




2020

Orthogonal Recurrent Neural Networks and Batch Normalization in Deep Neural Networks

Kyle Eric Helfrich

University of Kentucky, kylehelfrich@hotmail.com

Author ORCID Identifier:

 <https://orcid.org/0000-0001-6590-1334>

Digital Object Identifier: <https://doi.org/10.13023/etd.2020.223>

[Right click to open a feedback form in a new tab to let us know how this document benefits you.](#)

Recommended Citation

Helfrich, Kyle Eric, "Orthogonal Recurrent Neural Networks and Batch Normalization in Deep Neural Networks" (2020). *Theses and Dissertations--Mathematics*. 70.

https://uknowledge.uky.edu/math_etds/70

This Doctoral Dissertation is brought to you for free and open access by the Mathematics at UKnowledge. It has been accepted for inclusion in Theses and Dissertations--Mathematics by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Kyle Eric Helfrich, Student

Dr. Qiang Ye, Major Professor

Dr. Peter Hislop, Director of Graduate Studies

Orthogonal Recurrent Neural Networks and Batch Normalization in Deep Neural
Networks

DISSERTATION

A dissertation submitted in partial
fulfillment of the requirements for
the degree of Doctor of Philosophy
in the College of Arts and Sciences
at the University of Kentucky

By
Kyle E. Helfrich
Lexington, Kentucky

Director: Dr. Qiang Ye, Professor of Mathematics
Lexington, Kentucky

2020

Copyright© Kyle E. Helfrich 2020
<https://orcid.org/0000-0001-6590-1334>

ABSTRACT OF DISSERTATION

Orthogonal Recurrent Neural Networks and Batch Normalization in Deep Neural Networks

Despite the recent success of various machine learning techniques, there are still numerous obstacles that must be overcome. One obstacle is known as the vanishing/exploding gradient problem. This problem refers to gradients that either become zero or unbounded. This is a well known problem that commonly occurs in Recurrent Neural Networks (RNNs). In this work we describe how this problem can be mitigated, establish three different architectures that are designed to avoid this issue, and derive update schemes for each architecture. Another portion of this work focuses on the often used technique of batch normalization. Although found to be successful in decreasing training times and in preventing overfitting, it is still unknown why this technique works. In this paper we describe batch normalization and provide a potential alternative with the end goal of improving our understanding of how batch normalization works.

KEYWORDS: Machine Learning, Recurrent Neural Networks, Vanishing Gradients, Exploding Gradients, Batch Normalization, Neural Networks

Kyle E. Helfrich

May 13, 2020

Orthogonal Recurrent Neural Networks and Batch Normalization in Deep Neural
Networks

By
Kyle E. Helfrich

Qiang Ye

Director of Dissertation

Peter Hislop

Director of Graduate Studies

May 13, 2020

This is dedicated to my family. Without your patience and belief in me, this wouldn't be possible.

ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor, Dr. Qiang Ye. His patience and willingness to support me throughout this journey can never be replaced. I also want to thank him for teaching the numerical analysis courses which are by far some of my favorite courses here at the University of Kentucky. I want to thank Devin Willmott for his willingness to discuss any manner of questions pertaining to machine learning and programming. These discussions will be missed. I also want to thank Kehelwala Dewage Maduranga for his flexibility in sharing the computational resources we have and explaining how to submit jobs to the HPC. I would like to thank Dr. Nathan Jacobs for teaching an awesome course in machine learning. His knowledge in the field is top notch. I want to thank Dr. Lawrence Harris for conveying the importance of not taking anything for granted when writing proofs and Dr. Francis Chung for being the best instructor that I've ever had. Finally I want to thank all of my classmates and friends that supported me throughout my journey.

TABLE OF CONTENTS

Acknowledgments	iii
List of Tables	v
List of Figures	vi
Chapter 1 Introduction to Machine Learning	1
1.1 A Brief Overview of Machine Learning	1
1.2 Supervised Learning	2
1.3 Feedforward Networks	2
1.4 Nonlinear Functions	5
1.5 Loss Functions	8
1.6 Gradients	11
1.7 Optimizers	15
Chapter 2 Recurrent Neural Networks	19
2.1 Back-Propagation Through Time	20
2.2 Vanishing/Exploding Gradients	21
2.3 Gated Recurrent Neural Networks	22
2.4 Orthogonal/Unitary Recurrent Neural Networks	23
2.5 Scaled Cayley Orthogonal Recurrent Neural Network	33
2.6 Scaled Cayley Unitary Recurrent Neural Network	47
2.7 Eigenvalue Normalized Recurrent Neural Network	49
Chapter 3 Batch Normalization	64
3.1 Related Work	66
3.2 Issues with Batch Normalization	66
3.3 Batch Normalized Preconditioning	67
3.4 Experiments	71
Bibliography	74
Vita	79

LIST OF TABLES

2.1	Results for unpermuted and permuted pixel-by-pixel MNIST experiments. Evaluation accuracies are based on the best test accuracy at the end of every epoch. Asterisks indicate reported results from [23] and [35].	41
2.2	Results for the TIMIT speech dataset based on the best validation MSE.	44
2.3	Timing results for the unpermuted MNIST dataset.	47
2.4	TIMIT: Best validation MSE after 300 epochs with test MSE and perceptual metrics. N - dimension of h (for ENRNN, dimensions of $h^{(L)}/h^{(S)}$) that match $\approx 200k$ trainable parameters.	59
2.5	Character PTB: Best testing MSE in BPC after 20 epochs. N - dimension of h (for ENRNN, dimensions of $h^{(L)}/h^{(S)}$). Entries marked by an asterix (*), (**), and (***) are reported from [21], [35], and [25], resp.	60

LIST OF FIGURES

1.1	This figure shows how a validation data set can be used to select a network that will generalize well. Here the x -axis indicates the capacity of the network and the y -axis is the error score. The blue dashed line shows the performance of the network on the training data set and the green line is the performance of the network on the validation data set which is an indication of how the network will generalize. As can be seen, the user should adjust the size or capacity of the network until achieving the minimal validation or generalization error. If the network is too small, the network will underfit the data and if it is too large it will overfit the data. Image is from [12].	3
1.2	Diagram of a perceptron, P. The perceptron takes an input $x \in \mathbb{R}^n$ and applies an affine transformation $w^T x + b$ followed by a pointwise nonlinearity $\sigma(\cdot)$	4
1.3	Diagram of a single layer FFN. The layer takes the input $x \in \mathbb{R}^n$ and applies an affine transformation $Wx + b$ followed by a pointwise nonlinearity $\sigma(\cdot)$	5
1.4	Diagram of a multilayer feedforward network. Each layer takes the input $x \in \mathbb{R}^n$ and applies an affine transformation $Wx + b$ followed by a pointwise nonlinearity $\sigma(\cdot)$	6
1.5	Plot of the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ (Left) and its derivative $\sigma'(x) = \sigma(x)(1 - \sigma(x))$ (Right).	7
1.6	Plot of the tanh function $\sigma(x) = \tanh(x)$ (Left) and its derivative $\sigma'(x) = 1 - \tanh^2(x)$ (Right).	7
1.7	Plot of the ReLU function $\sigma(x) = \max\{0, x\}$ (Left) and its derivative $\sigma'(x) = 0$ or 1 (Right).	8
1.8	Plot of the Leaky ReLU function (Left) and its derivative (Right) using $\alpha = 0.2$	9
1.9	A visual explanation of SGD. The circles represent the level curves of a convex loss function, \mathcal{L} , and $w \in \mathbb{R}^2$ are the parameters of the network. The initial parameter w_0 is updated using SGD to w_1 and so on. This continues until the network reaches a sufficient minimum.	16
2.1	Diagram of a standard RNN. Here $y^{(t)}$ is the actual label and $o^{(t)}$ is the predicted output by the RNN and L is some loss function. Superscript indicates the time step. Image is from [12].	20
2.2	Diagram of an RNN structure used for classification. Note only one final output $o^{(\tau)}$ at the end of the sequence. Superscript indicates the time step. Image is courtesy of [12].	20
2.3	Plots of the modReLU activation function using real input, x , and $b=-0.5$ (left), $b= 0$ (center), and $b=0.5$ (right).	25

