower bound of the stable regime was determined by reflecting the maximum observed log adjoint ratio over the origin. We cede that the vanishing regime could very well be established with a much larger upper bound threshold (e.g., $\log_{10} \frac{||\lambda_1||_2}{||\lambda_T||_2} \leqslant -10.0$ would imply that $||\lambda_1||_T$ is ten orders smaller than $||\lambda_T||_2$), however by using the chosen threshold we employ a very conservative characterization of the vanishing regime. Thus, if we were to consider a larger threshold value for distinguishing vanishing adjoints, then our subsequent results would only be strengthened.

We specify learning quality using a range of values as follows. For each task (i.e., dataset $\times$ order $\times$ noise orientation), of which there are thirty, we compute the Q_p quantile as the evaluation accuracy such that $100 \times p\%$ of training attempts resulted in evaluation accuracy less than Q_p. As such, learning quality is determined relative to all RNN training attempts

on a given task.

We will examine the tenet and VEG with respect to two narratives: during training, and during evaluation (after epoch 25 or 50). When a RNN experiences VEG during training (see the discussion of (3.3)), the RNN is expected to be an ineffective learner owing to the data driven relationships being muddled during optimization. Hence, to understand VEG's relevance during training, we examine training attempts that fall into the vanishing regime according to (3.7) during training, and examine how well these RNNs are able to learn. In Table A4 of the appendix, we display the number of training attempts that are in the vanishing regime *at some point during training* for each task binned by the learning quality quantiles (specifically, $Q_{0.25}$, $Q_{0.50}$, $Q_{0.75}$, $Q_{1.0}$) discussed above. While the majority of training attempts that experience vanishing gradients do not learn, the table shows many training attempts that learn in the upper quartile and that are in the vanishing regime, contradicting the tenet's claim.

We now examine the tenet as it relates to evaluation of the network. Recall, VEG can be interpreted as the perturbation sensitivity of the memory state at a time point near the output layer to an earlier layer. Hence, if $||\lambda_1||_2 \approx 0$ for $t \ll T$, then changes in the memory state near $t = 1$ will have little impact on the memory state at time T. At evaluation, this means that relevant information at the beginning of the temporal sequence will have a limited impact on the quality of the prediction (as measured by the loss function to a true label). In Table A5, we display the number of training attempts that are in the vanishing regime *at the terminal iterate* for each task binned by the learning quality quantiles and in Table 3.4 a summary of these counts by task type. From Table 3.4, three of the six task varieties have training attempts with vanished adjoints in the top most quartile of evaluation accuracy, and if the threshold is loosened to consider training attempts attaining higher than median evaluation accuracy, all six task varieties include training attempts that incur VEG.

	sequential	permuted	sequential × post	permuted × post	sequential × uniform	permuted × uniform
$[Q_{0.75}, Q_{1.00}]$	113	0	0	8	49	0
$[Q_{0.50}, Q_{0.75}]$	7	30	18	75	234	130
$[Q_{0.25}, Q_{0.50}]$	52	35	298	218	549	701
$[Q_{0.00}, Q_{0.25}]$	711	584	688	592	739	641

Table 3.4: Summary of number of training attempts with vanished gradients according to (3.7) that fall within each learning quartile for each task type across all experimental datasets.

If an RNN learns long-term dependencies well, then VEG is mitigated The evidence from our interrogation of the preceding statement is sufficient to answer this question

negatively. However, we include Fig. 3.4 for additional evidence, and add a dimension to this discussion. In particular, we know that certain architectures have achieved the best performance on certain benchmark tasks. According to the tenet, we ought to believe that the best performing architectures have then mitigated VEG and should learn long-term dependencies well. Unfortunately, in our experiments, we observe cases when the basic RNN mitigates VEG as well, or better than the newer designed architectures claiming this characteristic. To demonstrate we use the terminal iterate and the IMDB permuted uniform noise orientation task. For each architecture, we select the replicate that best mitigates VEG, or in terms of our VEG metric, the replicate that produced a log adjoint ratio nearest zero. Note that a log adjoint ratio of zero is indicative of $||\lambda_1||_2 = ||\lambda_T||_2$ and the successful mitigation of VEG. In the left panel of Figure 3.4 we plot each of these replicate's evaluation accuracy on the vertical axis, and their corresponding log adjoint ratio on the horizontal. We draw a dashed black line at $x = 0$ to emphasize perfect avoidance of VEG (i.e., $||\lambda_1||_2 = ||\lambda_T||_2$). In this case, we observed that the basic RNN produced the replicate that most successfully mitigated VEG (i.e., closest to the dashed line in Figure 3.4).

As a secondary example, we examine the training attempts for the Fashion MNIST permuted post noise orientation task and select the training attempt that resulted in the highest evaluation accuracy for each architecture. In the right panel of Figure 3.4 we plot these training attempt's evaluation accuracy on the vertical axis and their log adjoint ratio on the horizontal axis. By the statement of the tenet's contrapositive, the architecture that resulted in the highest evaluation accuracy should best mitigate VEG. However, this is not the case. While the GRU best mitigates VEG in this example (i.e., closest to dashed black line), there are three architectures that attain higher evaluation accuracy yet fail to mitigate VEG as well as the GRU.

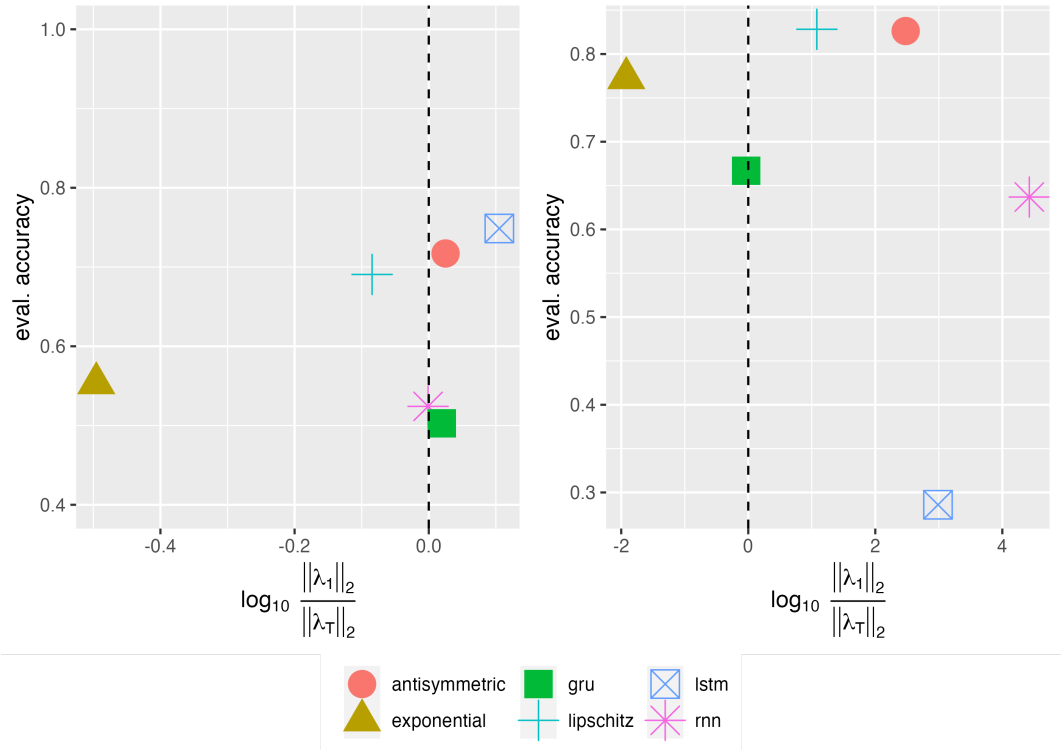


Figure 3.4: (Left) Evaluation accuracy and log adjoint ratio for a single replicate from each RNN architecture sampled based on its log adjoint ratio distance from 0 on the permuted uniform IMDB task. (Right) Evaluation accuracy and corresponding log adjoint ratio for the replicate that maximizes evaluation accuracy for each RNN architecture on the permuted post Fashion MNIST task. In both figures the black dashed line represents a log adjoint ratio of 0 (i.e. $||\lambda_1||_2 = ||\lambda_T||_2$).

While the evidence presented in Figure 3.4 could be rationalized along the argument that VEG and learning is not a hard rule, and thus, the relative nearness of the adjoint metric to zero is not important as long as the model has attained an adjoint metric sufficiently near zero. In this case, the models and adjoint metrics presented in Figure 3.4 are not a strong statement against the tenet's contrapositive. Owing to this, we include Figure 3.5, where we select training attempt treatment replicates from our experimental design (i.e., same architecture, training hyperparameters and task), and thus only differ by parameter initialization. For both treatments displayed in Figure 3.5 we have training attempts that are classified as both stable or vanished according to (3.7). If the tenet's contrapositive was true, then we would expect the replicates that attained the highest test accuracy to have also best mitigated VEG. However, we observe the opposite. Namely, there is an inverse relationship between test accuracy and the size of the adjoint metric within each of these treatments.

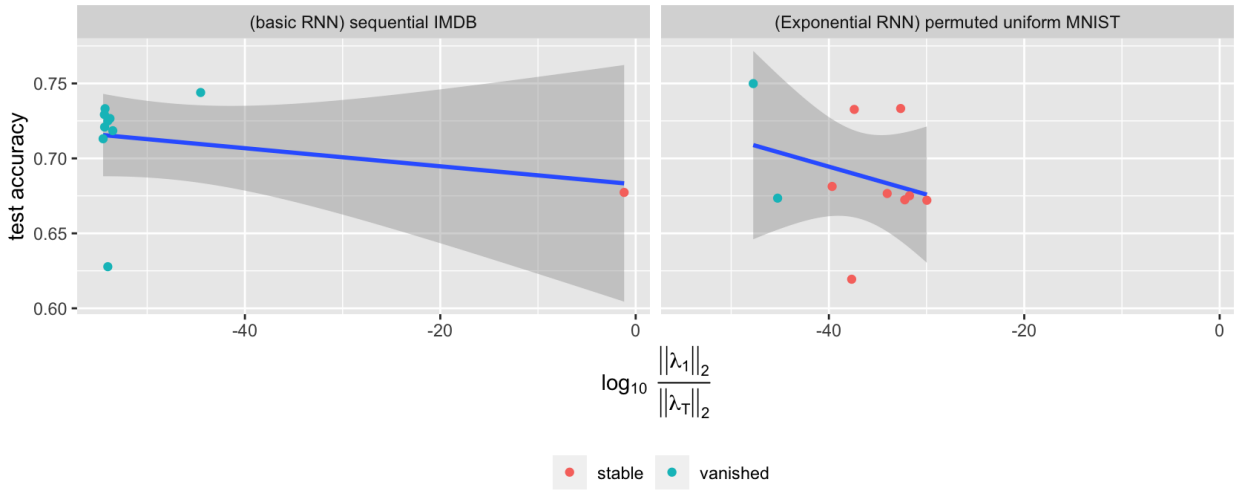


Figure 3.5: Test accuracy and adjoint metric at terminal iterate for replicates of two treatments. Ten replicates produced from the basic RNN on the sequential IMDB task (left) and ten replicates produced from the Exponential RNN on the permuted uniform MNIST task (right). For each figure, replicates are generated using the same recurrent dimension, learning rate and training length.

3.5 Statistical Analysis

In the previous section we provided counter examples that contradict the belief that if VEG occurs, then RNNs learn long-term dependencies poorly. While the counter examples alone suffice to demonstrate that this belief is often misconstrued, we further the discussion by framing the tenet and its contrapositive as a modeling problem. In the case of the tenet, if VEG occurs, then an RNN will learn long-term dependencies poorly, task accuracy is implied to be dependent on VEG. While in the case of its contrapositive, if an RNN learns long-term dependencies well, then VEG is mitigated, the direction of this relationship is reversed and VEG insinuated to be a by-product of task accuracy. Using linear models we interrogate these suggested relationships to better understand the validity of the tenet and its contrapositive.

For each of these statements, we develop six linear models: three with respect to data collected during training, and three with respect to data collected at the terminal iterate. For each statement and setting (i.e., train and terminal iterate) predictors are varied in a structured manner to probe at the importance of VEG in determining task accuracy (in the case of the tenet), and at the importance of task accuracy in determining VEG (in the case of the contrapositive). That is, in terms of the tenet, we exercise a three step modeling procedure:

1. accuracy ~ adjoint metric

2. accuracy ~ adjoint metric + covariates describing the experimental factors of Table 3.2

3. accuracy ~ covariates describing the experimental factors of Table 3.2

Under this procedure the first model is concerned with how well VEG alone can predict task accuracy; the second model interested in how this relationship changes when additional covariates describing the experimental treatment are included; and the third examining how the removal of information quantifying VEG impacts the predictiveness of the experimental factors. Furthermore, we perform the analogous three model procedure where we interchange task accuracy and the adjoint metric to investigate the tenet's contrapositive statement.

3.5.1 Modeling task accuracy as a function of VEG

We start with modeling the relationship suggested by the tenet's statement under the training narrative. That is, we model task training accuracy as a function of the adjoint metric.

During Training Recall that for each training attempt we collect summary statistics describing both performance (task accuracy) and VEG at each epoch. As such, the 42,000 training attempts produce a total of greater than one million observations ($N_{\text{train}} = 1,072,475$). Thus each training attempt contributes multiple observations implying dependent structure within these observations. To account for this dependency, we fit linear mixed effects models (LMMs) where training attempt is encoded as a random intercept. Furthermore, in accordance with the second and third steps of our modeling procedure we incorporate covariates of architecture, log learning rate (encoded as $\mathtt{LR}$), second order learning rate (encoded as $\mathtt{LR}^2$), recurrent dimension (encoded as $\mathtt{dim}$), dataset, order, noise orientation, and interactions of architecture $*$ $\mathtt{LR}$, architecture $*$ $\mathtt{LR}^2$, and architecture $*$ noise orientation.

Owing to the large number of coefficients in the subsequent models to be presented, we reserve full model coefficients to the appendix and include those encoding the fixed intercepts and adjoint metric effect. Conforming to our established modeling procedure,

we fit the first three models as,

$$\text{accuracy}_{ij} = 0.555 + 8.1 \times 10^{-4} \times \left[\log_{10} \frac{||\lambda_1||_2}{||\lambda_T||_2}\right]_{ij} + \text{training attempt}_i + \epsilon_{ij} \quad \text{(MODEL 1)}$$

$$\begin{aligned}\text{accuracy}_{ij} = &- 0.389 + 6.6 \times 10^{-4} \times \left[\log_{10} \frac{||\lambda_1||_2}{||\lambda_T||_2}\right]_{ij} + \beta_2 \times \text{LR} + \beta_3 \times \text{LR}^2 + \beta_4 \times \text{dim} \\ &+ \beta_a \times \text{architecture} + \beta_b \times \text{dataset} + \beta_c \times \text{order} + \beta_d \times \text{noise orientation} \\ &+ \tau_a \times (\text{LR} * \text{architecture}) + \tau_a \times (\text{LR}^2 * \text{architecture}) \\ &+ \tau_{ad} \times (\text{architecture} * \text{orientation}) + \text{training attempt}_i + \epsilon_{ij}\end{aligned} \quad \text{(MODEL 2)}$$

$$\begin{aligned}\text{accuracy}_{ij} = &- 0.389 + \beta_2 \times \text{LR} + \beta_3 \times \text{LR}^2 + \beta_4 \times \text{dim} \\ &+ \beta_a \times \text{architecture} + \beta_b \times \text{dataset} + \beta_c \times \text{order} + \beta_d \times \text{noise orientation} \\ &+ \tau_a \times (\text{LR} * \text{architecture}) + \tau_a \times (\text{LR}^2 * \text{architecture}) \\ &+ \tau_{ad} \times (\text{architecture} * \text{orientation}) + \text{training attempt}_i + \epsilon_{ij}\end{aligned} \quad \text{(MODEL 3)}$$

where $i = 1, 2, \ldots, 42,000$ indexes training attempt; $j = 1, 2, \ldots, n_i$ the n_i observations from training attempt i; a the observation's architecture; b the task data set; c the task order; d the task noise orientation; and ϵ_{ij} the residual error of the j^{th} observation of training attempt i. For models that include the additional covariates an implicit training attempt index is assumed and a slight abuse of notation used with respect to the discrete independent variables.[5]

In Table 3.5 we display summary statistics for MODEL 1, MODEL 2 and MODEL 3 where the conditional R^2 statistic measures the variance explained by both fixed and random effects, and the marginal R^2 statistic measures the variance explained by the fixed effects alone [75].

	# params	condition number	RMSE	conditional R2	marginal R2
MODEL 1	2	32.0	0.093	0.910	0.004
MODEL 2	43	8302.4	0.080	0.935	0.705
MODEL 3	42	8260.3	0.080	0.935	0.701

Table 3.5: Summary statistics for MODEL 1, MODEL 2 and MODEL 3.

Summarizing the conclusions from Table 3.5: the adjoint metric alone is a very weak predictor of task accuracy (MODEL 1 marginal $R^2 = 0.004$); including additional fixed effect

[5]We write $\beta_a \times$ architecture as a shorthand for $\beta_a \times \mathbf{1}\{\text{architecture} = a\}$ implying training attempt i belonged to architecture level a.

covariates along with the adjoint metric results in the highest explanatory power (MODEL 2 marginal $R^2 = 0.705$); and the omission of the adjoint metric as a predictor has a near negligible impact on the model explanatory power (MODEL 3 marginal $R^2 = 0.701$).

While the negligible marginal R^2 of MODEL 1 implies VEG is a rather poor predictor of task (training) accuracy, the estimated coefficients offer additional support. In both MODEL 1 and MODEL 2, the estimated coefficient for the adjoint metric is on the order of 10^{-4}, while the range of all collected adjoint metric values is on the order of 10^2. Combining these two pieces of information implies training accuracy will vary by at most 1% when all other covariates are fixed.

At Terminal Iterate To model VEG at the terminal iterate and its impact on the ability of an RNN to generalize we consider a response of evaluation accuracy and the adjoint metric computed over a disjoint test set at the terminal iterate (i.e., after either epoch 25 of epoch 50) as a predictor.[6] In contrast to the during training discussion, each training attempt contributes a single observation to the modeling data set. Subsequently, we drop the random intercept term used previously and fit fixed effects models,

$$\text{eval. accuracy}_i = 0.561 + 5.5 \times 10^{-3} \times \Big[\log_{10} \frac{||\lambda_1||_2}{||\lambda_T||_2}\Big]_i + \epsilon_i \qquad \text{(MODEL 4)}$$

$$\begin{aligned}\text{eval. accuracy}_i = &-0.167 + 3.0 \times 10^{-3} \times \Big[\log_{10} \frac{||\lambda_1||_2}{||\lambda_T||_2}\Big]_i + \beta_2 \times \text{LR} + \beta_3 \times (\text{LR})^2 + \beta_4 \times \text{dim} \\ &+ \beta_a \times \text{architecture} + \beta_b \times \text{dataset} + \beta_c \times \text{order} + \beta_d \times \text{orientation} \\ &+ \tau_a \times (\text{LR} * \text{architecture}) + \tau_a \times (\text{LR}^2 * \text{architecture}) \\ &+ \tau_{ad} \times (\text{architecture} * \text{orientation}) + \epsilon_i\end{aligned} \qquad \text{(MODEL 5)}$$

$$\begin{aligned}\text{eval. accuracy}_i = &-0.170 + \beta_2 \times \text{LR} + \beta_3 \times (\text{LR})^2 + \beta_4 \times \text{dim} \\ &+ \beta_a \times \text{architecture} + \beta_b \times \text{dataset} + \beta_c \times \text{order} + \beta_d \times \text{orientation} \\ &+ \tau_a \times (\text{LR} * \text{architecture}) + \tau_a \times (\text{LR}^2 * \text{architecture}) \\ &+ \tau_{ad} \times (\text{architecture} * \text{orientation}) + \epsilon_i\end{aligned} \qquad \text{(MODEL 6)}$$

In comparison to the predictive power of the adjoint metric on training accuracy, the adjoint metric appears to be a stronger predictor in the evaluation setting where MODEL 4 captures 25.4% of the variation in task evaluation accuracy (see Table 3.6) suggesting a mild relationship between VEG and task evaluation accuracy. Moreover, this is reflected in

[6] There were numerous training attempts that incurred numerical error during training, thus reducing the number of training attempts that we consider at evaluation time from 42,000 to 29,566.

MODEL 4 and MODEL 5 coefficient estimates where both models estimate this effect on the order of 10^{-3} (in comparison to 10^{-4} in the training narrative). Specifically, in terms of MODEL 5 and all other covariates fixed, an increase in the adjoint metric of 3.33 is associated with an increase in task evaluation accuracy of 1%.

While MODEL 4 suggests the relationship between VEG and task evaluation accuracy is not negligible, when additional covariates are included the explanatory power of VEG becomes insignificant. Namely, the difference in R^2 between MODEL 5 and MODEL 6 is a meager 0.018, suggesting VEG has very limited explanatory power in comparison to the experimental factors.

	# params	condition number	RMSE	R^2	adjusted R^2
MODEL 4	2	32.9	0.221	0.254	0.254
MODEL 5	43	8247.8	0.138	0.710	0.709
MODEL 6	42	8233.6	0.142	0.692	0.691

Table 3.6: Summary statistics for MODEL 4, MODEL 5 and MODEL 6.

3.5.2 Modeling VEG as a function of task accuracy

While the models constructed thus far have examined task (evaluation) accuracy as a function of VEG, we now consider the implications of the tenet's contrapositive and modeling the adjoint metric as a function of task accuracy. Again, we differentiate between the training and evaluation narratives.

During Training Following the during training narrative of the previous section we utilize a random intercept to account for observations collected from the same training attempt, and fit three LMMs as,

$$\left[\log_{10}\frac{||\lambda_1||_2}{||\lambda_T||_2}\right]_{ij} = -16.77 + 5.03 \times \text{accuracy}_{ij} + \text{training attempt}_i + \epsilon_{ij} \qquad \text{(MODEL 7)}$$

$$\begin{aligned}\left[\log_{10}\frac{||\lambda_1||_2}{||\lambda_T||_2}\right]_{ij} =& 2.22 + 4.80 \times \text{accuracy}_{ij} + \beta_2 \times \text{LR} + \beta_3 \times \text{LR}^2 + \beta_4 \times \text{dim} \\ &+ \beta_a \times \text{architecture} + \beta_b \times \text{dataset} + \beta_c \times \text{order} + \beta_d \times \text{orientation} \\ &+ \tau_a \times (\text{LR} * \text{architecture}) + \tau_a \times (\text{LR}^2 * \text{architecture}) \\ &+ \tau_{ad} \times (\text{architecture} * \text{orientation}) + \text{training attempt}_i + \epsilon_{ij}\end{aligned} \qquad \text{(MODEL 8)}$$

$$\begin{aligned}\Big[\log_{10}\frac{||\lambda_1||_2}{||\lambda_T||_2}\Big]_{ij} =& 0.715 + \beta_2 \times \text{LR} + \beta_3 \times \text{LR}^2 + \beta_4 \times \dim \\ &+ \beta_a \times \text{architecture} + \beta_b \times \text{dataset} + \beta_c \times \text{order} + \beta_d \times \text{orientation} \\ &+ \tau_a \times (\text{LR} * \text{architecture}) + \tau_a \times (\text{LR}^2 * \text{architecture}) \\ &+ \tau_{ad} \times (\text{architecture} * \text{orientation}) + \text{training attempt}_i + \epsilon_{ij}\end{aligned} \tag{MODEL 9}$$

Table 3.7 displays the summary statistics for these models. In a similar manner to what was observed with respect to the explanatory power that the adjoint metric had during training, training accuracy alone offers negligible explanatory power in determining the level of VEG experienced during training (MODEL 7 marginal $R^2 = 0.005$). Moreover, this limited explanatory power is further corroborated by the size of the estimated coefficients for accuracy in MODEL 7 and MODEL 8 (estimates of 5.03 and 4.80, respectively). As task accuracy is bound to the interval $[0, 1]$, the estimated coefficients imply a variation in the adjoint metric no greater than 5.03 irrespective of the task accuracy.

	# params	condition number	RMSE	conditional R^2	marginal R^2
MODEL 7	2	4.2	7.248	0.897	0.005
MODEL 8	42	8357.6	7.250	0.899	0.706
MODEL 9	41	8243.9	7.261	0.899	0.701

Table 3.7: Summary statistics for MODEL 7, MODEL 8 and MODEL 9.

At Terminal Iterate Lastly we formulate a fixed effects linear model capturing the tenet's contrapositive implications. Namely, we model the adjoint metric computed over the evaluation set as a function of task evaluation accuracy in MODEL 10; as a function of task evaluation accuracy and the experimental factors in MODEL 11; and as a function of the experimental factors alone in MODEL 12.

$$\Big[\log_{10}\frac{||\lambda_1||_2}{||\lambda_T||_2}\Big]_i = -36.65 + 45.94 \times \text{eval. accuracy} + \epsilon_i \tag{MODEL 10}$$

$$\begin{aligned}\Big[\log_{10}\frac{||\lambda_1||_2}{||\lambda_T||_2}\Big]_i =& 2.18 + 19.31 \times \text{eval. accuracy}_i + \beta_2 \times \text{LR} + \beta_3 \times \text{LR}^2 + \beta_4 \times \dim \\ &+ \beta_a \times \text{architecture} + \beta_b \times \text{dataset} + \beta_c \times \text{order} + \beta_d \times \text{orientation} \\ &+ \tau_a \times (\text{LR} * \text{architecture}) + \tau_a \times (\text{LR}^2 * \text{architecture}) \\ &+ \tau_{ad} \times (\text{architecture} * \text{orientation}) + \epsilon_i\end{aligned} \tag{MODEL 11}$$

$$\begin{aligned}\Big[\log_{10}\frac{||\lambda_1||_2}{||\lambda_T||_2}\Big]_i = &-1.10 + \beta_2 \times \mathrm{LR} + \beta_3 \times \mathrm{LR}^2 + \beta_4 \times \dim \\ &+ \beta_a \times \text{architecture} + \beta_b \times \text{dataset} + \beta_c \times \text{order} + \beta_d \times \text{orientation} \\ &+ \tau_a \times (\mathrm{LR} * \text{architecture}) + \tau_a \times (\mathrm{LR}^2 * \text{architecture}) \\ &+ \tau_{ad} \times (\text{architecture} * \text{orientation}) + \epsilon_i\end{aligned} \tag{MODEL 12}$$

Similar to what was observed when contrasting the training and terminal iterate narratives with respect to the models constituting the tenet's claim, evaluation accuracy is a more informative predictor of VEG at the terminal iterate than accuracy (measured on the training data) is during training. In particular, evaluation accuracy alone accounts for 25.4% of the variation in the adjoint metric (see Table 3.8) and in MODEL 10 its effect estimated as 45.94. However, the difference in explanatory power between MODEL 11 and MODEL 12 is less than 1.5% suggesting once additional covariates are accounted for, the benefit in explanatory power of including evaluation accuracy as a predictor is marginal. This latter point is further demonstrated by a substantial drop in the estimated coefficient for evaluation accuracy in MODEL 11 (19.31) in comparison to that of MODEL 10 (45.94).

	# params	condition number	RMSE	R^2	adjusted R^2
MODEL 10	2	5.1	20.172	0.254	0.254
MODEL 11	43	8418.8	11.025	0.777	0.777
MODEL 12	42	8233.6	11.362	0.763	0.763

Table 3.8: Summary statistics for MODEL 10, MODEL 11 and MODEL 12.

3.6 Revisiting the Mechanisms of Latching

In this section, we revisit the original work that offers explanation of why encoding long-term memory in RNNs is so difficult [9]. The framework presented in this work takes a dynamical systems viewpoint and poses RNNs as systems that learn hyperbolic attractors. Using this framework, they analyze the ability of RNNs to latch to input information over long periods of time. Through this lens of analysis, they mathematically deduce that only one of two possible conditions (see Theorems 1 and 4 of [9]) can occur: the system is very sensitive to noise, or the derivatives of the cost at time t with respect to the system activations x_0 converge exponentially to 0 as t increases. While their conclusions are informative in describing the learning process in RNNs when their parameterization generates hyperbolic attractors, we utilize the LTG latching experiment to demonstrate that RNNs can readily learn dynamics that are not hyperbolic in nature.

Recall that the LTG experiment (see Section 3.3.2) was designed to test the ability of an RNN to *latch* to label relevant information, which summarized, is the ability of a RNN to store information through time. In the case of the LTG23 experiment, the RNN must be able to latch to input u_{23} for an arbitrary amount of time, T. Additionally, the latter inputs, $\{u_t\}_{t=24}^{T}$, do not carry label relevant information, but will still impact the evolution of the state dynamics. Hence, the model must both latch to label information provided at u_{23} and also avoid deleting this information in the subsequent system evolution from $t = 24, \ldots, T$. The ability to do so is referred to as *robust latching* [9].

Using the notation introduced in section 3.2 for deriving our VEG metric and the LTG23 experiment, we reformulate the possible conditions that can arise according to the analysis of [9]. If the learned attractor is hyperbolic, then either

1. The system is sensitive to noise: in other words, there is adequate gradient information provided at time $t = 23$, (i.e., $||\lambda_{23}||_2 > 0$), but contributions to the gradient from the remaining time points are also substantial; or

2. $||\lambda_{23}||_2 \approx 0$ owing to the gradient of the loss (cost) at time t with respect to the system states, x_t, converging exponentially to 0 as the time horizon increases.

Hence, if neither of these situations are observed, then there is evidence to support that the RNN's dynamics do not correspond to a hyperbolic attractor. As we will now illustrate using the LTG experiment, neither of these situations are observed, which suggests that the RNN's dynamics do not correspond to a hyperbolic attractor.

In Fig. 3.6, we visualize the learned dynamics of M_{40}. The left image of Fig. 3.6 (displayed in red) are the norms of the recurrent states generated from a batch of observations sampled from D_{40} during the forward process (i.e., $||x_t||_2$ for $t = 1, \ldots, 40$); and the right image (blue) is an observation normalized figure of the backward adjoint dynamics (i.e., $||\lambda_t||_2$ for $t = 1, \ldots, 40$).[7] Notably, the backward adjoint figure corroborates the model's ability to robustly latch to u_{23}, evidenced by the emphasis of $||\lambda_{23}||_2$ relative to the other time indices, as well as its near zero task error rate.

[7] In the latter we perform normalization at an observation level in the following manner. Each observation pair, $(\{u_t^i\}_{t=1}^{40}, y^i)$, generates the set $\Lambda^i = \{||\lambda_1^i||_2, ||\lambda_2^i||_2, \ldots, ||\lambda_T^i||_2\}$, which we then normalize each value as $||\lambda_t^i||_2 / ||\Lambda^i||_2$ before plotting.