2.4	Surface plots of the modulus of the approximate modReLU activation function σ_ϵ (Left) and the modulus of the gradient of σ_ϵ with respect to \bar{z} (Right). Both plots use a bias of $b=0.5$	27
2.5	An illustration of the Adding Problem. The goal of the machine is to output the sum of the entries marked by one, in this case $0.84+0.22 = 1.06$	39
2.6	Test set MSE for each machine on the adding problem with sequence lengths of $T = 200$ (top), $T = 400$ (middle), and $T = 750$ (bottom). . . .	40
2.7	Example images of handwritten digits from the MNIST data set.	41
2.8	Test accuracy for unpermuted MNIST over time. All scoRNN models and the best performing LSTM, Restricted-Capacity uRNN, and Full-Capacity uRNN are shown.	42
2.9	Test accuracy for permuted MNIST over time. All scoRNN models and the best performing LSTM, Restricted-Capacity uRNN, and Full-Capacity uRNN are shown.	43
2.10	Unitary scores ($\ W^*W - I\ _F$) for the Full-Capacity uRNN recurrent weight matrix and orthogonality scores ($\ W^TW - I\ _F$) for the scoRNN recurrent weight matrix using a GPU on the pixel-by-pixel MNIST experiment. . . .	45
2.11	Gradient norms $\ \frac{\partial L}{\partial h_t}\ $ for scoRNN and LSTM models during training on the adding problem. The x -axis shows different values of t . The left plot shows gradients at the beginning of training, and the right shows gradients after 300 training iterations.	46
2.12	Test set MSE for the adding problem with sequence length of $T = 750$	56
2.13	Cross-entropy for the copying problem with sequence length of $T = 2000$	57
2.14	Validation set MSE for the TIMIT problem	58
2.15	Gradient norms $\ \frac{\partial h_\tau^{(S)}}{\partial x_t}\ $ The x -axis is t from left to right and y -axis is τ from top to bottom. The column at t shows dependence of states $h_\tau^{(S)}/h_\tau^{(L)}$ on x_t	61
2.16	Gradient norms $\ \frac{\partial h_\tau^{(L)}}{\partial x_t}\ $ The x -axis is t from left to right and y -axis is τ from top to bottom. The column at t shows dependence of states $h_\tau^{(S)}/h_\tau^{(L)}$ on x_t	62
2.17	Test set MSE for the ENRNN on the adding problem with sequence length of $T = 750$ with various short-term hidden state sizes $h^{(S)}$	62
2.18	Test set MSE for the ENRNN on the adding problem with sequence length of $T = 750$ with fixed hidden state size of 160 and various short-term and long-term hidden state sizes $h^{(S)}$ and $h^{(L)}$	63
3.1	Training set cross-entropy loss (left) and test accuracy (right) using batch normalization on the CIFAR10 dataset. The network is a simple 3 hidden layer network of size 100 per layer plus one output layer of size 10. The SGD optimizer is used with batch normalization applied before each layer using running batch statistics during inference. Models trained using a batch size of 60 and 6 are denoted BS60 and BS6 respectively. The learning rate was optimized for both batch sizes.	67
3.2	Example images from the CIFAR-10 data set.	72

3.3	Training loss (left) and test loss (right) on the CIFAR-10 dataset.	73
-----	---	----

Chapter 1 Introduction to Machine Learning

1.1 A Brief Overview of Machine Learning

The field of machine learning can trace its origins as far back as the 1940s [12] but its popularity has grown immensely since the early 2000s. This growth has been fueled in part by the development of advanced computational hardware such as graphics processing units (GPUs), increased availability of large data sets, and machine learning's strong track record of solving ever increasingly complex tasks [12]. The field of machine learning is developing at such a rapid pace that it is being used on wide range of problems and research fields. Examples include history departments using it to analyze historical documents to determine authorship to biomedical departments employing it to predict DNA sequencing.

The field of machine learning is expansive with innumerable different architectures and techniques but most machine learning tasks fall into one of four categories. The first category is known as **supervised learning**. This task requires training the network on labeled data. In this case, we know what the desired output or label should be given a particular example. The goal of the network is to learn a function that can take input data and output the correct label. All of the research in this thesis falls under the supervised learning category and will be the primary focus. The second category is **unsupervised learning**. The data for this task lacks any type of label and the goal of the network is to glean insights about the data. The third category is a mixture between supervised and unsupervised learning tasks called **semi-supervised learning**. In this case, some of the data is labeled and some of it is not. The last category is known as **reinforcement learning**. In this task, the network is to learn how to perform a particular task by exploring different options. An example of this type of learning is allowing a network to learn how to successfully navigate a maze.

In the most general case, the ultimate goal in any of the above categories is to find some nonlinear function or network that takes an input example x and maps it to a desired output y . The input x consists of **features** that represents a particular example or set of examples. This could consist of the pixels of an image, a vector representing a particular word, or the financial record of an applicant. In order to measure how well the network performs, an appropriate **loss** function must be selected as described further in Section 1.5. For each task, the architecture of the network is usually predetermined and fixed by the user and so the goal is to obtain the optimal function or network from a set of parametric family of functions. This is typically done by performing **gradient descent**, see Section 1.6. Gradient descent involves iteratively minimizing the chosen loss function by updating the **weights** associated with the network.

1.2 Supervised Learning

In supervised learning, labeled data is used to optimize the performance of the network. Examples of data used in supervised learning include images of objects, audio files, video files, and written text. The corresponding labels for these types of data sets could consist of the classification or type of object in the image, the next spoken word in the audio sequence, location or absence of cars in the video, and authorship of the selected text. The goal of the machine learning network is to learn the optimal network or function from a certain parametric family of functions to fit the data.

For most tasks, the labeled data is separated into three different sets. The first set is known as the **training data set** and is used to iteratively update the weights of the network by minimizing some appropriate loss function. It should be noted that it is important that networks do not simply perform well on the training data set but perform well on unseen data sets. Having a network that simply spits out the correct label on data that is already labeled is not helpful or useful. The goal is to use the trained network on unlabeled data in order to obtain the appropriate label. In other words, we want the network to be able to **generalize** well.

In order to obtain sufficient generalization, the second and third data sets are used. The second set of data is known as the **validation data set**. The network is never trained on this data set. It is only used to help the user to design the optimal network architecture or to determine what family of functions the network should be restricted to. The performance or loss score of the network on this data set is an indication of how well the network will generalize. The user will use this to determine the number of layers, the type of functions to use, and other **hyperparameters** that are set by the user to determine which architecture or family of functions to use. By doing this, the user is indirectly optimizing the network on the validation data set. See Figure 1.1 for more details.

Unfortunately, this may again result in selecting a network that has poor generalization because the architecture was designed to indirectly optimize performance on the validation data set. To obtain an approximation of how well the network will actually generalize, the performance of the network on the third data set, the **test data set**, is used. The network is never trained on this data set and it is good practice that the user should not take a close look at this data set to avoid biasing their hyperparameter decisions which could inadvertently result in poor generalization.

1.3 Feedforward Networks

A very common machine learning architecture is known as the **feedforward network (FFN)**. In some cases the FFN is referred to as a **multilayer perceptron (MLP)**. In other cases the MLP is identified as a type of FFN but these terms are commonly used interchangeably. Other names include **deep neural networks (DNNs)** or **multilayer networks (MLNs)**. In this text we will try to use the terminology of FFN or feedforward network.

FFNs were originally designed to emulate biological neurons. To illustrate how, we consider a single neuron which we call a **perceptron**. The perceptron receives an

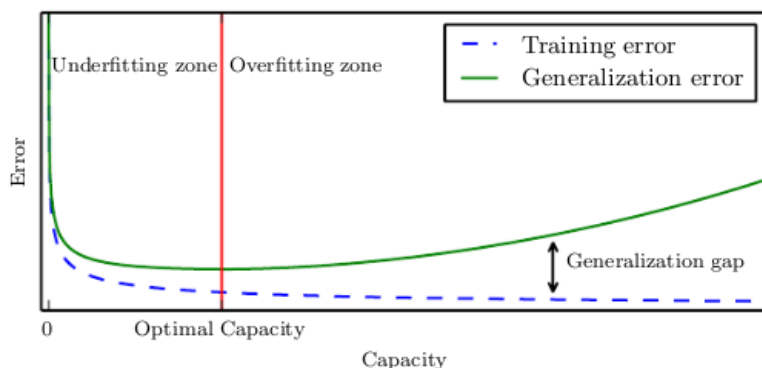


Figure 1.1: This figure shows how a validation data set can be used to select a network that will generalize well. Here the x -axis indicates the capacity of the network and the y -axis is the error score. The blue dashed line shows the performance of the network on the training data set and the green line is the performance of the network on the validation data set which is an indication of how the network will generalize. As can be seen, the user should adjust the size or capacity of the network until achieving the minimal validation or generalization error. If the network is too small, the network will underfit the data and if it is too large it will overfit the data. Image is from [12].

input signal $x \in \mathbb{R}^n$ and applies a weight, w_i , to each individual component of the input, x_i , and sums the result. If the resulting sum is greater than some threshold, $b \in \mathbb{R}$, then the perceptron fires and passes the result forward through the network. If it is less than the given threshold, the perceptron does not fire and the signal is not propagated forward. This is similar to what happens in biological neurons. Mathematically, this can be modeled by $f(x) = \sigma(w^T x + b)$ where $w \in \mathbb{R}^n$ is a vector of the given weights and $\sigma(x)$ is a pointwise nonlinearity. In this example, we suppose the nonlinearity is the **rectified linear unit (ReLU)** nonlinearity or $\sigma(x) = \max\{0, x\}$. In practice there are many different nonlinearities that can be used, see Section 1.4, but for now we assume it is ReLU. A diagram of how a perception works is given in Figure 1.2.

Similar to how biological systems can contain many interconnected neurons, perceptrons can be stacked in a layer, see Figure 1.3. We call such a layer of perceptrons a single layer of a FFN. In this case, the weight vector w is replaced with a weight matrix $W \in \mathbb{R}^{p \times n}$ where $W_{i,j}$ is the weight associated with the i th perceptron that is applied to the j th input and $b \in \mathbb{R}^p$ becomes a vector,

$$f(x) = \sigma(Wx + b) \tag{1.1}$$

In this case there are p perceptrons in the layer. We note that in some cases it is more intuitive to let $W_{i,j}$ be the weight associated with the j th perceptron that is applied to the i th input. In this case, just replace W in (1.1) with W^T .

Like neurological systems, multiple layers of perceptrons can be stacked together and this is where the term **deep neural network** comes from. An example of a 3

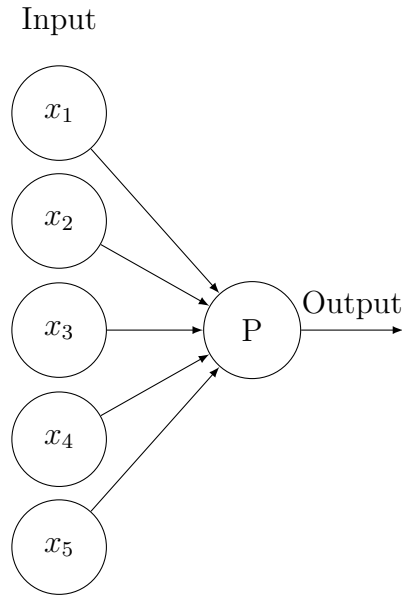


Figure 1.2: Diagram of a perceptron, P. The perceptron takes an input $x \in \mathbb{R}^n$ and applies an affine transformation $w^T x + b$ followed by a pointwise nonlinearity $\sigma(\cdot)$.

layer network is provided in Figure 1.4 and the following equation

$$f(x) = \sigma(W^{(3)}\sigma(W^{(2)}\sigma(W^{(1)}x + b^{(1)}) + b^{(2)}) + b^{(3)}) \quad (1.2)$$

In Figure 1.4 we denote $h^{(k)}$ as the k th layer of the network which consists of a vector of perceptrons. Here instead of using the notation of a single perceptron as $P_i^{(k)}$, we denote individual perceptrons as $h_i^{(k)}$. This is the notation we will be using for the rest of the paper. Each of these layers has an associated weight matrix, $W^{(k)}$, bias vector, $b^{(k)}$, and pointwise nonlinearity. As described previously, each perceptron layer performs an affine transformation followed by a pointwise nonlinearity.

We should note that the architecture of a FFN can vary greatly than the one shown in Figure 1.4. The architecture shown is known as a **fully connected neural network** since the output of each perceptron of the former layer is used as input to each perceptron in the latter layer. In other words, each of the perceptrons are connected between layers of the network. In some cases, the user may decide not to have each perceptron feed into each of the following perceptrons or they may want to take the output of the k th layer and feed it into the $k + 1$ layer as well as the $k + n$ layer. This last connection scheme is sometimes known as a **skip connection** and is used to avoid vanishing gradients. The **convolutional neural network** is a popular type of feedforward network where filters containing weights are convolved with layer inputs. Currently there are many variations of FFNs and it would be impossible to include a description for each one. These variations of the FFN will be discussed where appropriate.

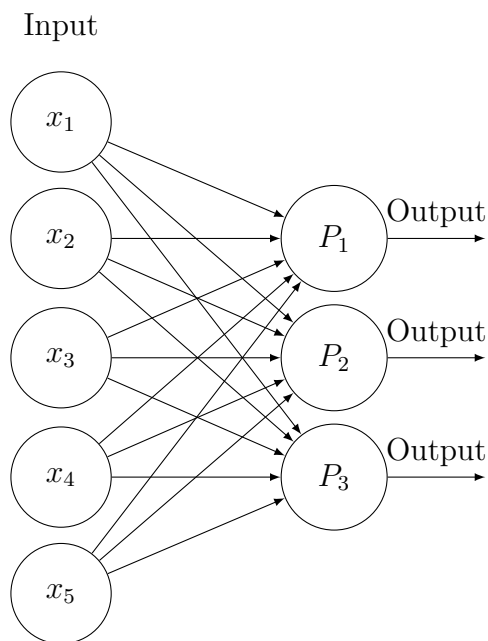


Figure 1.3: Diagram of a single layer FFN. The layer takes the input $x \in \mathbb{R}^n$ and applies an affine transformation $Wx + b$ followed by a pointwise nonlinearity $\sigma(\cdot)$.

1.4 Nonlinear Functions

As discussed in Section 1.3, a typical layer in a FFN consists of an affine transformation followed by a nonlinearity that is applied pointwise or to each element of the output vector. In this section we present four of the most common nonlinear functions that are used in machine learning. These are the sigmoid, hyperbolic tangent, rectified linear unit, and leaky rectified linear unit functions.