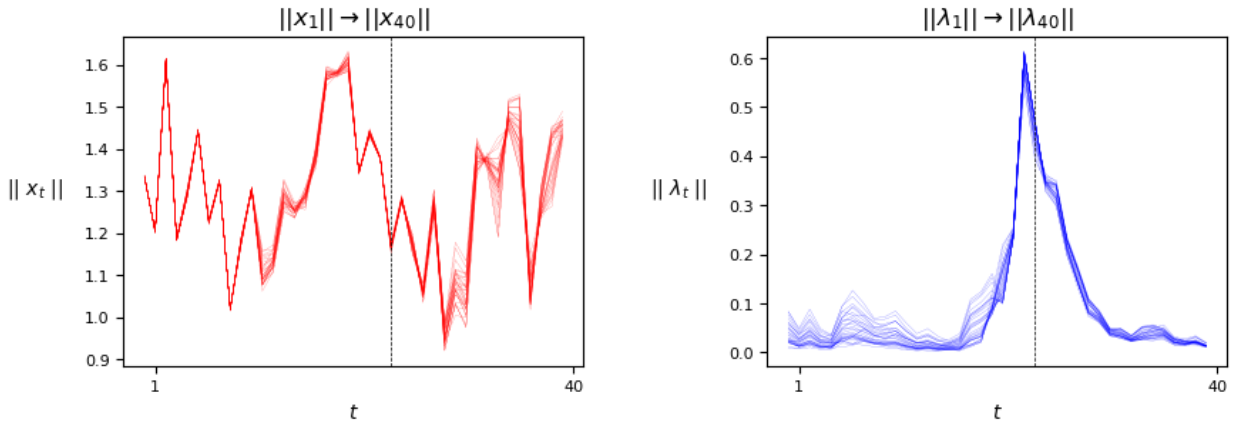


Figure 3.6: LTG23 state (red) and adjoint (blue) trajectories for a temporal horizon of $T = 40$. In both figures the dashed vertical line is positioned at $t = 23$, corresponding to the time index that is correlated to the label.

While attractor dynamics are typically not amenable to empirical observation as they are limiting behaviors, we leverage the time-shared parameterization of RNNs such that we can (loosely) illustrate the limiting system dynamics. To do so, we analyze the behavior of M_{80}, M_{160}, M_{400} and M_{800}, which retained the recurrent weight of M_{40} and only had the output layer trained to an error rate of less than 1%.

If the recurrent parameterization learned by the RNN generates an attractive set across the input observations than $||x_t||_2$ should remain bounded for all $t = \{1, 2, \ldots, T\}$. And indeed, this is observed as illustrated by Fig. 3.7 where (forward) state trajectories are displayed for each of the additional RNNs (each processed on a single batch drawn from their respective data sets, D_T).

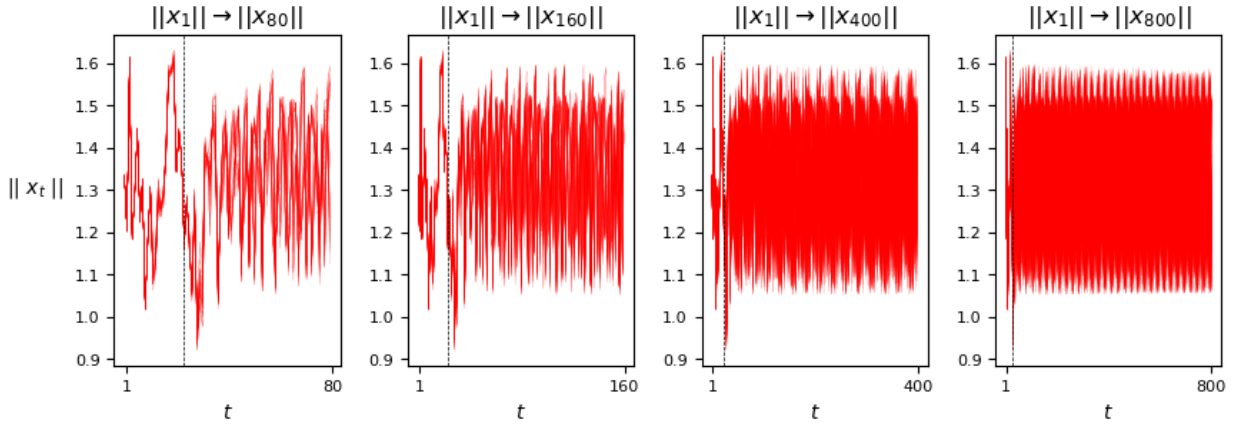


Figure 3.7: LTG23 state trajectories for temporal horizons of $T = \{80, 160, 400, 800\}$. In all four figures, the trajectories remain bounded through time.

Furthermore, if the attractive set generated over the input observations is hyperbolic, then by the conditions derived in [9], as T increases, then $||\lambda_t||_2$ will either exponentially vanish or explode. Again, in both cases, the model should fail to learn the task as expanding adjoints implies an inability to robustly latch, and vanishing adjoints implies an inability to latch. However, as we illustrate in Fig 3.8, the adjoint states do not behave in this manner for any of the models. Moreover, each of these models successfully learn the task (classification error rate less than 1%) and robustly latch to u_{23} as evidenced by a sizable increase in magnitude of $||\lambda_{23}||_2$ relative to the other time indices.

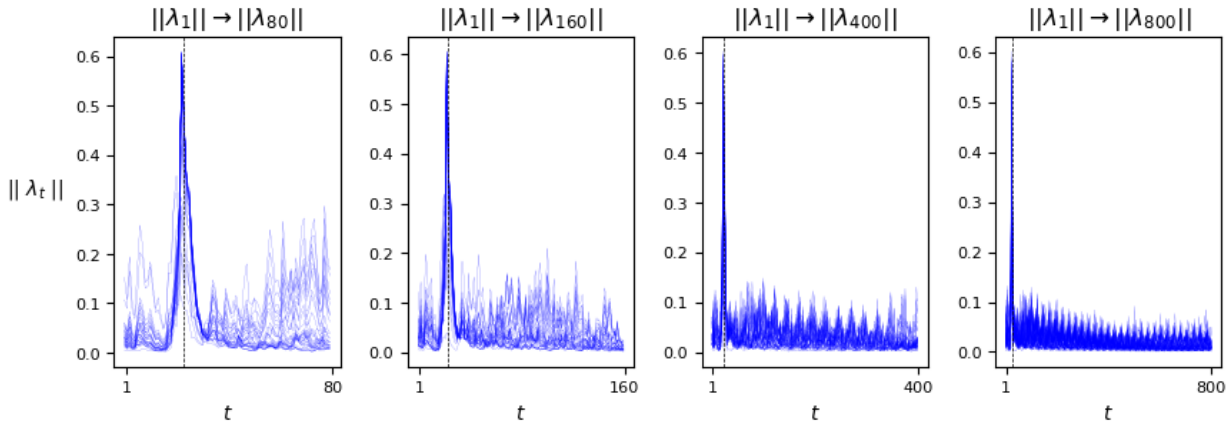


Figure 3.8: LTG23 adjoint trajectories for temporal horizons of $T = \{80, 160, 400, 800\}$. In all four figures, the trajectories do not vanish or explode through time, but rather are able to emphasize position $t = 23$, corresponding to the time index relevant for correct label classification.

These observations illustrate that the RNN's dynamics seem to generate an attractive set over the input (Fig. 3.7) and robustly latch to u_{23}. However, the RNN's dynamics behave in a manner that is not prescribed by the conditions that would appear if the attractive set was hyperbolic (Fig. 3.8). Thus, while the original framework presented in [9] is accurate in relation to its assumed framework (i.e., RNNs as generators of hyperbolic attractors), it does not account for RNN's capacity of learning non-hyperbolic attractor dynamics.

3.7 Conclusion

In this chapter we examined the validity of the belief (i.e., tenet) *if VEG occurs, RNNs will learn long-term dependencies poorly.* In doing so, we re-examined the original analysis that mathematically showed VEG to be debilitating to RNNs learning and maintaining long-term memory when framed as generators of hyperbolic attractors. Using a simple

designed experiment, we expanded upon their analysis by demonstrating that RNNs are capable of encoding long-term memory by means of non-hyperbolic attractor dynamics. To empirically investigate the tenet we trained more than 40,000 RNNs on thirty different tasks that all require learning long-term dependencies. Using the adjoints generated during backpropagation, we developed a metric that quantifies the degree of VEG experienced, and used this to interrogate the tenet's statement. Our experimental results yielded counter examples to the tenet and its contrapositive, demonstrating: when VEG occurs, RNNs are still capable of learning long-term dependencies; and when an RNN learns long-term dependencies well they can also experience VEG.

In addition to the counter examples, we constructed linear models where their structure mirrored the implications of the VEG tenet and its contrapositive. Namely, we quantified both the impact of VEG on task accuracy, and the impact of task accuracy on the degree of VEG experienced. In both settings, we observed very limited explanatory power between VEG and RNN learning, and when variables describing the underlying RNN architecture, hyperparameters and task included as additional covariates the relationship between RNN learning and VEG became negligible. Furthermore, the factors that were most predictive in determining the learning quality of a model were the model architecture, the optimization learning rate, and the interaction between these two factors. However, these relationships appeared idiosyncratic to the task, and thus cannot be confidently determined a priori to training.

In conclusion, the empirical evidence presented in this chapter support the idea that the causal relationship between VEG and the ability of an RNN to learn long-term dependencies is widely misconstrued and often exaggerated. Rather, the empirical evidence generated in this work is more aptly aligned with VEG being a commonly observed symptom when RNNs learn poorly, but not a prohibitive factor to RNNs learning long-term dependencies. Ultimately, we hope that this work can provide a shift in perspective on the understanding of VEG and reinvigorate the exploration of the ailments of learning long-term dependencies with RNNs.

4 SUMMARY

Throughout this thesis, the learning characteristics of RNNs were investigated. In particular, each chapter contextualized RNN learning through a different perspective and question of interest. Namely,

1. In Chapter 1, we studied the question of how can one RNN be compared to another, and contextualized our discussion in relation to benchmarking and task generalization.

2. In Chapter 2, we introduced a second-order optimization method to address the question of whether the gradient dynamics generated by backpropagation can be controlled.

3. In Chapter 3, we reexamined the tenet: if VEG occurs, then an RNN will learn long-term time dependencies poorly, and asked whether this is true.

While these chapters were motivated by unique learning questions, the analysis of each revealed a shared theme. Namely, the quality in which an RNN learns is incredibly sensitive to the interactions between the RNN architecture, training hyperparameters and task features. Furthermore, these interactions can (and will) directly influence the gradient dynamics learned by the model over the data. In particular, (inappropriately) large learning rates can readily induce the gradient dynamics described by VEG, and also poor learning quality. However, as corroborated in Chapter 3, VEG gradient dynamics alone, are not predictive features of RNN learning quality. Again, we observed that the interactions between RNN architecture and hyperparameters were much stronger predictors of RNN learning quality.

These observations will motivate and guide my future work. In particular, as I move forward in my career, I am interested in developing methods that improve the robustness of RNNs to hyperparameter selection, as well as developing metrics that quantify an architecture's sensitivity to hyperparameter selection.

BIBLIOGRAPHY

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Kanav Anand, Ziqi Wang, Marco Loog, and Jan van Gemert. Black magic in deep learning: How human skill impacts network training, 2020.

[3] Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks, 2016.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2014.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[6] Aristotelis Ballas and Christos Diou. A domain generalization approach for out-of-distribution 12-lead ECG classification with convolutional neural networks. In *2022 IEEE Eighth International Conference on Big Data Computing Service and Applications* (*BigDataService*). IEEE, aug 2022.

[7] Andrei Barbu, David Mayo, Julian Alverio, William Luo, Christopher Wang, Dan Gutfreund, Josh Tenenbaum, and Boris Katz. Objectnet: A large-scale bias-controlled dataset for pushing the limits of object recognition models. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[8] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Int. Res.*, 47(1):253–279, may 2013.

[9] Y. Bengio, P. Frasconi, and P. Simard. The problem of learning long-term dependencies in recurrent networks. In *IEEE International Conference on Neural Networks*, pages 1183–1188 vol.3, 1993.

[10] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[11] Nils Bertschinger, Thomas Natschläger, and Robert Legenstein. At the edge of chaos: Real-time computations and self-organized criticality in recurrent neural networks., 01 2004.

[12] Samuel R. Bowman and George E. Dahl. What will it take to fix benchmarking in natural language understanding? *CoRR*, abs/2104.02145, 2021.

[13] Morton B. Brown and Alan B. Forsythe. Robust tests for the equality of variances. *Journal of the American Statistical Association*, 69(346):364–367, 1974.

[14] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[15] Yuxuan Cai, Yizhuang Zhou, Qi Han, Jianjian Sun, Xiangwen Kong, Jun Li, and Xiangyu Zhang. Reversible column networks, 2023.

[16] Bo Chang, Minmin Chen, Eldad Haber, and Ed H. Chi. Antisymmetricrnn: A dynamical system view on recurrent neural networks, 2019.

[17] Bo Chang, Minmin Chen, Eldad Haber, and Ed H. Chi. Antisymmetricrnn: A dynamical system view on recurrent neural networks, 2019.

[18] Luonan Chen and K. Aihara. Strange attractors in chaotic neural networks. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 47(10):1455–1468, 2000.

[19] Luonan Chen and Kazuyuki Aihara. Chaos and asymptotical stability in discrete-time neural networks. *Physica D: Nonlinear Phenomena*, 104(3):286–325, 1997.

[20] Minmin Chen. Minimalrnn: Toward more interpretable and trainable recurrent neural networks, 2018.

[21] KyungHyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259, 2014.

[22] Jasmine Collins, Jascha Sohl-Dickstein, and David Sussillo. Capacity and trainability in recurrent neural networks, 2017.

[23] Tim Cooijmans, Nicolas Ballas, Cesar Laurent, Caglar Gulcehre, and Aaron Courville. Recurrent batch normalization, 2017.

[24] Christian Dallago, Jody Mou, Jody Mou, Kadina Johnston, Bruce Wittmann, Nicholas Bhattacharya, Samuel Goldman, Ali Madani, and Kevin Yang. Flip: Benchmark tasks in fitness landscape inference for proteins. In J. Vanschoren and S. Yeung, editors, *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1. Curran, 2021.

[25] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. IEEE, 2009.

[26] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[28] Mingyu Ding, Bin Xiao, Noel Codella, Ping Luo, Jingdong Wang, and Lu Yuan. Davit: Dual attention vision transformers, 2022.

[29] Matthew Dixon and Justin London. Financial forecasting with α-rnns: A time series modeling approach. *Frontiers in Applied Mathematics and Statistics*, 6, 2021.

[30] Ahmed Elnaggar, Hazem Essam, Wafaa Salah-Eldin, Walid Moustafa, Mohamed Elkerdawy, Charlotte Rochereau, and Burkhard Rost. Ankh: Optimized protein language model unlocks general-purpose modelling, 2023.

[31] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation matters in deep rl: A case study on ppo and trpo. In *International Conference on Learning Representations*, 2020.

[32] N. Benjamin Erichson, Omri Azencot, Alejandro Queiruga, Liam Hodgkinson, and Michael W. Mahoney. Lipschitz recurrent neural networks, 2020.

[33] Mark R Fahey, Yuri Alexeev, Bill Allcock, Benjamin S Allen, Ramesh Balakrishnan, Anouar Benali, Liza Booker, Ashley Boyle, Laural Briggs, Edouard Brooks, et al. Theta and mira at argonne national laboratory. In *Contemporary High Performance Computing*, pages 31–61. CRC Press, 2019.

[34] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[35] F.A. Gers and J. Schmidhuber. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194 vol.3, 2000.

[36] Andreas Griewank et al. On automatic differentiation. *Mathematical Programming: recent developments and applications*, 6(6):83–107, 1989.

[37] Eldad Haber and Lars Ruthotto. Stable architectures for deep neural networks. *Inverse Problems*, 34(1):014004, December 2017.

[38] Jung Min Han, Yu Qian Ang, Ali Malkawi, and Holly W. Samuelson. Using recurrent neural networks for localized weather prediction with combined use of public airport data and on-site measurements. *Building and Environment*, 192:107601, 2021.

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[40] Mikael Henaff, Arthur Szlam, and Yann LeCun. Recurrent orthogonal networks and long-memory tasks. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2034–2042, New York, New York, USA, 20–22 Jun 2016. PMLR.