Sigmoid

One of the most basic activation functions is the **sigmoid** activation function as shown in Figure 1.5 and defined by

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.3}$$

The output of the sigmoid function has a range of $(0, 1)$ which can be interpreted as a measure of confidence or probability. For example, suppose we want to determine whether an image contains a car or not. If the output is greater than or equal to 0.5, we conclude there is indeed a car in the image and if it is less than 0.5, we conclude that there is no car in the image or vice-versa. The actual output value in this scenario can be interpreted as how confident the network is in whether there is a car. If the sigmoid function returns a value of 0.75, the network believes there is a car in the image with 75% confidence or probability. If the output is 0.10, the

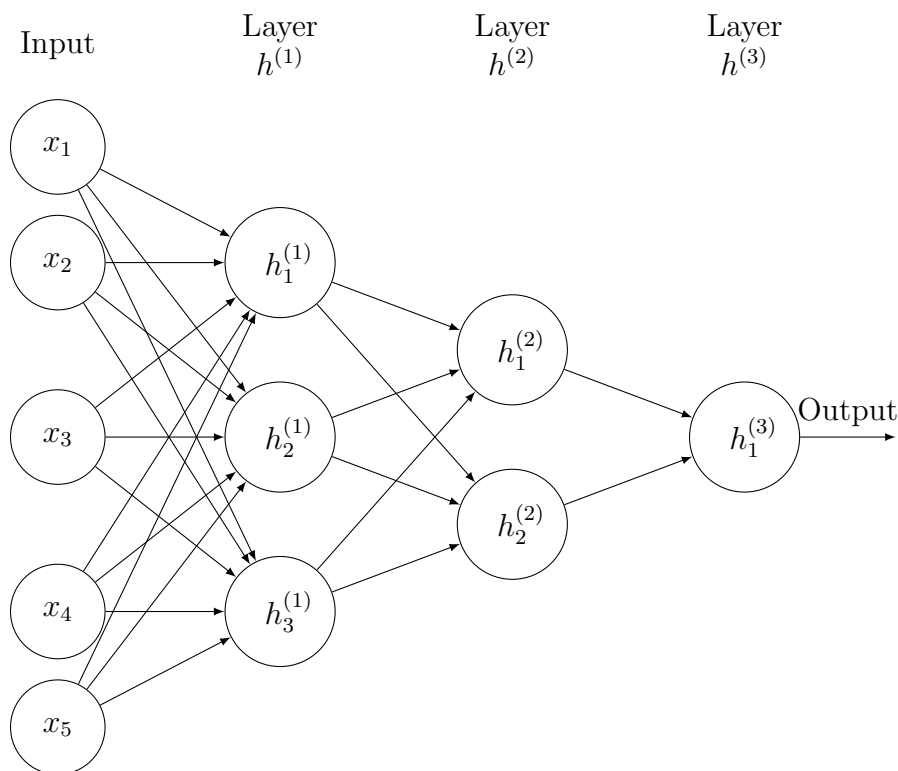


Figure 1.4: Diagram of a multilayer feedforward network. Each layer takes the input $x \in \mathbb{R}^n$ and applies an affine transformation $Wx + b$ followed by a pointwise nonlinearity $\sigma(\cdot)$.

probability of a car in the image is only 10% and so we label the image as most likely not containing a car.

In addition to being interpretable, the sigmoid function has a very simple derivative,

$$\sigma'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = \frac{1}{1 + e^{-x}} \frac{1 + e^{-x} - 1}{1 + e^{-x}} = \sigma(x) (1 - \sigma(x)) \quad (1.4)$$

Thus the computation of the gradients used in gradient descent, see Section 1.6, are relatively easy to compute. A major disadvantage of the sigmoid function occurs when the magnitude of the activation is large. If the activation is either very positive or negative, the gradient of the sigmoid function becomes close to zero. In this case, little or no update to the weights occur during gradient descent. In this case, the network becomes “stuck” and weights will not be updated.

Hyperbolic Tangent

The hyperbolic tangent or **tanh** function behaves similar to the sigmoid function with a few differences. First, the output of the function has a range of $(-1, 1)$. Using the

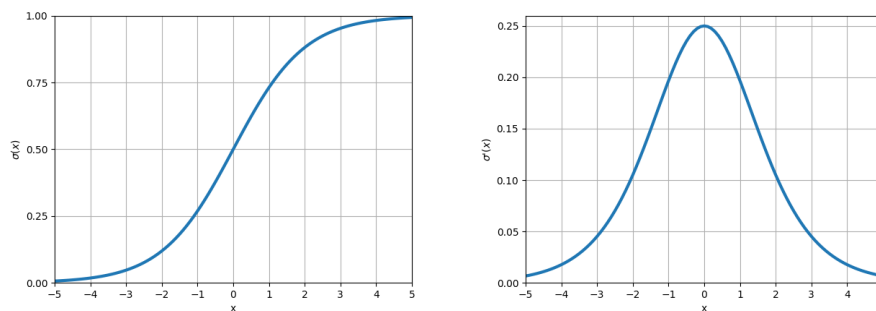


Figure 1.5: Plot of the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ (Left) and its derivative $\sigma'(x) = \sigma(x)(1 - \sigma(x))$ (Right).

similar example of determining whether an image contains a car, one can interpret a positive output as indicating the presence of a car and a negative output as indicating the absence of a car or vice-versa. The tanh function derivative is simply $1 - \tanh^2(x)$. Note that the derivatives of tanh can be much larger than the sigmoid function and the output can be positive or negative. In practice, the tanh function is typically used more often than the sigmoid function.

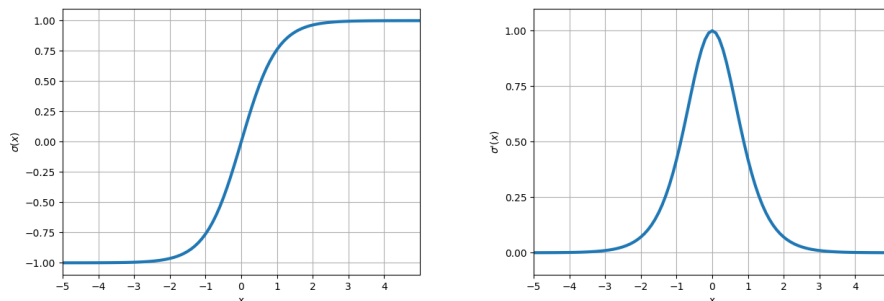


Figure 1.6: Plot of the tanh function $\sigma(x) = \tanh(x)$ (Left) and its derivative $\sigma'(x) = 1 - \tanh^2(x)$ (Right).

Rectified Linear Unit

The **rectified linear unit** or **ReLU** is arguably one of the most popular activation functions due to its computational simplicity,

$$\sigma(x) = \max\{0, x\} \tag{1.5}$$

The ReLU nonlinearity simply sets all negative values to zero and all positive values pass through. The resulting gradient is either 0 or 1. It should be noted that computer programs are designed to address the discontinuity of the gradient at this point by either setting the gradient either to 1 or 0. See Figure 1.7 for a plot of the ReLU function and its derivative.

Although computationally simple to implement, the ReLU function can result in dead perceptrons. If the input value is negative, the ReLU will return a value of zero and the gradients associated with the perceptron weights will also be zero. In this case, the weights will not be updated during gradient descent. Thus if the weights of a network are not initialized correctly or if the weights become such that the input to the ReLU is always nonpositive at certain perceptrons, these perceptrons are always dead and contribute nothing to the final output of the network. These dead perceptrons then serve only to use up computational memory.

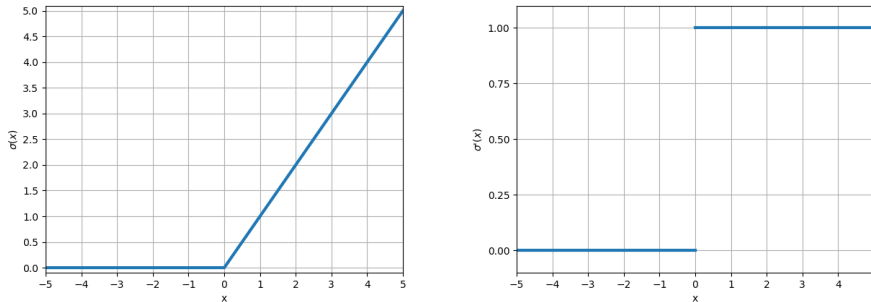


Figure 1.7: Plot of the ReLU function $\sigma(x) = \max\{0, x\}$ (Left) and its derivative $\sigma'(x) = 0$ or 1 (Right).

Leaky Rectified Linear Unit

As previously discussed, the ReLU function is a very popular function used in machine learning but it may result in dead perceptrons. To avoid dead perceptrons, the leaky rectified linear unit or **leaky ReLU** can be used. Instead of setting all negative values to zero, the leaky ReLU returns the negative value scaled by a small α ,

$$\sigma(x) = \begin{cases} \alpha x & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \quad (1.6)$$

A typical default value of α is around 0.20. The simplicity of implementing the leaky ReLU is comparable to the ReLU but has the advantage that gradients are always positive. Thus all weights associated with perceptrons are trained during gradient descent and no perceptron can become dead. In addition, perceptrons can now have both positive and negative values. A plot of the leaky ReLU and the derivative is shown in Figure 1.8.