[41] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters, 2019.

[42] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[43] Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[44] Stephen Hudson, Jeffrey Larson, John-Luke Navarro, and Stefan M. Wild. libEnsemble: A library to coordinate the concurrent evaluation of dynamic ensembles of calculations. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):977–988, apr 2022.

[45] Abigail Z. Jacobs and Hanna Wallach. Measurement and fairness. In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. ACM, mar 2021.

[46] Li Jing, Yichen Shen, Tena Dub?ek, John Peurifoy, Scott Skirlo, Yann LeCun, Max Tegmark, and Marin Solja?i? Tunable efficient unitary neural networks (eunn) and their application to rnns, 2017.

[47] L. Johnston and V. Patel. Second-order sensitivity methods for robustly training recurrent neural network models. *IEEE Control Systems Letters*, 5(2):529–534, 2021.

[48] Liam Johnston and Vivak Patel. Second-order sensitivity methods for robustly training recurrent neural network models. *IEEE Control Systems Letters*, 5(2):529–534, 2021.

[49] Cijo Jose, Moustpaha Cisse, and Francois Fleuret. Kronecker recurrent units, 2017.

[50] John M. Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Zídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Andy Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David A. Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with alphafold. *Nature*, 596:583 – 589, 2021.

[51] Kaggle and EyePacs. Kaggle diabetic retinopathy detection, jul 2015.

[52] Steven Kapturowski, Georg Ostrovski, John Quan, Rémi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. In *International Conference on Learning Representations*, 2018.

[53] Sohier Dane Karthik, Maggie. Aptos 2019 blindness detection, 2019.

[54] Minsu Kim, Joanna Hong, and Yong Man Ro. Lip to speech synthesis with visual context attentional gan. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 2758–2770. Curran Associates, Inc., 2021.

[55] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, San Diega, CA, USA, 2015.

[56] David Krueger and Roland Memisevic. Regularizing rnns by stabilizing activations, 2016.

[57] Andriy Kryshtafovych, Torsten Schwede, Maya Topf, Krzysztof Fidelis, and John Moult. Critical assessment of methods of protein structure prediction (casp)—round xiv. *Proteins: Structure, Function, and Bioinformatics*, 89(12):1607–1617, 2021.

[58] Emmanuel Laporte and Patrick Le Tallec. Numerical methods in sensitivity analysis and shape optimization. In *Modeling and Simulation in Science, Engineering and Technology*, 2002.

[59] Thomas Laurent and James H. von Brecht. A recurrent neural network without chaos. *CoRR*, abs/1612.06212, 2016.

[60] Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. A simple way to initialize recurrent networks of rectified linear units, 2015.

[61] Yann LeCun. Learning process in an asymmetric threshold network. In *Disordered systems and biological organization*, pages 233–240. Springer, 1986.

[62] Yann LeCun, D Touresky, G Hinton, and T Sejnowski. A theoretical framework for back-propagation. In *Proceedings of the 1988 connectionist models summer school*, volume 1, pages 21–28, 1988.

[63] Bryon C. Lewis and Albert E. Crews. The evolution of benchmarking as a computer performance evaluation technique. *MIS Q.*, 9:7–16, 1985.

[64] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models, 2022.

[65] Mario Lezcano-Casado and David Martinez-Rubio. Cheap orthogonal constraints in neural networks: A simple parametrization of the orthogonal and unitary group, 2019.

[66] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2015.

[67] Feifei Liu, Chengyu Liu, Lina Zhao, Xiangyu Zhang, Xiaoling Wu, Xiaoyan Xu, Yulin Liu, Caiyun Ma, Shoushui Wei, Jianqing Li, and Eddie Ng. An open access database for evaluating the algorithms of electrocardiogram rhythm and morphology abnormality detection. *Journal of Medical Imaging and Health Informatics*, 8:1368–1373, 09 2018.

[68] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[69] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[70] Gábor Melis, Chris Dyer, and Phil Blunsom. On the state of the art of evaluation in neural language models. In *International Conference on Learning Representations*, 2018.

[71] Tomáš Mikolov, Stefan Kombrink, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. Extensions of recurrent neural network language model. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5528–5531, 2011.

[72] Melanie Mitchell. Why ai is harder than we think, 2021.

[73] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[74] Signe Moe, Filippo Remonato, Esten Ingar Grøtli, and Jan Tommy Gravdahl. Linear antisymmetric recurrent neural networks. In Alexandre M. Bayen, Ali Jadbabaie, George Pappas, Pablo A. Parrilo, Benjamin Recht, Claire Tomlin, and Melanie Zeilinger, editors, *Proceedings of the 2nd Conference on Learning for Dynamics and Control*, volume 120 of *Proceedings of Machine Learning Research*, pages 170–178. PMLR, 10–11 Jun 2020.

[75] Shinichi Nakagawa and Holger Schielzeth. A general and simple method for obtaining r2 from generalized linear mixed-effects models. *Methods in Ecology and Evolution*, 4(2):133–142, 2013.

[76] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models, 2021.

[77] Scott W O'Leary-Kelly and Robert J. Vokurka. The empirical assessment of construct validity. *Journal of Operations Management*, 16(4):387–405, 1998.

[78] Tesla Owners Online. Tesla full self driving explained by andrej karpathy, Aug. 2021.

[79] OpenAI. Gpt-4 technical report, 2023.

[80] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks, 2013.

[81] Yao Qin, Dongjin Song, Haifeng Chen, Wei Cheng, Guofei Jiang, and Garrison Cottrell. A dual-stage attention-based recurrent neural network for time series prediction, 2017.

[82] Inioluwa Deborah Raji, Emily M. Bender, Amandalynne Paullada, Emily Denton, and Alex Hanna. AI and the everything in the whole wide world benchmark. *CoRR*, abs/2111.15366, 2021.

[83] Nils Reimers and Iryna Gurevych. Reporting score distributions makes a difference: Performance study of lstm-networks for sequence tagging. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 338–348, Copenhagen, Denmark, 09 2017.

[84] Sebastian Ruder. Challenges and Opportunities in NLP Benchmarking. `http://ruder.io/nlp-benchmarking`, 2021.

[85] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In David E. Rumelhart and James L. Mcclelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, pages 318–362. MIT Press, Cambridge, MA, 1986.

[87] T. Konstantin Rusch and Siddhartha Mishra. Unicornn: A recurrent model for learning very long time dependencies, 2021.

[88] Bryan Russell, Antonio Torralba, Kevin Murphy, and William Freeman. Labelme: A database and web-based tool for image annotation. *International Journal of Computer Vision*, 77, 05 2008.

[89] Massimo Salvi, U. Rajendra Acharya, Filippo Molinari, and Kristen M. Meiburger. The impact of pre- and post-image processing techniques on deep learning frameworks: A comprehensive review for digital pathology image analysis. *Computers in Biology and Medicine*, 128:104129, 2021.

[90] Andrew W. Senior, Richard Evans, John M. Jumper, James Kirkpatrick, L. Sifre, Tim Green, Chongli Qin, Augustin Zídek, Alexander W. R. Nelson, Alex Bridgland, Hugo Penedones, Stig Petersen, Karen Simonyan, Steve Crossan, Pushmeet Kohli, David C. Jones, David Silver, Koray Kavukcuoglu, and Demis Hassabis. Improved protein structure prediction using potentials from deep learning. *Nature*, 577:706 – 710, 2020.

[91] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[92] Nitish Srivastava, Elman Mansimov, and Ruslan Salakhutdinov. Unsupervised learning of video representations using lstms, 2016.

[93] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[94] Nils Strodthoff, Temesgen Mehari, Claudia Nagel, Philip J Aston, Ashish Sundar, Claus Graff, Jørgen K Kanters, Wilhelm Haverkamp, Olaf Dössel, Axel Loewe,

Markus Bär, and Tobias Schaeffter. Ptb-xl+, a comprehensive electrocardiographic feature dataset. *Scientific data*, 10(1):279, May 2023.

[95] Nils Strodthoff, Patrick Wagner, Tobias Schaeffter, and Wojciech Samek. Deep learning for ecg analysis: Benchmarks and insights from ptb-xl. *IEEE journal of biomedical and health informatics*, PP, 09 2020.

[96] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp, 2019.

[97] Corentin Tallec and Yann Ollivier. Can recurrent neural networks warp time?, 2018.

[98] Gouhei Tanaka, Toshiyuki Yamane, Jean Benoit Héroux, Ryosho Nakane, Naoki Kanazawa, Seiji Takeda, Hidetoshi Numata, Daiju Nakano, and Akira Hirose. Recent advances in physical reservoir computing: A review. *Neural Networks*, 115:100–123, 2019.

[99] William Taylor-Melanson, Gordon MacDonald, and Andrew Godbout. Volume-preserving recurrent neural networks (vprnn). In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10, 2021.

[100] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[101] Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, Nicolas Heess, and Yuval Tassa. dm_control: Software and tasks for continuous control. *Software Impacts*, 6:100022, nov 2020.

[102] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[103] Aaron Voelker, Ivana Kajić, and Chris Eliasmith. Legendre memory units: Continuous-time representation in recurrent neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[104] Eugene Vorontsov, Chiheb Trabelsi, Samuel Kadoury, and Chris Pal. On orthogonality and learning recurrent networks with long term dependencies, 2017.

[105] Eugene Vorontsov, Chiheb Trabelsi, Samuel Kadoury, and Chris Pal. On orthogonality and learning recurrent networks with long term dependencies. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 3570–3578. JMLR.org, 2017.

[106] Eugene Vorontsov, Chiheb Trabelsi, Samuel Kadoury, and Chris Pal. On orthogonality and learning recurrent networks with long term dependencies. *CoRR*, abs/1702.00071, 2017.

[107] Patrick Wagner, Nils Strodthoff, Ralf-Dieter Bousseljot, Dieter Kreiseler, Fatima Lunze, Wojciech Samek, and Tobias Schaeffter. Ptb-xl, a large publicly available electrocardiography dataset. *Scientific Data*, 7:154, 05 2020.

[108] Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems. *CoRR*, abs/1905.00537, 2019.

[109] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding, 2019.

[110] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *CoRR*, abs/2201.11903, 2022.

[111] Scott Wisdom, Thomas Powers, John R. Hershey, Jonathan Le Roux, and Les Atlas. Full-capacity unitary recurrent neural networks, 2016.

[112] Jiahui Yu, Zirui Wang, Vijay Vasudevan, Legg Yeung, Mojtaba Seyedhosseini, and Yonghui Wu. Coca: Contrastive captioners are image-text foundation models, 2022.

[113] Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Avnish Narayan, Hayden Shively, Adithya Bellathur, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning, 2021.

[114] Xiaohua Zhai, Xiao Wang, Basil Mustafa, Andreas Steiner, Daniel Keysers, Alexander Kolesnikov, and Lucas Beyer. Lit: Zero-shot transfer with locked-image text tuning, 2022.

[115] Jiong Zhang, Qi Lei, and Inderjit S. Dhillon. Stabilizing gradients for deep neural networks via efficient svd parameterization, 2018.

[116] Marvin Zhang, Zoe McCarthy, Chelsea Finn, Sergey Levine, and Pieter Abbeel. Learning deep neural network policies with continuous memory states, 2015.

[117] Tony Z. Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. Calibrate before use: Improving few-shot performance of language models, 2021.

[118] Hattie Zhou, Azade Nova, Aaron Courville, Hugo Larochelle, Behnam Neyshabur, and Hanie Sedghi. Teaching algorithmic reasoning via in-context learning, 2023.

A1 Task Generalization Supplementary Material

A1.1 First-order Linear Model Summary Table

Task	adj. R^2	σ
CIFAR10 Sequential None	0.50	0.13
CIFAR10 Permuted None	0.74	0.05
CIFAR10 Sequential Post	0.99	0.04
CIFAR10 Permuted Post	0.86	0.12
CIFAR10 Sequential Uniform	0.78	0.11
CIFAR10 Permuted Uniform	0.06	0.16
Fashion MNIST Sequential None	0.24	0.07
Fashion MNIST Permuted None	0.84	0.01
Fashion MNIST Sequential Post	0.69	0.20
Fashion MNIST Permuted Post	0.91	0.09
Fashion MNIST Sequential Uniform	0.58	0.12
Fashion MNIST Permuted Uniform	0.66	0.03
IMDB Sequential None	0.96	0.01
IMDB Permuted None	0.87	0.02
IMDB Sequential Post	0.92	0.13
IMDB Permuted Post	0.88	0.15
IMDB Sequential Uniform	0.97	0.02
IMDB Permuted Uniform	0.42	0.19
MNIST Sequential None	0.24	0.18
MNIST Permuted None	0.89	0.01
MNIST Sequential Post	0.98	0.07
MNIST Permuted Post	0.97	0.06
MNIST Sequential Uniform	0.86	0.14
MNIST Permuted Uniform	0.61	0.07
Reuters Sequential None	0.86	0.04
Reuters Permuted None	0.96	0.02
Reuters Sequential Post	0.95	0.04
Reuters Permuted Post	0.92	0.05
Reuters Sequential Uniform	0.76	0.05
Reuters Permuted Uniform	0.93	0.03

Table A1: Summaries of first-order linear models, (1.8), fit on data generated from one of thirty experimental tasks.

A1.2 First-order Behavior Summary Table: Convergence Time

We measure the first-order behavior *convergence time* with the variable, $Y_{\texttt{epoch}}$, which is defined in Section 1.4 as the number of training epochs completed prior to the RNN

attaining its maximum task accuracy. We forego a formal test for this behavior, and instead display the median and standard deviation of Y_{epoch} for each architecture and task in Table A2 below. Similar to the other first-order behaviors, the behavior *convergence time* varies across tasks and architecture.