1.5 Loss Functions

In this section, we discuss three common loss functions that are used in supervised learning tasks. These would be the **cross-entropy** loss function for binary and multi-class classification tasks and the **mean squared error (MSE)** loss function. In this

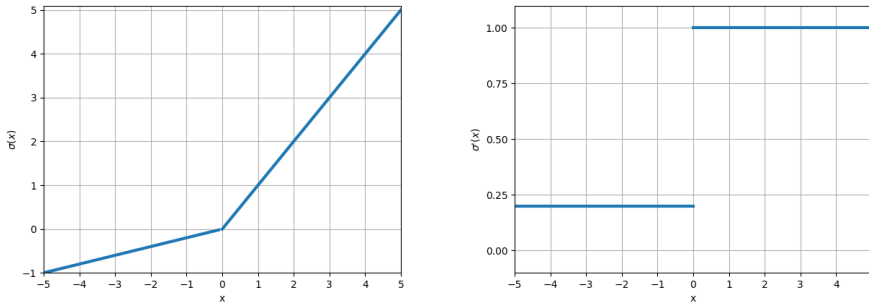


Figure 1.8: Plot of the Leaky ReLU function (Left) and its derivative (Right) using $\alpha = 0.2$.

section, we approach these loss functions from the perspective of the **maximum likelihood principle** as discussed in [12].

In supervised machine learning tasks, we are given a data set $X = [x_1, \dots, x_n]$ of n training examples where each x_j is independently drawn from the probability distribution of the data, $p_{\text{data}}(x)$. We are also given a set of labels $Y = [y_1, \dots, y_n]$. We want the probability distribution of the network, $p_{\text{model}}(x)$, to approximate $p_{\text{data}}(x)$. In other words, if we let θ represent the parameters associated with the network, we want to maximize the conditional probability of the model $p_{\text{model}}(Y|X, \theta)$ with respect to θ . Using the fact that the examples are drawn independently, we want to find the weights θ^* such that

$$\theta^* = \arg \max_{\theta} p_{\text{model}}(Y|X, \theta) = \arg \max_{\theta} \prod_{i=1}^n p_{\text{model}}(y_i|x_i, \theta) \quad (1.7)$$

The problem with (1.7) occurs when several of the $p_{\text{model}}(y_i|x_i, \theta)$ terms are close to zero. When this happens, the entire product will be close to zero. To avoid this, we simply apply a logarithm so we can change the product to a sum which doesn't change the maximum argument. We can also scale it by $\frac{1}{n}$ so we can use the expected value. Finally, we can change the problem to a minimization problem by adding a negative.

$$\theta^* = \arg \max_{\theta} \sum_{i=1}^n \log(p_{\text{model}}(y_i|x_i, \theta)) \quad (1.8)$$

$$= \arg \max_{\theta} \frac{1}{n} \sum_{i=1}^n \log(p_{\text{model}}(y_i|x_i, \theta)) \quad (1.9)$$

$$= \arg \max_{\theta} \mathbb{E}_{x \sim X} [\log(p_{\text{model}}(y|x, \theta))] \quad (1.10)$$

$$= \arg \min_{\theta} \mathbb{E}_{x \sim X} [-\log(p_{\text{model}}(y|x, \theta))] \quad (1.11)$$

Binary Cross-Entropy Loss

Suppose we only have two types of classes with labels $y_i \in \{0, 1\}$. Now let \hat{y}_i be the predicted label given by the model which is the output of a sigmoid function. Thus $\hat{y}_i \in (0, 1)$. We use the Bernoulli distribution for our model and replace $p_{\text{model}}(y_i|x_i, \theta)$ with $\hat{y}_i^{y_i} (1 - \hat{y}_i)^{1-y_i}$. Plugging this into (1.11) we obtain

$$\theta^* = \arg \min_{\theta} \mathbb{E}_{x \sim X} [-\log(\hat{y}^y (1 - \hat{y})^{1-y})] \quad (1.12)$$

$$= \arg \min_{\theta} -\mathbb{E}_{x \sim X} [y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})] \quad (1.13)$$

where the binary cross-entropy loss function is the term inside the argmin.

Multi-Class Cross Entropy Loss

Instead of only having two class types, suppose the data set contains K different classes. The label for the i th example is given by $y^{(i)} \in \{e_1, \dots, e_n\}$ where each $e_j \in \mathbb{R}^K$ with the j th entry 1 and all other entries 0. (Note that we have switched to using a superscript to denote a particular example.) In other words, if the i th example belongs to class s , then $y^{(i)} = e_s$.

For $p_{\text{model}}(y^{(i)}|x^{(i)}, \theta)$, we use the **softmax** function. The softmax function is defined by $\hat{y} = \sigma(z) : \mathbb{R}^K \rightarrow \mathbb{R}^K$ where each entry of \hat{y} is

$$\hat{y}_j = \frac{e^{z_j}}{\sum_{i=1}^K e^{z_i}} \quad (1.14)$$

Note that $\sum_{j=1}^K \hat{y}_j = 1$. Now if we denote $y_j^{(i)}$ as the j th entry of the i th example, (1.11) becomes

$$\theta^* = \arg \min_{\theta} \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^K -y_j^{(i)} \log(\hat{y}_j^{(i)}) \quad (1.15)$$

If we denote the label of the i th example as $e_{s(i)}$ then $y_j^{(i)} = 1$ if $j = s(i)$ and 0 otherwise. Thus the multi-class cross entropy loss function is the term inside the argmin below

$$\theta^* = \arg \min_{\theta} \frac{1}{n} \sum_{i=1}^n -\log(\hat{y}_{s(i)}^{(i)}) \quad (1.16)$$

We should note that for assigning the final label to an example, the class associated with the largest entry of the softmax output vector is used.

Mean Square Error

For regression tasks, the mean square error (MSE) loss function is used. If y_i is the label and \hat{y}_i the predicted output of the network, then (1.11) becomes

$$\theta^* = \arg \min_{\theta} \frac{1}{n} \sum_{i=1}^n \|y_i - \hat{y}_i\|_2^2 \quad (1.17)$$

where the MSE loss function is the term inside the argmin.

To see how MSE loss relates to the maximum likelihood principle, suppose $p_{\text{model}}(y_i|x_i, \theta)$ follows a Gaussian distribution with standard deviation σ . Thus

$$p_{\text{model}}(y_i|x_i, \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{\|y_i - \hat{y}_i\|_2^2}{2\sigma^2}} \quad (1.18)$$

Plugging (1.18) into (1.11) we obtain

$$\theta^* = \arg \min_{\theta} -\frac{1}{n} \sum_{i=1}^n \log \left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{\|y_i - \hat{y}_i\|_2^2}{2\sigma^2}} \right) \quad (1.19)$$

$$= \arg \min_{\theta} -\frac{1}{n} \sum_{i=1}^n \left[\log \left(\frac{1}{\sqrt{2\pi\sigma^2}} \right) - \frac{1}{2\sigma^2} \|y_i - \hat{y}_i\|_2^2 \right] \quad (1.20)$$

Since the log term in (1.20) is a constant, we can ignore this term and obtain

$$\theta^* = \arg \min_{\theta} \frac{1}{n} \sum_{i=1}^n \frac{1}{2\sigma^2} \|y_i - \hat{y}_i\|_2^2 \quad (1.21)$$

$$= \arg \min_{\theta} \frac{1}{n} \sum_{i=1}^n \|y_i - \hat{y}_i\|_2^2 \quad (1.22)$$

$$(1.23)$$

Thus the MSE loss function is related to the maximum likelihood principle.

1.6 Gradients

For a typical FFN, the final output of the network, $o \in \mathbb{R}^q$, is fed into a loss function which we denote as \mathcal{L} . See Section 1.5 for examples of common loss functions. Since it is not known a priori what the optimal weights and biases are that minimizes \mathcal{L} , the network parameters are randomly initialized following a set distribution such as Gaussian normal. The parameters are then iteratively updated by using a gradient descent algorithm or **optimizer** such as stochastic gradient descent (SGD), see Section 1.7. In SGD, a batch of samples are fed into the network and the loss is computed over the batch. This step is known as the **forward-pass**. The gradient of the loss function with respect to the various parameters is then computed. This is known as the **backward-pass**. The resulting gradients are then used by the selected optimizer to update the parameters. For SGD, the update step is simply

$$\theta^{(k+1)} \leftarrow \theta^{(k)} - \lambda \frac{\partial \mathcal{L}}{\partial \theta^{(k)}} \quad (1.24)$$

where θ denotes the network weights and biases and λ is the learning rate or step size. To compute the gradients, the standard chain-rule in calculus is applied starting from \mathcal{L} and working backwards to the desired parameter.

To see how backpropagation of the gradients works, consider a simple fully-connected feedforward network with L layers. Each layer k has an associated weight matrix $W^{(k)}$ and bias vector $b^{(k)}$. For simplicity, let us consider a single training example which is a vector x consisting of various features. We feed the example through the network starting with layer 1 and proceed to layer L . At layer 1, an affine transformation is applied $z^{(1)} = W^{(1)}x + b^{(1)}$ followed by a point-wise nonlinearity $h^{(1)} = \sigma(z^{(1)})$. The hidden state or output of the first layer is then feed into the second layer where another affine transformation is applied $z^{(2)} = W^{(2)}h^{(1)} + b^{(2)}$. This is again followed by another pointwise nonlinearity $h^{(2)} = \sigma(z^{(2)})$. This continues through the network to the final layer where the output is given by $o = W^{(L)}h^{(L-1)} + b^{(L)}$. Note that it is typical not to apply a nonlinearity to the final layer output but to incorporate any appropriate nonlinearity into the loss function. This output and the actual data label is then used in the loss function $\mathcal{L}(y, o)$.

To update the weights in the network, we compute the gradients by using the simple chain-rule from calculus. Computed gradients for the last layer L , k th layer, and first layer weights and biases are provided in the following equations. We should note that vectors and gradient vectors are considered to be vertical vectors in this notation.

$$\frac{\partial \mathcal{L}}{\partial b^{(L)}} = \frac{\partial \mathcal{L}}{\partial o} \tag{1.25}$$

$$\frac{\partial \mathcal{L}}{\partial W^{(L)}} = \frac{\partial \mathcal{L}}{\partial o} h^{(L-1)T} \tag{1.26}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(k)}} = \frac{\partial \mathcal{L}}{\partial z^{(k)}} \tag{1.27}$$

$$\frac{\partial \mathcal{L}}{\partial W^{(k)}} = \frac{\partial \mathcal{L}}{\partial z^{(k)}} h^{(k-1)T} \tag{1.28}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(1)}} = \frac{\partial \mathcal{L}}{\partial z^{(1)}} \tag{1.29}$$

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} = \frac{\partial \mathcal{L}}{\partial z^{(1)}} x^T \tag{1.30}$$

Note in (1.25) through (1.30) the term $\frac{\partial \mathcal{L}}{\partial z^{(k)}}$ can be expanded using the chain-rule. If we let $G^{(k)} = \text{diag} \left(\left[\sigma'(z_1^{(k)}), \dots, \sigma'(z_n^{(k)}) \right] \right)$ then:

$$\frac{\partial \mathcal{L}}{\partial z^{(k)}} = G^{(k)} \frac{\partial \mathcal{L}}{\partial h^{(k)}} \tag{1.31}$$

$$= G^{(k)} \left[\frac{\partial h^{(k+1)}}{\partial h^{(k)}} \right]^T \frac{\partial \mathcal{L}}{\partial h^{(k+1)}} \tag{1.32}$$

$$= G^{(k)} \left[G^{(k+1)} W^{(k+1)} \right]^T \left[\frac{\partial h^{(k+2)}}{\partial h^{(k+1)}} \right]^T \frac{\partial \mathcal{L}}{\partial h^{(k+2)}} \tag{1.33}$$

$$= G^{(k)} \left(\prod_{i=k+1}^L \left[G^{(i)} W^{(i)} \right]^T \right) \frac{\partial \mathcal{L}}{\partial o} \tag{1.34}$$

Wirtinger Derivatives

In order to determine the effectiveness of a particular model, a real-valued loss function is selected, as described in Section 1.5, with gradient descent used to update the trainable weights. When using complex valued networks with a real-valued loss function, see Section 2.4, computing the required gradients for gradient descent is substantially different and requires the use of Wirtinger Calculus [47]. In this section we try to explain why Wirtinger Calculus is required and how to compute the necessary gradients for a real-valued loss function that has a domain in the complex plane.

In complex analysis, a complex function is considered differentiable if:

Definition 1.6.1. [19, Definition 2.0.1] *Let $A \subset \mathbb{C}$ be an open set. The function $f : A \rightarrow \mathbb{C}$ is said to be (complex) differentiable at $z_0 \in A$ if the limit*

$$\lim_{z \rightarrow z_0} \frac{f(z) - f(z_0)}{z - z_0}$$

exists independent of the manner in which $z \rightarrow z_0$. This limit is then denoted by $f'(z_0)$ and is called the derivative of f with respect to z at the point z_0 .

A well known variation of Definition 1.6.1 involving the Cauchy-Riemann equations is given in the following Theorem.

Theorem 1.6.1. [6] *Let $f : \mathbb{C} \rightarrow \mathbb{C}$ where $f(z) = u(x, y) + iv(x, y)$ with $u, v, x, y \in \mathbb{R}$, $z = x + iy$, and $i = \sqrt{-1}$. Suppose u and v have continuous partial derivatives. Then f is complex differentiable if and only if the Cauchy-Riemann equations are satisfied:*

$$u_x = v_y \text{ and } u_y = -v_x \tag{1.35}$$

Using (1.35), it is now clear that any real-valued cost function that has a complex domain will not be differentiable unless the function is constant. In particular, if $f : \mathbb{C} \rightarrow \mathbb{R}$ we can write $f(z) = u(x, y) + iv(x, y)$ where $z = x + iy$, $x, y, u \in \mathbb{R}$, and $v(x, y) \equiv 0$. From the Cauchy-Riemann equations, $u_x = u_y = 0$ and so $f(z)$ is a constant function. Thus any non-constant cost function will not be complex differentiable [19].

In order to overcome this difficulty, a different approach is required as explained in [19]. Instead of thinking of a function as a function from the complex plane to the complex plane, $f(z) : \mathbb{C} \rightarrow \mathbb{C}$, we consider the function as a function that maps two real-valued inputs to two real-valued outputs, $f(x, y) : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R}$. Looking at the total differentiable of such a function, we obtain

$$dF = \frac{\partial f(x, y)}{\partial x} dx + \frac{\partial f(x, y)}{\partial y} dy \tag{1.36}$$

Now if we suppose $f(x, y) = u(x, y) + iv(x, y)$, we can rewrite (1.36) as

$$df = \frac{\partial u(x, y)}{\partial x} dx + i \frac{\partial v(x, y)}{\partial x} dx + \frac{\partial u(x, y)}{\partial y} dy + i \frac{\partial v(x, y)}{\partial y} dy \tag{1.37}$$

Computing the differentials of $z = x + iy$ we obtain

$$dz = dx + idy \quad (1.38)$$

$$d\bar{z} = dx - idy \quad (1.39)$$

Now adding and subtracting (1.38) and (1.39) we have

$$dx = \frac{1}{2}(dz + d\bar{z}) \quad (1.40)$$

$$dy = \frac{1}{2i}(dz - d\bar{z}) \quad (1.41)$$

Plugging (1.40) and (1.41) into (1.37) and rearranging,

$$\begin{aligned} df = & \frac{1}{2} \left[\frac{\partial u(x, y)}{\partial x} + \frac{\partial v(x, y)}{\partial y} + i \left(\frac{\partial v(x, y)}{\partial x} - \frac{\partial u(x, y)}{\partial y} \right) \right] dz \\ & + \frac{1}{2} \left[\frac{\partial u(x, y)}{\partial x} - \frac{\partial v(x, y)}{\partial y} + i \left(\frac{\partial v(x, y)}{\partial x} + \frac{\partial u(x, y)}{\partial y} \right) \right] d\bar{z} \end{aligned} \quad (1.42)$$