dataset	orientation	order	Basic RNN	Antisymmetric RNN	Exponential RNN	GRU	Lipschitz RNN	LSTM	UnICORNN
CIFAR10	none	sequential	23 ± 1.4	24 ± 0.8	24.5 ± 0.8	23.5 ± 3.7	25 ± 1.3	25 ± 3.1	24 ± 1.2
CIFAR10	none	permute	25 ± 0.7	24 ± 1.3	24 ± 3	22 ± 2	25 ± 0.8	24 ± 5.9	21 ± 3.5
CIFAR10	post	sequential	20.5 ± 8.3	24 ± 1.3	24.5 ± 0.7	23 ± NA	24 ± 2.2	13 ± 9.2	NA ± NA
CIFAR10	post	permute	13 ± 7.1	25 ± 1.2	24.5 ± 1.1	17 ± 9.9	22 ± 1.4	1 ± NA	NA ± NA
CIFAR10	uniform	sequential	21.5 ± 5	24 ± 0.7	24 ± 1.1	22 ± 6.2	24 ± 2.6	19 ± 8.5	NA ± NA
CIFAR10	uniform	permute	24.5 ± 1.1	23.5 ± 2.8	24 ± 1.4	22 ± 5.1	23 ± 2.3	25 ± 1.3	NA ± NA
Fashion MNIST	none	sequential	24 ± 1.6	24.5 ± 0.9	25 ± 1.5	22.5 ± 2.1	24.5 ± 1.7	23.5 ± 1.6	23.5 ± 1
Fashion MNIST	none	permute	24.5 ± 0.8	24 ± 1	24.5 ± 1.7	19 ± 3.8	24 ± 1	22.5 ± 4.2	23.5 ± 1.6
Fashion MNIST	post	sequential	20 ± 5.9	24 ± 1.7	20 ± NA	21.5 ± 3.3	23.5 ± 1.6	15.5 ± 7.1	NA ± NA
Fashion MNIST	post	permute	20.5 ± 7.2	24 ± 1.4	24 ± 1.9	25 ± NA	24 ± 1.2	16.5 ± 8	NA ± NA
Fashion MNIST	uniform	sequential	24 ± 1.7	24 ± 0.9	24 ± 2.3	25 ± 3.1	24 ± 1.1	24 ± 7.5	NA ± NA
Fashion MNIST	uniform	permute	25 ± 0.9	24 ± 0.9	24 ± 2.2	23 ± 1.2	25 ± 0.7	25 ± 5.9	NA ± NA
IMDB	none	sequential	22 ± 2.9	8 ± 0.5	25 ± 0.3	22 ± 2.7	1 ± 0.5	21 ± 2.5	1 ± 0.7
IMDB	none	permute	24.5 ± 1.2	7.5 ± 1.5	6 ± 0.9	25 ± 0.9	1.5 ± 1.4	8.5 ± 2.9	2 ± 0.6
IMDB	post	sequential	7 ± 11.7	1.5 ± 0.5	10 ± 5.8	10.5 ± 10.6	2 ± 1.3	13.5 ± 6.4	NA ± NA
IMDB	post	permute	9.5 ± 7.3	2 ± 1.3	5 ± 7.1	5 ± NA	10.5 ± 1.6	5 ± NA	NA ± NA
IMDB	uniform	sequential	24 ± 2	25 ± 0.3	1 ± 4.7	19 ± 3.4	2 ± 0.5	17 ± 4.6	1 ± NA
IMDB	uniform	permute	24 ± 2.5	25 ± 0.4	11.5 ± 9.6	25 ± 1.2	3 ± 1.3	23.5 ± 1.9	NA ± NA
MNIST	none	sequential	24 ± 1.9	25 ± 1.1	24 ± 2.1	22 ± 2.3	21.5 ± 2.9	22.5 ± 4.7	24 ± 1.1
MNIST	none	permute	24 ± 3	25 ± 0.8	24 ± 2	23 ± 1.5	22.5 ± 2.5	23.5 ± 1.9	24 ± 0.9
MNIST	post	sequential	12.5 ± 7.2	25 ± 1	25 ± NA	16 ± 5.7	24 ± 1.6	1 ± 0.5	NA ± NA
MNIST	post	permute	19 ± 6.2	25 ± 2.1	24 ± 0.9	24 ± NA	24.5 ± 0.7	18 ± 9.2	NA ± NA
MNIST	uniform	sequential	24 ± 6.7	24 ± 3.2	24 ± 1.2	25 ± 9	22.5 ± 1	12 ± 6.6	NA ± NA
MNIST	uniform	permute	25 ± 1.5	24 ± 1.1	24 ± 1.4	21.5 ± 0.7	25 ± 0.5	24 ± 0.8	NA ± NA
Reuters	none	sequential	18.5 ± 5.4	23.5 ± 2.9	8.5 ± 6.5	17.5 ± 4.9	13.5 ± 4.6	10.5 ± 4.2	11.5 ± 7.1
Reuters	none	permute	19.5 ± 6.7	21.5 ± 2.6	20 ± 2.7	16.5 ± 4.4	8.5 ± 6.3	12.5 ± 5.7	8.5 ± 4.8
Reuters	post	sequential	1 ± 6.9	15 ± 8.5	9 ± 2.9	2 ± 0.4	22 ± 4.1	1 ± 0.6	22 ± 6.7
Reuters	post	permute	2.5 ± 6.1	24 ± 1.7	8 ± 1.3	1.5 ± 0.5	21.5 ± 3.3	1 ± 0.5	1 ± 8.2
Reuters	uniform	sequential	18.5 ± 8.9	14.5 ± 7.5	2 ± 0.3	2 ± 0.4	17.5 ± 6.1	24 ± 1.6	1 ± 10.4
Reuters	uniform	permute	24.5 ± 9.7	25 ± 0.7	24 ± 0.7	2 ± 0	3.5 ± 0.7	23.5 ± 2.1	25 ± 0.4

Table A2: Median epoch until maximum task accuracy (± standard deviation) at the maximizing hyperparameter configuration, $h^* \in H$, for each RNN architecture and task.

A1.3 Second-order Linear Model Summary Table

A2 Derivation of Gradient Flow Equations and Control

A2.1 Linear RNN Adjoint System

The linear RNN solves the following supervised optimization problem

$$\begin{aligned} \min_{\theta \in \Theta} \quad & F(y, \hat{y}) \\ \text{s.t} \quad & x_i = W x_{i-1} + R u_i \qquad i = 1, \dots, N \\ & \hat{y} = V x_N \end{aligned} \tag{1}$$

Task	Condition Number	adj. R^2	σ
MNIST Sequential Post	13519.63	0.85	0.14
MNIST Permuted Post	13411.21	0.74	0.16
IMDB Permuted Uniform	11418.28	0.80	0.15
CIFAR10 Sequential Post	11576.63	0.90	0.10
Fashion MNIST Permuted Post	12562.04	0.80	0.15
IMDB Sequential Post	11020.76	0.85	0.16
CIFAR10 Sequential Uniform	11207.60	0.80	0.09
CIFAR10 Permuted Uniform	11781.88	0.77	0.11
MNIST Permuted Uniform	12282.17	0.79	0.15
IMDB Permuted Post	10586.32	0.85	0.16
CIFAR10 Permuted Post	35155.99	0.84	0.10
Fashion MNIST Sequential Uniform	14362.85	0.87	0.12
MNIST Sequential Uniform	12074.71	0.86	0.13
Fashion MNIST Permuted Uniform	14610.45	0.85	0.12
IMDB Sequential Uniform	5.62e+18	0.77	0.13
Fashion MNIST Sequential Post	12976.09	0.91	0.11
Fashion MNIST Permuted None	11530.65	0.87	0.10
Fashion MNIST Sequential None	11356.01	0.84	0.12
CIFAR10 Sequential None	12059.23	0.78	0.10
MNIST Permuted None	13270.05	0.84	0.13
CIFAR10 Permuted None	11780.48	0.81	0.11
MNIST Sequential None	11233.34	0.83	0.15
Reuters Sequential Uniform	12272.45	0.60	0.11
Reuters Permuted Uniform	12221.92	0.76	0.08
Reuters Sequential Post	13595.45	0.87	0.09
IMDB Permuted None	9724.36	0.86	0.12
Reuters Permuted Post	15571.87	0.88	0.08
IMDB Sequential None	10253.44	0.81	0.11
Reuters Sequential None	11794.02	0.81	0.09
Reuters Permuted None	11538.50	0.82	0.08

Table A3: Summaries of linear model, (1.11), approximating the second-order behavior *average task accuracy*.

The adjoint system can be derived from the Lagrangian

$$\mathcal{L} = F(y, \hat{y}) + \delta'(\hat{y} - Vx_N) + \sum_{j=1}^{N} \lambda_j'[x_j - Wx_{j-1} - Ru_j] \tag{2}$$

where the adjoint states are the derivatives of (2) with respect to $\hat{y}$ and x_i $i = 1, \ldots, N$. The resulting system is described by

$$\begin{aligned} \delta &= -\frac{\partial F}{\partial \hat{y}} \\ \lambda_N &= \delta V' \\ \lambda_j &= W'\lambda_{j+1} = W^{(N-j)'}\delta V' \qquad j = 1, \ldots, N-1 \end{aligned} \tag{3}$$

A2.2 Linear RNN Coadjoint System

The coadjoint method solves a similar optimization to (1) with the addition of an adjoint control function $G(\Lambda)$. The coadjoint method places additional constraints on the objective such that the form of (3) must be obeyed.

$$\begin{aligned} \min_{\theta \in \Theta} \quad & F(y, \hat{y}) + G(\Lambda) \\ \text{s.t} \quad & x_i = Wx_{i-1} + Ru_i \qquad i = 1, \ldots, N \\ & \hat{y} = Vx_N \\ & \delta = -\frac{\partial F}{\partial \hat{y}} \\ & \lambda_N = \delta V' \\ & \lambda_j = \lambda'_{j+1} W \qquad j = 1, \ldots, N-1 \end{aligned} \tag{4}$$

The associated Lagrangian has form

$$\begin{aligned} \mathcal{L} =& F(y, \hat{y}) + G(\Lambda) + \phi'\left[\hat{y} - Vx_N\right] + \sum_{j=1}^{N} \gamma'_j \left[x_j - Wx_{j-1} - Ru_j\right] \\ &+ \psi'\left[\delta + \frac{\partial F}{\partial \hat{y}}\right] + \alpha'_N \left[\lambda_N - \delta V'\right] + \sum_{j=1}^{N-1} \alpha'_j \left[\lambda_j - W'\lambda_{j+1}\right] \end{aligned} \tag{5}$$

Differentiating (5) with respect to $\hat{y}, x_i$ and the adjoint states δ, λ_i for $i = 1, \ldots, N$ (where the latter are now viewed as constants) we can form the coadjoint system (shown below).

$$\begin{aligned}
\phi &= -\frac{\partial F}{\partial \hat{y}} \\
\gamma_N &= \phi V' \\
\gamma_j &= W'\gamma_{j+1} \qquad j = 1, \ldots, N-1 \\
\psi &= V\alpha_N \\
\alpha_j &= -\frac{\partial G}{\partial \lambda_j} + W\alpha_{j-1} \qquad j = 2, \ldots, N \\
\alpha_1 &= -\frac{\partial G}{\partial \lambda_1}
\end{aligned} \tag{6}$$

A2.3 Adjoint System Derivation (Nonlinear)

Given an RNN, an example $(\{u_1, \ldots, u_N\}, y)$, and a loss function $F(y, \hat{y})$, we aim to solve

$$\begin{aligned}
\min_{\theta \in \Theta} \quad & F(y, \hat{y}) \\
\text{s.t} \quad & x_j = \sigma(Wx_{j-1} + Ru_j + b_1) \qquad j = 1, \ldots, N \\
& \hat{y} = \phi(Vx_N + b_0)
\end{aligned} \tag{7}$$

where $\theta = \{V, W, R, b_1, b_0\}$. Then the adjoint system can be derived using Lagrangian formalism where we define the Lagrangian as,

$$\mathcal{L} = F(y, \hat{y}) - \frac{\partial F}{\partial \hat{y}}\Big[\hat{y} - \phi(Vx_N + b_0)\Big] + \sum_{j=1}^{N} \lambda_j'\Big[x_j - \sigma(Wx_{j-1} + Ru_j + b_1)\Big] \tag{8}$$

The adjoint system is derived by differentiating (8) with respect to x_i for $i = 1, \ldots, N$.

$$\frac{\partial \mathcal{L}}{\partial x_N} = V'\phi^{(1)}(Vx_N + b_0)'\frac{\partial F}{\partial \hat{y}} + \lambda_N \tag{9}$$

$$\frac{\partial \mathcal{L}}{\partial x_j} = \lambda_j - W'\sigma^{(1)}(Wx_j + Ru_{j+1} + b_1)'\lambda_{j+1} \qquad j = 1, \ldots, N-1 \tag{10}$$

where $\phi^{(1)}$ and $\sigma^{(1)}$ represent the Jacobians of ϕ and σ, respectively. The adjoint system is then the solution vectors to these gradients.

$$\lambda_j = \begin{cases} -V'\phi^{(1)}(Vx_N + b_0)'\frac{\partial F}{\partial \hat{y}} & j = N \\ W'\sigma^{(1)}(Wx_i + Ru_{i+1} + b_1)'\lambda_{j+1} & j < N, \end{cases} \tag{11}$$

A2.4 Coadjoint System Derivation (Nonlinear)

We now define a differentiable function of the adjoint variables $G(\{\lambda_j\})$ and solve a similar optimization problem to (7),

$$\begin{aligned}
\min_{\theta\in\Theta} \quad & F(y,\hat{y}) + G(\{\lambda_j\}) \\
\text{s.t} \quad & x_j = \sigma(Wx_{j-1} + Ru_j + b_1) \qquad j = 1,\dots,N \\
& \hat{y} = \phi(Vx_N + b_0) \\
& \lambda_j = W'\sigma^{(1)}(Wx_j + Ru_{j+1} + b_1)'\lambda_{j+1} \qquad j = 1,\dots,N-1 \\
& \lambda_N = -V'\phi^{(1)}(Vx_N + b_0)'\frac{\partial F}{\partial \hat{y}}
\end{aligned} \tag{12}$$

where the adjoint states in (6) are viewed as constants. In a similar fashion we can derive the coadjoint states by forming a Lagrangian,

$$\begin{aligned}
\mathcal{L} =& F(y,\hat{y}) + G(\{\lambda_j\}) + \sum_{j=1}^{N} \alpha_j' \Big[x_j - \sigma(Wx_{j-1} + Ru_j + b_1) \Big] \\
&+ \sum_{j=1}^{N-1} \gamma_j' \Big[\lambda_j - W'\sigma^{(1)}(Wx_j + Ru_{j+1} + b_1)'\lambda_{j+1} \Big] \\
&+ \gamma_N' \Big[\lambda_N + V'\phi^{(1)}(Vx_N + b_0)'\frac{\partial F}{\partial \hat{y}} \Big] + \kappa' \Big[\hat{y} - \phi(Vx_N + b_0) \Big]
\end{aligned} \tag{13}$$

The coadjoint system is then the solution vectors of differentiating (13) by $\{x_j\}$ and $\{\lambda_j\}$ for $j = 1,\dots,N$.

$$\frac{\partial \mathcal{L}}{\partial \lambda_1} = \frac{\partial G}{\partial \lambda_1} + \gamma_1 \tag{14}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_j} = \frac{\partial G}{\partial \lambda_j} + \gamma_j - \sigma^{(1)}(Wx_{j-1} + Ru_j + b_1)W\gamma_{j-1} \qquad j = 2,\dots,N \tag{15}$$

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial x_j} =& \alpha_j - W'\sigma^{(1)}(Wx_j + Ru_{j+1} + b_1)'\alpha_{j+1} \\
&- W'\Big(\sum_{k=1}^{d} \lambda_{j+1}[k]\sigma_{[k]}^{(2)}(Wx_j + Ru_{j+1} + b_1)' \Big) W\gamma_j \qquad j = 1,\dots,N-1.
\end{aligned} \tag{16}$$

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial x_N} =& \alpha_N - V'\phi^{(1)}(Vx_N + b_0)\frac{\partial F}{\partial \hat{y}} - V'\phi^{(1)}(Vx_N + b_0)'\frac{\partial^2 F}{\partial \hat{y}^2}\phi^{(1)}(Vx_N + b_0)V\gamma_N \\
&+ V'\Big(\sum_{k=1}^{l} \frac{\partial F}{\partial \hat{y}_{[k]}}\phi_{[k]}^{(2)}(Vx_N + b_0)' \Big) V\gamma_N
\end{aligned} \tag{17}$$

where $[k]$ represents the k^{th} component of a given quantity; and $\phi^{(2)}_{[k]}$ and $\sigma^{(2)}_{[k]}$ represent the Hessians of $\phi_{[k]}$ and $\sigma_{[k]}$, respectively. Solving for the solution vectors in (14) and (15) yields the forward coadjoint system,

$$\gamma_j = \begin{cases} -\frac{\partial G}{\partial \lambda_1} & j = 1 \\ \sigma^{(1)}(Wx_{j-1} + Ru_j + b_1)W\gamma_{j-1} - \frac{\partial G}{\partial \lambda_j} & j > 1, \end{cases} \tag{18}$$

and doing the same for (16) and (17) yields the backward coadjoint system,

$$\alpha_j = \begin{cases} V'\phi^{(1)}(Vx_N + b_0)\dfrac{\partial F}{\partial \hat{y}} + V'\phi^{(1)}(Vx_N + b_0)'\dfrac{\partial^2 F}{\partial \hat{y}^2}\phi^{(1)}(Vx_N + b_0)V\gamma_N \\ \qquad - V'\left(\displaystyle\sum_{k=1}^{\ell} \frac{\partial F}{\partial \hat{y}_{[k]}}\phi^{(2)}_{[k]}(Vx_N + b_0)'\right)V\gamma_N & j = N \\ W'\sigma^{(1)}(Wx_j + Ru_{j+1} + b_1)'\alpha_{j+1} \\ \quad + W'\left(\displaystyle\sum_{k=1}^{d} \lambda_{j+1}[k]\sigma^{(2)}_{[k]}(Wx_j + Ru_{j+1} + b_1)'\right)W\gamma_j & j < N, \end{cases} \tag{19}$$