To simplify, we define the following operators which are known as the Wirtinger derivatives:

$$\begin{aligned} \frac{\partial}{\partial z} & := \frac{1}{2} \left(\frac{\partial}{\partial x} - i \frac{\partial}{\partial y} \right) \\ \frac{\partial}{\partial \bar{z}} & := \frac{1}{2} \left(\frac{\partial}{\partial x} + i \frac{\partial}{\partial y} \right) \end{aligned} \quad (1.43)$$

Plugging (1.43) into (1.42) we obtain the following differential.

$$df = \frac{\partial f(z)}{\partial z} dz + \frac{\partial f(z)}{\partial \bar{z}} d\bar{z} \quad (1.44)$$

Now that we have a method to compute gradients, we look at the resulting gradients for a real-valued loss function with a complex domain. Let $f(z) : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$. Since $f(z) \in \mathbb{R}$, we can write $f(z) = u(x, y)$ where $u \in \mathbb{R}$. The differential of such a function is given by

$$df = \frac{\partial u(x, y)}{\partial x} dx + \frac{\partial u(x, y)}{\partial y} dy \quad (1.45)$$

Rewriting (1.45) in terms of $\frac{\partial f(z)}{\partial z}$ and dz using $\mathcal{R}(\cdot)$ as the real-valued component,

$$df = \mathcal{R} \left[\frac{\partial u(x, y)}{\partial x} dx + \frac{\partial u(x, y)}{\partial y} dy + i \left(\frac{\partial u(x, y)}{\partial x} dy - \frac{\partial u(x, y)}{\partial y} dx \right) \right] \quad (1.46)$$

$$= \mathcal{R} \left[\left(\frac{\partial u(x, y)}{\partial x} - i \frac{\partial u(x, y)}{\partial y} \right) (dx + idy) \right] \quad (1.47)$$

$$= 2\mathcal{R} \left[\frac{1}{2} \left(\frac{\partial u(x, y)}{\partial x} - i \frac{\partial u(x, y)}{\partial y} \right) (dx + idy) \right] \quad (1.48)$$

$$= 2\mathcal{R} \left(\frac{\partial f(z)}{\partial z} dz \right) \quad (1.49)$$

Now we consider the general form of the inner product, $\langle \cdot, \cdot \rangle: \mathbb{C} \times \mathbb{C} \rightarrow \mathbb{C}$ given by $\langle z_1, z_2 \rangle = \overline{z_2} z_1$. Using this inner product, the Cauchy-Schwarz inequality, and the fact that since f is real-valued we have $\overline{\left(\frac{\partial f}{\partial z}\right)} = \frac{\partial f}{\partial \bar{z}}$,

$$\mathcal{R} \left[\frac{\partial f(z)}{\partial z} dz \right] = \mathcal{R} \left[\left\langle dz, \frac{\partial f(z)}{\partial \bar{z}} \right\rangle \right] \leq \left| \left\langle dz, \frac{\partial f(z)}{\partial \bar{z}} \right\rangle \right| \leq |dz| \left| \frac{\partial f(z)}{\partial \bar{z}} \right| \quad (1.50)$$

We have equality in (1.50) if dz is real-valued multiple of $\frac{\partial f(z)}{\partial \bar{z}}$ which is the direction of greatest-descent. Thus the gradient descent step for a general real-valued function with a complex domain is given by

$$f_{k+1} = f_k - \gamma \left(2 \frac{\partial f(z)}{\partial \bar{z}} \right) \quad (1.51)$$

where γ is the step size.

1.7 Optimizers

For a given machine learning task, the user restricts the network to a set of parametric functions by setting the architecture. This could include determining the number of layers, the connections between layers, the loss function, the nonlinearity used, and other user defined settings which we call **hyperparameters**. The weights and biases or parameters are then iteratively updated using gradients and a selected optimizer. In this chapter, we describe the four most common optimizers that are used in machine learning as discussed in [12].

Stochastic Gradient Descent (SGD)

The stochastic gradient descent (SGD) optimizer is the most basic optimizer and is the basis for all others. In SGD, the gradients of the loss function with respect to the network parameters, $\frac{\partial \mathcal{L}}{\partial \theta}$, are computed during the backward pass and used to directly update the parameters.

$$\theta^{(k+1)} \leftarrow \theta^{(k)} - \lambda \frac{\partial \mathcal{L}}{\partial \theta^{(k)}} \quad (1.52)$$

Here λ is a small number and denotes the step size. If the step size is too large, the parameters may oscillate too greatly and if too small, the resulting update to the parameters might be too negligible. To visually understand why SGD works, see Figure 1.9.

AdaGrad

The **AdaGrad** optimizer [9] implements an adaptive learning rate. This is done by scaling the gradients by the running sum of gradients squared, see Algorithm 1. For parameters with larger gradients, the effective learning rate is decreased much more than parameters with smaller gradients. This allows for a more even update

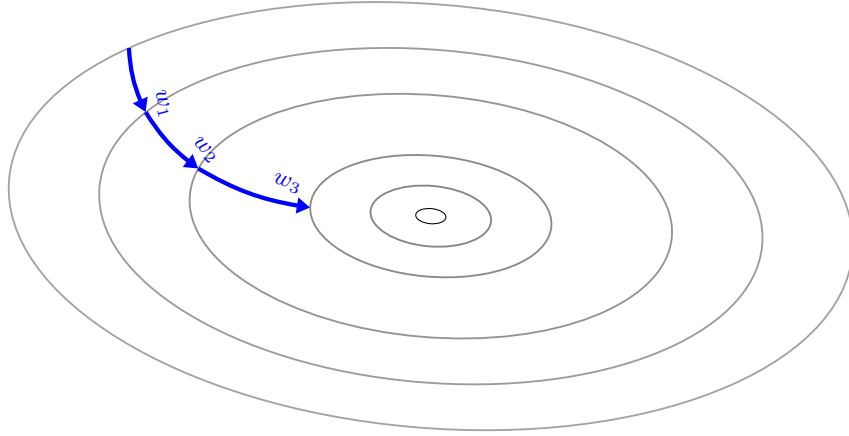


Figure 1.9: A visual explanation of SGD. The circles represent the level curves of a convex loss function, \mathcal{L} , and $w \in \mathbb{R}^2$ are the parameters of the network. The initial parameter w_0 is updated using SGD to w_1 and so on. This continues until the network reaches a sufficient minimum.

of weights and provides for an adaptive learning rate. It should be noted that the accumulated sum of the squared gradients will result in the effective learning rate to decrease and eventually go to zero.

Algorithm 1: Adagrad

Given:

Learning rate $\lambda > 0$

Small $\epsilon > 0$ to avoid division by zero

Running squared gradient term γ (initialized as 0)

Input:

Model parameters θ

Gradient $g = \frac{\partial \mathcal{L}}{\partial \theta}$

Procedure:

Square the gradient entrywise: $g^2 = g \odot g$

Update running gradient: $\gamma \leftarrow \gamma + g^2$

Compute update entrywise: $u \leftarrow \frac{1}{\epsilon + \sqrt{\gamma}} \odot g$

Update parameters: $\theta \leftarrow \theta - \lambda u$

RMSProp

To address the issue of cumulative sum of squared gradients, the **RMSProp** optimizer [17] uses an exponentially decaying average of squared gradients, see Algorithm 2. In RMSProp the effect of a particular squared gradient will decay as the number of

iterations increases.