A3 Additional Figures

A3.1 LTG Figures

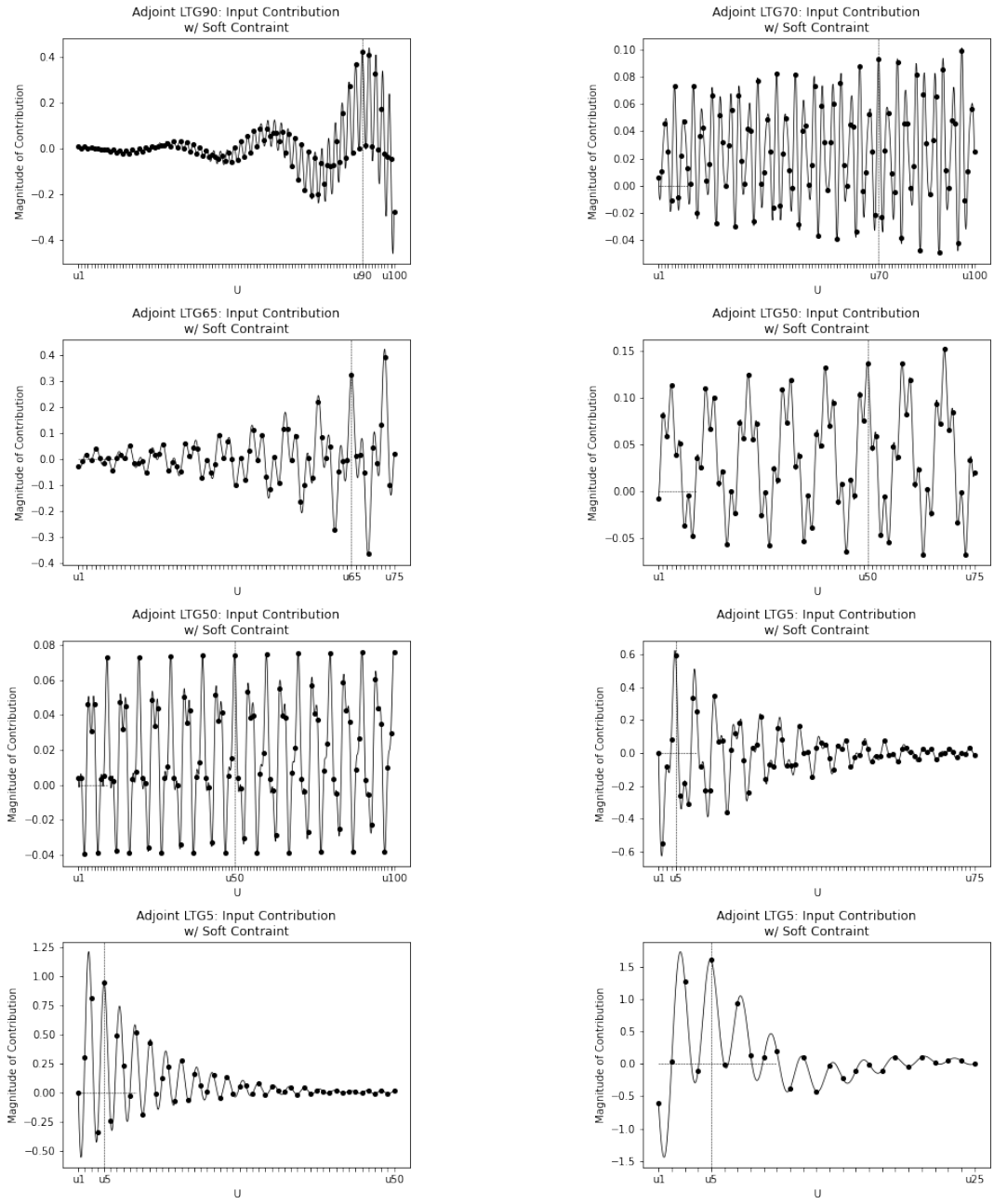


Figure A1: Magnitude of input expression for various LTGi tasks with soft penalty on W_{rec}.

A3.2 Impact of Coadjoint Regularization Weight

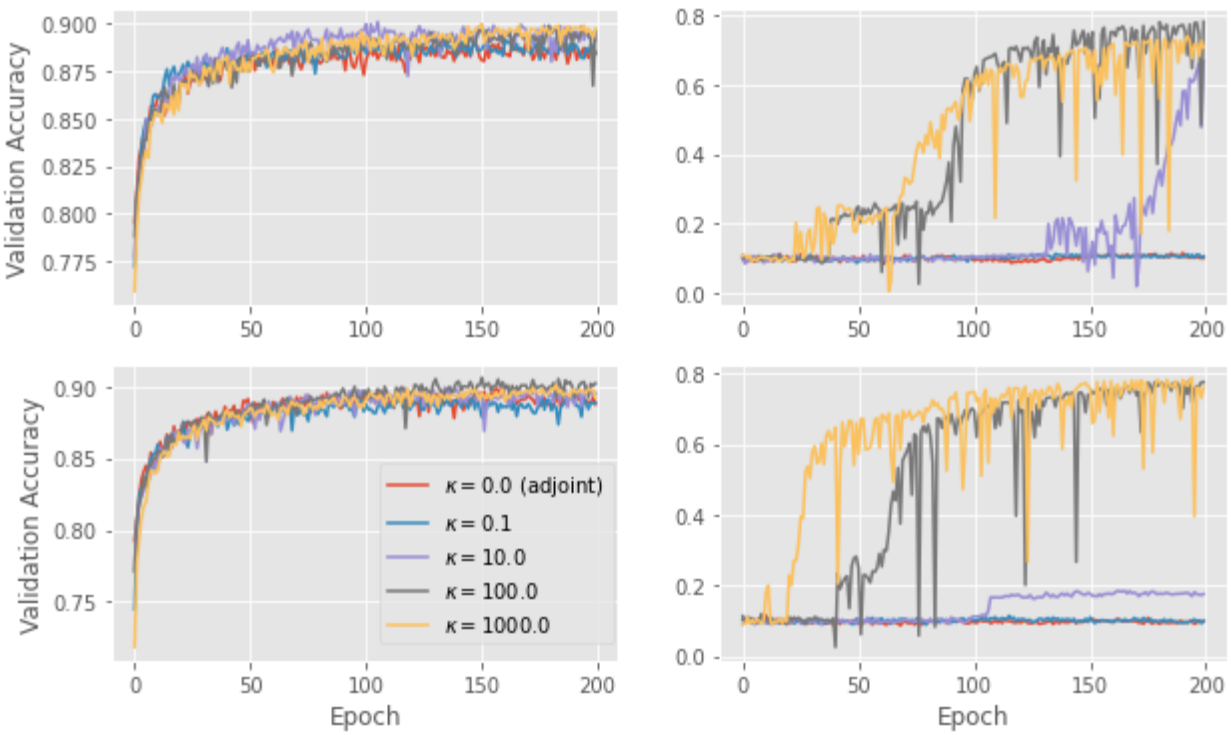


Figure A2: The impact of the coadjoint penalty weight on task accuracy when gradient management is not prohibitive (left column) and when it is (right column) for the Fashion MNIST padded tasks. The left column figures are 0 padding tasks and the right column is the 3x padded experiment. The rows correspond to orthogonal initialization (top row) and Glorot initialization (bottom row).

From Fig. A2 we observe that training with the coadjoint method when gradient flow is not prohibitive is not detrimental to learning as it performs as well as backpropagation across a large variety of weightings. When backpropagation fails to adequately manage gradient flow (right column of Fig. A2) selecting an appropriate penalty weight will induce learning.

In practice, we found that a course hyperparameter search across differing values of $\kappa \in [0, 10^3]$ was sufficient to induce learning. We remark that the results reported throughout all of our experiments could benefit by a finer selection of penalty weights. Furthermore, we acknowledge that the method as a whole would benefit from a procedure that automatically selects an appropriate weighting based on the data and time scale of the problem; we leave this for future work.

A4 Revisiting VEG Model Summaries & Additional Tables

CIFAR10

	sequential				sequential × post				sequential × uniform				permuted				permuted × post				permuted × uniform			
	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$
antisymmetric	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
exponential	1420	320	0	0	1481	345	71	0	852	452	142	0	990	430	54	196	1536	307	100	0	840	102	266	197
gru	81	0	0	0	416	870	14	0	68	3	0	0	54	0	0	0	419	691	31	0	43	1	0	0
lipschitz	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
lstm	68	1	0	0	865	911	16	0	45	50	1	0	85	18	0	0	815	856	14	0	38	9	2	8
rnn	2978	88	0	0	2654	2023	652	0	2927	798	147	0	2521	103	0	0	2634	2234	606	0	2699	456	248	100
unicornn	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
threshold	0.199	0.281	0.38	0.95	0.101	0.114	0.268	0.474	0.136	0.196	0.294	0.696	0.213	0.274	0.356	0.782	0.102	0.114	0.216	0.423	0.149	0.219	0.271	0.557

Fashion MNIST

	sequential				sequential × post				sequential × uniform				permuted				permuted × post				permuted × uniform			
	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$
antisymmetric	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
exponential	1197	260	0	0	1354	1223	8	0	871	173	221	0	632	753	217	0	1367	998	6	0	34	812	104	280
gru	0	0	0	0	600	874	0	0	102	83	0	0	21	2	0	0	816	703	9	0	66	73	0	0
lipschitz	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
lstm	136	0	0	0	907	823	13	0	195	24	0	0	36	0	0	0	1145	759	20	0	15	13	5	0
rnn	3698	0	0	0	2812	2176	78	0	3848	36	0	0	2587	303	0	0	2203	1716	955	0	2052	542	299	0
unicornn	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
threshold	0.375	0.66	0.786	0.97	0.1	0.11	0.583	0.865	0.105	0.472	0.633	0.906	0.473	0.655	0.785	0.971	0.101	0.143	0.346	0.842	0.264	0.421	0.66	0.906

IMDB

	sequential				sequential × post				sequential × uniform				permuted				permuted × post				permuted × uniform			
	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$
antisymmetric	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
exponential	1334	0	0	0	1288	89	885	0	1304	0	0	0	1564	0	0	0	1323	84	843	0	1273	0	275	0
gru	128	0	0	0	730	580	214	1136	127	4	21	0	455	20	0	0	708	510	182	1195	495	197	209	1146
lipschitz	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	21	0	0	0	0
lstm	542	0	0	0	541	180	203	884	515	17	127	186	461	0	0	0	559	189	213	849	457	92	241	662
rnn	1808	444	43	0	939	933	366	815	1094	910	625	514	2800	83	0	0	899	888	389	778	1106	959	801	320
unicornn	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
threshold	0.84	0.964	0.996	1	0.759	0.978	1	1	0.785	0.964	0.999	1	0.759	0.946	0.994	1	0.767	0.979	1	1	0.746	0.96	1	1

MNIST

	sequential				sequential × post				sequential × uniform				permuted				permuted × post				permuted × uniform			
	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$
antisymmetric	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
exponential	1460	1439	482	3919	1194	1641	318	1600	936	940	0	0	1050	370	29	1	1655	886	7	0	551	341	67	129
gru	161	0	0	0	31	705	201	0	30	1733	0	0	20	0	0	0	51	1490	0	0	48	93	7	0
lipschitz	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
lstm	1149	18	0	0	47	974	270	0	70	1829	2	0	120	0	0	0	111	1982	32	0	77	13	2	0
rnn	4096	0	0	0	3546	1458	200	0	3177	788	0	0	2508	239	0	0	2717	1063	1406	0	2652	648	139	0
unicornn	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
threshold	0.188	0.642	0.851	0.996	0.105	0.112	0.556	0.977	0.106	0.162	0.594	0.976	0.375	0.617	0.85	0.991	0.109	0.135	0.336	0.922	0.179	0.31	0.663	0.965

Reuters

	sequential				sequential × post				sequential × uniform				permuted				permuted × post				permuted × uniform			
	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$
antisymmetric	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
exponential	1418	2	0	0	257	1003	768	0	1132	287	756	303	265	1151	0	0	231	1111	663	12	256	1162	609	323
gru	60	0	0	0	410	261	504	1048	206	1	0	0	64	0	0	0	285	313	545	1081	301	266	501	814
lipschitz	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
lstm	684	1	0	0	2689	444	810	873	1926	75	120	410	131	1	0	0	2573	422	720	919	1438	340	287	919
rnn	2292	615	50	0	2000	621	1338	644	2265	766	997	418	1646	1209	28	0	1854	593	1280	577	1549	976	1049	586
unicornn	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
threshold	0.384	0.628	0.922	0.961	0.386	0.756	0.999	1	0.368	0.623	0.982	1	0.39	0.595	0.903	0.959	0.376	0.751	0.998	1	0.372	0.654	0.981	1

Table A4: Number of training attempts that experience vanished gradients during training and their corresponding training accuracy binned by task quartile.

CIFAR10

	sequential				sequential × post				sequential × uniform				permuted				permuted × post				permuted × uniform			
	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$
antisymmetric	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
exponential	40	13	0	0	39	31	0	0	29	18	0	0	38	2	12	1	43	32	0	0	27	2	20	1
gru	0	0	0	0	14	18	1	0	0	0	0	0	0	0	0	0	12	14	0	0	0	0	0	0
lipschitz	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
lstm	2	0	0	0	30	19	2	0	4	5	0	0	1	0	0	0	34	15	2	0	2	0	0	0
rnn	108	3	0	0	54	60	30	0	90	33	4	0	87	6	2	0	38	83	25	0	76	22	9	3
unicornn	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
threshold	0.216	0.332	0.422	0.701	0.1	0.105	0.304	0.476	0.148	0.218	0.331	0.637	0.236	0.313	0.387	0.489	0.1	0.107	0.242	0.433	0.163	0.251	0.292	0.462

Fashion MNIST

	sequential				sequential × post				sequential × uniform				permuted				permuted × post				permuted × uniform			
	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$
antisymmetric	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
exponential	40	0	0	0	42	20	0	0	27	20	2	0	33	6	13	0	32	18	1	0	19	8	22	1
gru	0	0	0	0	20	16	0	0	0	0	0	0	0	0	0	0	12	21	1	0	1	0	0	0
lipschitz	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
lstm	3	0	0	0	20	32	1	0	8	0	0	0	0	0	0	0	19	34	1	0	0	0	0	0
rnn	113	0	0	0	54	38	2	0	118	3	0	0	80	17	0	0	36	35	23	0	65	20	6	0
unicornn	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
threshold	0.431	0.702	0.815	0.915	0.1	0.115	0.661	0.855	0.119	0.543	0.686	0.884	0.553	0.706	0.803	0.864	0.1	0.167	0.509	0.828	0.305	0.503	0.704	0.837

IMDB

	sequential				sequential × post				sequential × uniform				permuted				permuted × post				permuted × uniform			
	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$
antisymmetric	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
exponential	39	0	0	0	32	39	2	0	40	0	0	0	47	0	0	0	41	23	6	0	35	16	0	0
gru	1	0	0	0	23	59	10	0	0	0	0	0	1	0	0	0	19	53	15	0	53	13	0	0
lipschitz	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0
lstm	12	0	0	0	13	33	7	0	15	9	0	0	10	0	0	0	17	28	7	0	13	29	0	0
rnn	84	0	0	0	74	12	0	0	82	6	0	0	103	0	0	0	69	14	0	0	78	12	0	0
unicornn	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
threshold	0.804	0.842	0.852	0.887	0.5	0.505	0.776	0.835	0.646	0.743	0.775	0.844	0.701	0.826	0.844	0.886	0.501	0.505	0.768	0.846	0.51	0.603	0.71	0.793

MNIST

	sequential				sequential × post				sequential × uniform				permuted				permuted × post				permuted × uniform			
	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$
antisymmetric	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
exponential	40	40	6	114	40	18	9	49	32	28	0	0	34	6	5	0	52	35	0	0	25	3	16	5
gru	0	0	0	0	0	18	3	0	0	48	0	0	0	0	0	0	4	37	0	0	0	0	0	0
lipschitz	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
lstm	34	0	0	0	4	25	7	0	9	44	0	0	4	0	0	0	7	52	1	0	1	0	0	0
rnn	132	0	0	0	102	23	1	0	105	28	0	0	70	12	0	0	77	33	32	0	76	30	0	0
unicornn	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
threshold	0.279	0.721	0.897	0.989	0.105	0.113	0.668	0.982	0.109	0.268	0.69	0.978	0.429	0.706	0.887	0.948	0.11	0.163	0.421	0.923	0.192	0.415	0.749	0.928

Reuters

	sequential				sequential × post				sequential × uniform				permuted				permuted × post				permuted × uniform			
	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$	$Q_{0.25}$	$Q_{0.50}$	$Q_{0.75}$	$Q_{1.00}$
antisymmetric	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
exponential	40	0	0	0	42	18	0	0	26	40	13	0	40	0	0	0	37	22	0	0	3	64	2	0
gru	0	0	0	0	33	21	0	0	7	1	0	0	0	0	0	0	39	16	0	0	42	6	0	0
lipschitz	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
lstm	14	1	0	0	43	47	0	0	27	21	6	0	2	0	0	0	44	40	0	0	38	20	1	0
rnn	80	4	0	0	77	27	0	0	95	23	0	0	66	9	0	0	72	23	1	0	76	25	9	0
unicornn	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
threshold	0.46	0.592	0.696	0.764	0.208	0.361	0.532	0.733	0.342	0.396	0.485	0.654	0.407	0.547	0.635	0.765	0.211	0.361	0.496	0.742	0.275	0.365	0.415	0.606

Table A5: Number of training attempts that incurred vanished gradients at the terminal iterate and their corresponding evaluation accuracy binned by task quartile.

CIFAR10

	sequential ($Q_{0.75} = 0.380$)		sequential × post ($Q_{0.75} = 0.268$)		sequential × uniform ($Q_{0.75} = 0.294$)		permuted ($Q_{0.75} = 0.356$)		permuted × post ($Q_{0.75} = 0.216$)		permuted × uniform ($Q_{0.75} = 0.271$)	
	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished
antisymmetric	1362	0	3601	0	2915	0	374	0	3766	0	558	0
exponential	2172	0	3	0	305	0	4244	196	310	0	1894	197
gru	1807	0	15	0	439	0	1574	0	36	0	657	0
lipschitz	1254	0	2735	0	2413	0	685	0	2195	0	1034	0
lstm	519	0	10	0	271	0	462	0	0	0	323	8
rnn	0	0	0	0	0	0	960	0	0	0	1542	100
unicornn	3998	0	0	0	0	0	2559	0	0	0	0	0

Fashion MNIST

	sequential ($Q_{0.75} = 0.786$)		sequential × post ($Q_{0.75} = 0.583$)		sequential × uniform ($Q_{0.75} = 0.633$)		permuted ($Q_{0.75} = 0.785$)		permuted × post ($Q_{0.75} = 0.346$)		permuted × uniform ($Q_{0.75} = 0.660$)	
	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished
antisymmetric	946	0	3266	0	2744	0	375	0	2921	0	472	0
exponential	3502	0	1213	0	1980	0	3807	0	1738	0	2411	280
gru	1754	0	37	0	403	0	1801	0	110	0	689	0
lipschitz	1335	0	1931	0	1053	0	1204	0	1411	0	649	0
lstm	756	0	4	0	15	0	1632	0	3	0	536	0
rnn	162	0	0	0	263	0	937	0	175	0	1401	0
unicornn	2974	0	0	0	0	0	1506	0	0	0	0	0

IMDB

	sequential ($Q_{0.75} = 0.996$)		sequential × post ($Q_{0.75} = 1.00$)		sequential × uniform ($Q_{0.75} = 0.999$)		permuted ($Q_{0.75} = 0.994$)		permuted × post ($Q_{0.75} = 1.00$)		permuted × uniform ($Q_{0.75} = 0.999$)	
	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished
antisymmetric	1831	0	2171	0	1845	0	2074	0	2023	0	1603	0
exponential	2301	0	1620	0	2114	0	2504	0	1719	0	1781	0
gru	1771	0	363	1136	714	0	1527	0	349	1195	314	1146
lipschitz	1642	0	1223	0	1727	0	1575	0	1193	21	949	0
lstm	1381	0	235	884	358	186	1165	0	314	849	458	662
rnn	424	0	1175	815	589	514	386	0	1006	778	785	320
unicornn	2510	0	0	0	19	0	2760	0	0	0	0	0

MNIST

	sequential ($Q_{0.75} = 0.851$)		sequential × post ($Q_{0.75} = 0.556$)		sequential × uniform ($Q_{0.75} = 0.594$)		permuted ($Q_{0.75} = 0.851$)		permuted × post ($Q_{0.75} = 0.336$)		permuted × uniform ($Q_{0.75} = 0.663$)	
	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished
antisymmetric	1019	0	2743	0	2371	0	542	0	2692	0	251	0
exponential	0	3919	0	1600	2360	0	4167	1	1824	0	2612	129
gru	1658	0	40	0	44	0	1448	0	101	0	767	0
lipschitz	1850	0	2180	0	1795	0	1184	0	1840	0	381	0
lstm	213	0	0	0	0	0	1664	0	6	0	596	0
rnn	210	0	0	0	99	0	1562	0	69	0	1870	0
unicornn	2386	0	0	0	0	0	611	0	0	0	0	0

Reuters

	sequential ($Q_{0.75} = 0.922$)		sequential × post ($Q_{0.75} = 0.999$)		sequential × uniform ($Q_{0.75} = 0.982$)		permuted ($Q_{0.75} = 0.903$)		permuted × post ($Q_{0.75} = 0.998$)		permuted × uniform ($Q_{0.75} = 0.981$)	
	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished
antisymmetric	1771	0	1307	0	1337	0	1718	0	1278	0	1166	0
exponential	2768	0	2625	0	2293	303	2810	0	2825	12	2549	323
gru	1028	0	830	1048	1121	0	1006	0	461	1081	324	814
lipschitz	1905	0	830	0	1959	0	1836	0	806	0	1289	0
lstm	1611	0	1357	873	1376	410	1242	0	1351	919	1127	919
rnn	137	0	407	644	236	418	17	0	350	577	301	586
unicornn	2505	0	1388	0	1746	0	2949	0	1425	0	1634	0

Table A6: Count of training attempts by adjoint regime that attain training accuracy in the top quartile (i.e., greater than $Q_{0.75}$) on a per-task basis for each architecture.

CIFAR10

	sequential ($Q_{0.75} = 0.382$)		sequential × post ($Q_{0.75} = 0.277$)		sequential × uniform ($Q_{0.75} = 0.300$)		permuted ($Q_{0.75} = 0.360$)		permuted × post ($Q_{0.75} = 0.221$)		permuted × uniform ($Q_{0.75} = 0.277$)	
	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished
antisymmetric	36	0	93	0	83	0	12	0	110	0	19	0
exponential	46	0	0	0	1	0	97	1	10	0	58	1
gru	48	0	1	0	20	0	45	0	1	0	15	0
lipschitz	40	0	84	0	70	0	28	0	56	0	23	0
lstm	24	0	0	0	10	0	9	0	0	0	10	0
rnn	0	0	0	0	0	0	33	0	0	0	50	3
unicornn	110	0	0	0	0	0	76	0	0	0	0	0

Fashion MNIST

	sequential ($Q_{0.75} = 0.780$)		sequential × post ($Q_{0.75} = 0.589$)		sequential × uniform ($Q_{0.75} = 0.631$)		permuted ($Q_{0.75} = 0.779$)		permuted × post ($Q_{0.75} = 0.349$)		permuted × uniform ($Q_{0.75} = 0.660$)	
	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished
antisymmetric	18	0	80	0	63	0	8	0	65	0	15	0
exponential	95	0	45	0	58	0	102	0	54	0	59	1
gru	53	0	3	0	14	0	56	0	3	0	22	0
lipschitz	41	0	53	0	38	0	44	0	48	0	29	0
lstm	31	0	1	0	2	0	54	0	0	0	19	0
rnn	1	0	0	0	7	0	20	0	7	0	35	0
unicornn	71	0	0	0	0	0	21	0	0	0	0	0

IMDB

	sequential ($Q_{0.75} = 0.854$)		sequential × post ($Q_{0.75} = 0.773$)		sequential × uniform ($Q_{0.75} = 0.772$)		permuted ($Q_{0.75} = 0.844$)		permuted × post ($Q_{0.75} = 0.766$)		permuted × uniform ($Q_{0.75} = 0.704$)	
	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished
antisymmetric	61	0	101	0	42	0	118	0	116	0	149	0
exponential	105	0	15	0	90	0	101	0	8	0	27	0
gru	68	0	0	0	29	0	42	0	0	0	8	0
lipschitz	19	0	103	0	2	0	2	0	93	0	1	0
lstm	38	0	0	0	25	0	33	0	0	0	27	0
rnn	33	0	0	0	34	0	40	0	0	0	7	0
unicornn	0	0	0	0	0	0	0	0	0	0	0	0

MNIST

	sequential ($Q_{0.75} = 0.857$)		sequential × post ($Q_{0.75} = 0.573$)		sequential × uniform ($Q_{0.75} = 0.605$)		permuted ($Q_{0.75} = 0.856$)		permuted × post ($Q_{0.75} = 0.342$)		permuted × uniform ($Q_{0.75} = 0.676$)	
	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished
antisymmetric	29	0	75	0	53	0	14	0	70	0	6	0
exponential	0	114	0	49	84	0	101	0	58	0	52	5
gru	67	0	2	0	1	0	41	0	3	0	23	0
lipschitz	40	0	57	0	46	0	43	0	48	0	27	0
lstm	15	0	0	0	0	0	49	0	0	0	24	0
rnn	5	0	0	0	3	0	46	0	3	0	49	0
unicornn	37	0	0	0	0	0	7	0	0	0	0	0

Reuters

	sequential ($Q_{0.75} = 0.682$)		sequential × post ($Q_{0.75} = 0.513$)		sequential × uniform ($Q_{0.75} = 0.471$)		permuted ($Q_{0.75} = 0.624$)		permuted × post ($Q_{0.75} = 0.479$)		permuted × uniform ($Q_{0.75} = 0.410$)	
	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished	stable	vanished
antisymmetric	94	0	76	0	136	0	133	0	97	0	139	0
exponential	3	0	0	0	45	0	2	0	10	0	36	0
gru	12	0	11	0	2	0	39	0	5	0	3	0
lipschitz	88	0	123	0	69	0	93	0	124	0	105	0
lstm	45	0	0	0	22	0	38	0	0	0	18	0
rnn	0	0	0	0	7	0	0	0	0	0	2	0
unicornn	77	0	101	0	29	0	9	0	72	0	1	0

Table A7: Count of training attempts in each adjoint regime that attain evaluation accuracy in the top quartile (i.e., greater than $Q_{0.75}$) on a per-task basis for each architecture.

A5 Linear Model Coefficient Tables

	Estimate	Std. Error	t value
(Intercept)	0.554629	0.001696	327.1
adjoint metric	0.000808	0.000012	65.5

Table A8: MODEL 1 coefficient estimates and standard errors.

	Estimate	Std. Error	t value
(Intercept)	-0.389208	0.010304	-37.8
adjoint metric	0.000659	0.000011	61.5
exponential	0.062467	0.014306	4.4
gru	-0.010889	0.024022	-0.5
lipschitz	-0.253604	0.025034	-10.1
lstm	-0.039498	0.018285	-2.2
rnn	0.109528	0.015192	7.2
unicornn	0.137791	0.017465	7.9
LR	-0.206917	0.003107	-66.6
LR^2	-0.014326	0.000221	-65.0
dim	0.000033	0.000014	2.4
dataset = fashion mnist	0.224876	0.002875	78.2
dataset = imdb	0.637598	0.002786	228.9
dataset = mnist	0.203151	0.002867	70.9
dataset = reuters	0.424152	0.002756	153.9
post noise	-0.073278	0.004769	-15.4
uniform noise	-0.089835	0.004774	-18.8
order = permute	-0.004596	0.001731	-2.7
epochs	0.003895	0.000007	596.4
exponential * LR	0.011361	0.004565	2.5
gru * LR	-0.014677	0.006689	-2.2
lipschitz * LR	-0.051046	0.006972	-7.3
lstm * LR	0.005386	0.005488	1.0
rnn * LR	0.116526	0.004693	24.8
unicornn * LR	-0.005282	0.005683	-0.9
exponential * LR^2	0.001789	0.000324	5.5
gru * LR^2	-0.001472	0.000434	-3.4
lipschitz * LR^2	-0.002115	0.000447	-4.7
lstm * LR^2	0.000192	0.000375	0.5
rnn * LR^2	0.009572	0.000326	29.3
unicornn * LR^2	-0.001558	0.000403	-3.9
exponential * post noise	-0.091826	0.006999	-13.1
gru * post noise	-0.102433	0.007906	-13.0
lipschitz * post noise	0.029330	0.007329	4.0
lstm * post noise	-0.012013	0.007712	-1.6
rnn * post noise	0.005539	0.006900	0.8
unicornn * post noise	0.014250	0.010705	1.3
exponential * uniform noise	0.005152	0.006985	0.7
gru * uniform noise	-0.007158	0.007991	-0.9
lipschitz * uniform noise	0.037987	0.007340	5.2
lstm * uniform noise	0.059214	0.007710	7.7
rnn * uniform noise	0.074150	0.006888	10.8
unicornn * uniform noise	0.012388	0.010718	1.2

Table A9: MODEL 2 coefficient estimates and standard errors.

	Estimate	Std. Error	t value
(Intercept)	-0.388855	0.010404	-37.4
exponential	0.003710	0.014413	0.3
gru	0.010079	0.024250	0.4
lipschitz	-0.246101	0.025275	-9.7
lstm	-0.060495	0.018459	-3.3
rnn	0.067864	0.015324	4.4
unicornn	0.139739	0.017635	7.9
LR	-0.206913	0.003137	-66.0
LR^2	-0.014336	0.000223	-64.4
dim	0.000032	0.000014	2.3
dataset = fashion mnist	0.225447	0.002902	77.7
dataset = imdb	0.639595	0.002812	227.4
dataset = mnist	0.201172	0.002895	69.5
dataset = reuters	0.424946	0.002783	152.7
post noise	-0.072876	0.004815	-15.1
uniform noise	-0.089473	0.004820	-18.6
order = permute	-0.003844	0.001748	-2.2
epochs	0.003901	0.000007	596.4
exponential * LR	0.000569	0.004606	0.1
gru * LR	-0.009402	0.006753	-1.4
lipschitz * LR	-0.049051	0.007039	-7.0
lstm * LR	0.001335	0.005540	0.2
rnn * LR	0.113617	0.004738	24.0
unicornn * LR	-0.004800	0.005738	-0.8
exponential * LR^2	0.001291	0.000327	3.9
gru * LR^2	-0.001211	0.000438	-2.8
lipschitz * LR^2	-0.001983	0.000451	-4.4
lstm * LR^2	-0.000040	0.000379	-0.1
rnn * LR^2	0.009614	0.000330	29.2
unicornn * LR^2	-0.001517	0.000407	-3.7
exponential * post noise	-0.098075	0.007066	-13.9
gru * post noise	-0.126337	0.007972	-15.8
lipschitz * post noise	0.029498	0.007400	4.0
lstm * post noise	-0.036978	0.007776	-4.8
rnn * post noise	-0.005566	0.006964	-0.8
unicornn * post noise	0.013428	0.010809	1.2
exponential * uniform noise	0.004242	0.007052	0.6
gru * uniform noise	-0.016471	0.008067	-2.0
lipschitz * uniform noise	0.037555	0.007411	5.1
lstm * uniform noise	0.047581	0.007782	6.1
rnn * uniform noise	0.068904	0.006955	9.9
unicornn * uniform noise	0.011509	0.010822	1.1

Table A10: MODEL 3 coefficient estimates and standard errors.

	Estimate	Std. Error	t value
(Intercept)	0.561104	0.001519	369.5
adjoint metric	0.005536	0.000055	100.4

Table A11: MODEL 4 coefficient estimates and standard errors.

	Estimate	Std. Error	t value
(Intercept)	-0.166764	0.009667	-17.3
adjoint metric	0.003031	0.000071	42.8
exponential	0.385485	0.014315	26.9
gru	0.121142	0.024798	4.9
lipschitz	-0.402029	0.025817	-15.6
lstm	0.297915	0.017426	17.1
rnn	0.368652	0.014360	25.7
unicornn	0.337191	0.015792	21.4
LR	-0.167420	0.002809	-59.6
LR^2	-0.010889	0.000199	-54.6
dim	-0.000007	0.000013	-0.6
dataset = fashion mnist	0.244648	0.002686	91.1
dataset = imdb	0.440448	0.002615	168.4
dataset = mnist	0.244248	0.002685	91.0
dataset = reuters	0.196438	0.002582	76.1
post noise	-0.099586	0.004312	-23.1
uniform noise	-0.119898	0.004316	-27.8
order = permute	-0.020505	0.001611	-12.7
epochs	0.000401	0.000065	6.2
exponential * LR	0.107020	0.004260	25.1
gru * LR	0.026398	0.006609	4.0
lipschitz * LR	-0.087220	0.006859	-12.7
lstm * LR	0.098358	0.005110	19.2
rnn * LR	0.162722	0.004280	38.0
unicornn * LR	0.075794	0.005138	14.8
exponential * LR^2	0.007822	0.000297	26.4
gru * LR^2	0.001342	0.000419	3.2
lipschitz * LR^2	-0.004449	0.000428	-10.4
lstm * LR^2	0.006217	0.000347	17.9
rnn * LR^2	0.011629	0.000299	38.8
unicornn * LR^2	0.004100	0.000365	11.2
exponential * post noise	-0.147453	0.006372	-23.1
gru * post noise	-0.135198	0.008219	-16.4
lipschitz * post noise	0.027762	0.006914	4.0
lstm * post noise	-0.107871	0.007655	-14.1
rnn * post noise	-0.068440	0.006367	-10.7
unicornn * post noise	-0.006035	0.009691	-0.6
exponential * uniform noise	-0.016781	0.006322	-2.7
gru * uniform noise	-0.033231	0.007668	-4.3
lipschitz * uniform noise	-0.010733	0.006919	-1.6
lstm * uniform noise	0.012609	0.007303	1.7
rnn * uniform noise	0.047723	0.006302	7.6
unicornn * uniform noise	-0.115179	0.009703	-11.9

Table A12: MODEL 5 coefficient estimates and standard errors.

	Estimate	Std. Error	t value
(Intercept)	-0.170097	0.009962	-17.1
exponential	0.122263	0.013325	9.2
gru	0.185399	0.025510	7.3
lipschitz	-0.388064	0.026605	-14.6
lstm	0.192344	0.017779	10.8
rnn	0.212007	0.014312	14.8
unicornn	0.350552	0.016272	21.5
LR	-0.168041	0.002895	-58.0
LR^2	-0.010968	0.000205	-53.4
dim	-0.000016	0.000013	-1.2
dataset = fashion mnist	0.248049	0.002767	89.7
dataset = imdb	0.450829	0.002684	168.0
dataset = mnist	0.237333	0.002762	85.9
dataset = reuters	0.204586	0.002654	77.1
post noise	-0.097962	0.004444	-22.0
uniform noise	-0.118527	0.004448	-26.6
order = permute	-0.016621	0.001657	-10.0
epochs	0.000443	0.000067	6.6
exponential * LR	0.061894	0.004254	14.6
gru * LR	0.043679	0.006799	6.4
lipschitz * LR	-0.081938	0.007068	-11.6
lstm * LR	0.081873	0.005252	15.6
rnn * LR	0.167283	0.004410	37.9
unicornn * LR	0.079121	0.005295	14.9
exponential * LR^2	0.005895	0.000302	19.5
gru * LR^2	0.002277	0.000432	5.3
lipschitz * LR^2	-0.004016	0.000441	-9.1
lstm * LR^2	0.005503	0.000357	15.4
rnn * LR^2	0.013172	0.000306	43.0
unicornn * LR^2	0.004353	0.000376	11.6
exponential * post noise	-0.177436	0.006527	-27.2
gru * post noise	-0.257643	0.007942	-32.4
lipschitz * post noise	0.028186	0.007126	4.0
lstm * post noise	-0.222673	0.007391	-30.1
rnn * post noise	-0.111168	0.006481	-17.2
unicornn * post noise	-0.011722	0.009986	-1.2
exponential * uniform noise	-0.022935	0.006514	-3.5
gru * uniform noise	-0.081524	0.007816	-10.4
lipschitz * uniform noise	-0.012487	0.007131	-1.8
lstm * uniform noise	-0.039824	0.007420	-5.4
rnn * uniform noise	0.025873	0.006474	4.0
unicornn * uniform noise	-0.121506	0.009999	-12.2

Table A13: MODEL 6 coefficient estimates and standard errors.

	Estimate	Std. Error	t value
(Intercept)	-16.773395	0.129014	-130.0
accuracy	5.028647	0.075263	66.8

Table A14: MODEL 7 coefficient estimates and standard errors.

	Estimate	Std. Error	t value
(Intercept)	2.220014	0.691035	3.2
accuracy	4.798466	0.075041	63.9
exponential	-89.256280	0.956861	-93.3
gru	31.609801	1.619733	19.5
lipschitz	12.800656	1.683300	7.6
lstm	-31.744651	1.227192	-25.9
rnn	-63.828688	1.018045	-62.7
unicornn	2.291405	1.170710	2.0
LR	0.998814	0.208819	4.8
LR^2	0.053961	0.014819	3.6
dim	-0.002009	0.000927	-2.2
dataset = fashion mnist	-0.218300	0.193713	-1.1
dataset = imdb	-0.033683	0.193049	-0.2
dataset = mnist	-3.966318	0.193025	-20.5
dataset = reuters	-0.844784	0.187777	-4.5
post noise	0.958344	0.319691	3.0
uniform noise	0.976469	0.320040	3.1
order = permute	1.157570	0.116175	10.0
exponential * LR	-16.395001	0.305769	-53.6
gru * LR	7.984143	0.450203	17.7
lipschitz * LR	3.315589	0.468174	7.1
lstm * LR	-6.187261	0.368135	-16.8
rnn * LR	-5.025122	0.314820	-16.0
unicornn * LR	0.757340	0.380932	2.0
exponential *LR^2	-0.763514	0.021720	-35.2
gru * LR^2	0.397572	0.029195	13.6
lipschitz * LR^2	0.211058	0.030005	7.0
lstm * LR^2	-0.352016	0.025169	-14.0
rnn * LR^2	0.015125	0.021904	0.7
unicornn * LR^2	0.069483	0.027045	2.6
exponential * post noise	-9.000246	0.469182	-19.2
gru * post noise	-35.702164	0.530637	-67.3
lipschitz * post noise	0.145393	0.492080	0.3
lstm * post noise	-37.783471	0.516707	-73.1
rnn * post noise	-16.867696	0.462405	-36.5
unicornn * post noise	-1.279895	0.717710	-1.8
exponential * uniform noise	-1.385131	0.468176	-3.0
gru * uniform noise	-14.100072	0.536844	-26.3
lipschitz * uniform noise	-0.808041	0.492722	-1.6
lstm * uniform noise	-17.923256	0.517185	-34.7
rnn * uniform noise	-8.318314	0.461786	-18.0
unicornn * uniform noise	-1.357448	0.718598	-1.9

Table A15: MODEL 8 coefficient estimates and standard errors.

	Estimate	Std. Error	t value
(Intercept)	0.714736	0.700824	1.0
exponential	-89.236565	0.970976	-91.9
gru	31.434463	1.643113	19.1
lipschitz	11.279579	1.707690	6.6
lstm	-32.089892	1.245194	-25.8
rnn	-63.520565	1.033018	-61.5
unicornn	2.967955	1.187938	2.5
LR	0.005746	0.211314	0.0
LR^2	-0.014840	0.014998	-1.0
dim	-0.001980	0.000940	-2.1
dataset = fashion mnist	0.869623	0.195798	4.4
dataset = imdb	3.043875	0.189699	16.0
dataset = mnist	-2.995671	0.195253	-15.3
dataset = reuters	1.222724	0.187689	6.5
post noise	0.611903	0.324363	1.9
uniform noise	0.550985	0.324694	1.7
order = permute	1.138084	0.117882	9.7
exponential * LR	-16.392288	0.310280	-52.8
gru * LR	7.902100	0.456745	17.3
lipschitz * LR	3.008173	0.475012	6.3
lstm * LR	-6.187125	0.373549	-16.6
rnn * LR	-4.481409	0.319342	-14.0
unicornn * LR	0.734813	0.386553	1.9
exponential *LR^2	-0.757299	0.022040	-34.4
gru * LR^2	0.390378	0.029621	13.2
lipschitz * LR^2	0.197920	0.030445	6.5
lstm * LR^2	-0.352263	0.025539	-13.8
rnn * LR^2	0.061277	0.022215	2.8
unicornn * LR^2	0.062237	0.027444	2.3
exponential * post noise	-9.498218	0.476039	-20.0
gru * post noise	-36.359196	0.538295	-67.5
lipschitz * post noise	0.260421	0.499296	0.5
lstm * post noise	-37.985775	0.524295	-72.5
rnn * post noise	-16.893334	0.469225	-36.0
unicornn * post noise	-1.277259	0.728291	-1.8
exponential * uniform noise	-1.391343	0.475084	-2.9
gru * uniform noise	-14.216689	0.544694	-26.1
lipschitz * uniform noise	-0.655344	0.499948	-1.3
lstm * uniform noise	-17.725272	0.524779	-33.8
rnn * uniform noise	-7.987948	0.468567	-17.0
unicornn * uniform noise	-1.365918	0.729192	-1.9

Table A16: MODEL 9 coefficient estimates and standard errors.

	Estimate	Std. Error	t value
(Intercept)	-36.654309	0.249102	-147.1
eval. accuracy	45.936038	0.457481	100.4

Table A17: MODEL 10 coefficient estimates and standard errors.

	Estimate	Std. Error	t value
(Intercept)	2.184928	0.775342	2.8
eval. accuracy	19.308812	0.450724	42.8
exponential	-89.204996	1.033448	-86.3
gru	17.620471	1.977424	8.9
lipschitz	12.100298	2.067829	5.9
lstm	-38.544742	1.379628	-27.9
rnn	-55.775004	1.112488	-50.1
unicornn	-2.360664	1.270095	-1.9
LR	3.039840	0.236643	12.8
LR^2	0.185911	0.016663	11.2
dim	-0.002536	0.001028	-2.5
dataset = fashion mnist	-3.667266	0.241688	-15.2
dataset = imdb	-5.279935	0.290667	-18.2
dataset = mnist	-6.863920	0.239188	-28.7
dataset = reuters	-1.262113	0.225275	-5.6
post noise	2.427226	0.346959	7.0
uniform noise	2.740764	0.348611	7.9
order = permute	1.602190	0.128584	12.5
epochs	0.005276	0.005179	1.0
exponential * LR	-16.083421	0.330609	-48.6
gru * LR	4.857792	0.526898	9.2
lipschitz * LR	3.324929	0.548624	6.1
lstm * LR	-7.019596	0.408398	-17.2
rnn * LR	-1.725084	0.349733	-4.9
unicornn * LR	-0.430121	0.411620	-1.0
exponential * LR^2	-0.749718	0.023544	-31.8
gru * LR^2	0.264431	0.033436	7.9
lipschitz * LR^2	0.220148	0.034196	6.4
lstm * LR^2	-0.341875	0.027779	-12.3
rnn * LR^2	0.254818	0.024455	10.4
unicornn * LR^2	-0.000429	0.029180	-0.0
exponential * post noise	-6.466182	0.511777	-12.6
gru * post noise	-35.423014	0.625911	-56.6
lipschitz * post noise	-0.404399	0.552002	-0.7
lstm * post noise	-33.576700	0.581098	-57.8
rnn * post noise	-11.950707	0.504406	-23.7
unicornn * post noise	-1.650018	0.773405	-2.1
exponential * uniform noise	-1.587609	0.504574	-3.1
gru * uniform noise	-14.359034	0.606456	-23.7
lipschitz * uniform noise	-0.337459	0.552262	-0.6
lstm * uniform noise	-16.529956	0.574945	-28.8
rnn * uniform noise	-7.708681	0.501515	-15.4
unicornn * uniform noise	0.258714	0.776298	0.3

Table A18: MODEL 11 coefficient estimates and standard errors.

	Estimate	Std. Error	t value
(Intercept)	-1.099439	0.795148	-1.4
exponential	-86.844250	1.063551	-81.7
gru	21.200315	2.036103	10.4
lipschitz	4.607237	2.123458	2.2
lstm	-34.830804	1.419028	-24.5
rnn	-51.681408	1.142288	-45.2
unicornn	4.408076	1.298786	3.4
LR	-0.204835	0.231054	-0.9
LR^2	-0.025860	0.016399	-1.6
dim	-0.002845	0.001060	-2.7
dataset = fashion mnist	1.122275	0.220829	5.1
dataset = imdb	3.425045	0.214198	16.0
dataset = mnist	-2.281292	0.220479	-10.3
dataset = reuters	2.688192	0.211826	12.7
post noise	0.535692	0.354667	1.5
uniform noise	0.452144	0.355034	1.3
order = permute	1.281249	0.132293	9.7
epochs	0.013827	0.005333	2.6
exponential * LR	-14.888331	0.339509	-43.9
gru * LR	5.701172	0.542640	10.5
lipschitz * LR	1.742807	0.564127	3.1
lstm * LR	-5.438720	0.419172	-13.0
rnn * LR	1.504960	0.351958	4.3
unicornn * LR	1.097613	0.422619	2.6
exponential * LR^2	-0.635894	0.024109	-26.4
gru * LR^2	0.308391	0.034443	9.0
lipschitz * LR^2	0.142598	0.035192	4.1
lstm * LR^2	-0.235627	0.028514	-8.3
rnn * LR^2	0.509150	0.024449	20.8
unicornn * LR^2	0.083628	0.030004	2.8
exponential * post noise	-9.892254	0.520955	-19.0
gru * post noise	-40.397796	0.633861	-63.7
lipschitz * post noise	0.139831	0.568740	0.2
lstm * post noise	-37.876246	0.589877	-64.2
rnn * post noise	-14.097230	0.517267	-27.3
unicornn * post noise	-1.876359	0.797048	-2.4
exponential * uniform noise	-2.030459	0.519902	-3.9
gru * uniform noise	-15.933172	0.623863	-25.5
lipschitz * uniform noise	-0.578563	0.569129	-1.0
lstm * uniform noise	-17.298907	0.592247	-29.2
rnn * uniform noise	-7.209108	0.516720	-14.0
unicornn * uniform noise	-2.087427	0.798056	-2.6

Table A19: MODEL 12 coefficient estimates and standard errors.