Algorithm 2: RMSProp

Given:

Learning rate $\lambda > 0$

Decay rate ρ (typical value of 0.9)

Running squared gradient term γ (initialized as 0)

Small $\epsilon > 0$ to avoid division by zero

Input:

Model parameters θ

Gradient $g = \frac{\partial \mathcal{L}}{\partial \theta}$

Procedure:

Square the gradient entrywise: $g^2 = g \odot g$

Update running gradient: $\gamma \leftarrow \rho\gamma + (1 - \rho)g^2$

Compute update entrywise: $u \leftarrow \frac{1}{\sqrt{\gamma + \epsilon}} \odot g$

Update parameters: $\theta \leftarrow \theta - \lambda u$

Adam

The **Adaptive Moment Estimation (Adam)** optimizer [26] uses an exponential decay on both the gradients and squared gradients. The moving average of the gradients can be considered as an estimate of the first-order moment or mean and the moving average of the squared gradients can be considered as an estimate of the second-order raw moment of the uncentered variance. In the Adam algorithm, the first and second-order moment estimates are initialized as zero and thus are initially biased towards zero. Note that this is the same as in the RMSProp optimizer. To mitigate this initial bias, the Adam optimizer scales the momentum estimates by a

correcting factor. See Algorithm 3.

Algorithm 3: Adam

Given:

Learning rate $\lambda > 0$

Decay rate ρ_1 (typical value of 0.9)

Decay rate ρ_2 (typical value of 0.99)

Running 1st moment term γ_1 (initialized as 0)

Running 2nd moment term γ_2 (initialized as 0)

Small $\epsilon > 0$ to avoid division by zero

Number of update iterations: t (initialized as 0)

Input:

Model parameters θ

Gradient $g = \frac{\partial \mathcal{L}}{\partial \theta}$

Procedure:

Update 1st Moment: $\gamma_1 \leftarrow \rho_1 \gamma_1 + (1 - \rho_1)g$

Square the gradient entrywise: $g^2 = g \odot g$

Update 2nd Moment: $\gamma_2 \leftarrow \rho_2 \gamma_2 + (1 - \rho_2)g^2$

Update number of iterations: $t \leftarrow t + 1$

Correct 1st Moment: $\hat{\gamma}_1 \leftarrow \frac{\gamma_1}{1 - \rho_1^t}$

Correct 2nd Moment: $\hat{\gamma}_2 \leftarrow \frac{\gamma_2}{1 - \rho_2^t}$

Compute update entrywise: $u \leftarrow \frac{1}{\epsilon + \sqrt{\hat{\gamma}_2}} \odot \hat{\gamma}_1$

Update parameters: $\theta \leftarrow \theta - \lambda u$

Chapter 2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a type of FFN that are designed to process sequential data. The goal of such networks is to capture temporal dependencies between inputs of the sequence. They are most commonly used in tasks such as speech recognition and text prediction. For example, given a sequence of words: “The color of the sun is”, the task of the RNN is to predict the next word in the sequence which in this example would be “yellow”. Standard FFNs are ill-suited for such tasks as they lack any memory mechanism to capture the information contained in the previous five words before the marker word “is”. On the other hand, an RNN is equipped with a hidden state that is designed to act like memory in order to capture information from the previous inputs. In our example, key phrases would be “color”, “sun”, and “is” which hopefully would be encoded in the hidden state of the RNN in order for it to predict the next word being “yellow”.

Unlike an FFN, RNNs are a dynamical system. In particular, given an input sequence of variable length τ , denoted $X_\tau = \{x_1, x_2, \dots, x_\tau\}$, each $x_i \in \mathbb{R}^n$ is fed into the RNN sequentially. The goal of the RNN is to output a desired sequence $Y_\tau = \{y_1, y_2, \dots, y_\tau\}$ where each $y_i \in \mathbb{R}^p$. The network predicts corresponding outputs $o_i \in \mathbb{R}^p$ that are computed as follows:

$$a_i = Ux_i + Wh_{i-1} + b \quad (2.1)$$

$$h_i = \sigma(a_i) \quad (2.2)$$

$$o_i = Vh_i + c \quad (2.3)$$

where $U \in \mathbb{R}^{m \times n}$ is the input weight matrix, $W \in \mathbb{R}^{m \times m}$ the recurrent weight matrix, $b \in \mathbb{R}^m$ the input bias, $V \in \mathbb{R}^{p \times m}$ the output weight matrix, and $c \in \mathbb{R}^p$ the output bias. Here $\sigma(\cdot)$ is a nonlinearity function that is applied pointwise and $H_\tau = \{h_0, h_1, \dots, h_{\tau-1}\}$, $h_i \in \mathbb{R}^m$ is the hidden state that is passed recurrently into the model at each time step. It should be noted that for implementation purposes, the initial hidden state is initialized as zero, $h_0 = 0$, or is considered trainable. See Figure 2.1 for a graphical representation of how a standard RNN processes sequential data.

RNNs are not just limited to tasks that require producing a desired sequence at each time step, but they can also be used in classification tasks. This is done by restricting the network to a single output at the end of the entire sequence. This final output is used to determine which class the input sequence belongs to. In such a system, (2.3) is replaced by:

$$o_\tau = Vh_\tau + c \quad (2.4)$$

Note in this case that there is only one output at time step τ . A graphical representation of such a network is shown in Figure 2.2.

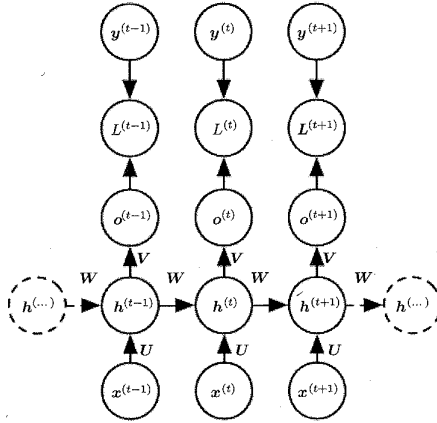


Figure 2.1: Diagram of a standard RNN. Here $y^{(t)}$ is the actual label and $o^{(t)}$ is the predicted output by the RNN and L is some loss function. Superscript indicates the time step. Image is from [12].

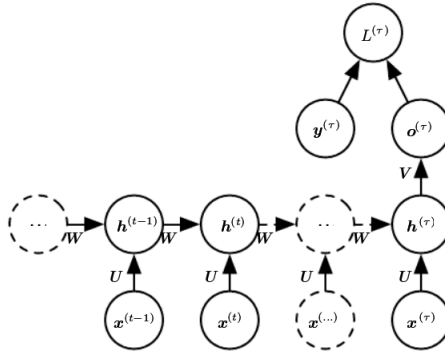


Figure 2.2: Diagram of an RNN structure used for classification. Note only one final output $o^{(\tau)}$ at the end of the sequence. Superscript indicates the time step. Image is courtesy of [12].

2.1 Back-Propagation Through Time

Similar to a FFN, the trainable parameters of an RNN are updated by minimizing some appropriate loss function through gradient descent. Since each sequential output of an RNN is dependent on the previous outputs and parameters are shared across each time step through h_i , the computed gradients used in gradient descent must be computed using **back-propagation through time (BPTT)**. Using the previous notation in (2.1) through (2.3) with $\mathcal{L} : \mathbb{R}^p \rightarrow \mathbb{R}$ a given loss function, the gradients

of the trainable parameters using BPTT are given below:

$$\frac{\partial \mathcal{L}}{\partial c} = \sum_{t=1}^{\tau} \frac{\partial \mathcal{L}}{\partial o_t} \quad (2.5)$$

$$\frac{\partial \mathcal{L}}{\partial V} = \sum_{t=1}^{\tau} \frac{\partial \mathcal{L}}{\partial o_t} h_t^T \quad (2.6)$$

$$\frac{\partial \mathcal{L}}{\partial b} = \sum_{t=1}^{\tau} D_t \frac{\partial \mathcal{L}}{\partial h_t} \quad (2.7)$$

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_{t=1}^{\tau} D_t \frac{\partial \mathcal{L}}{\partial h_t} h_{t-1}^T \quad (2.8)$$

$$\frac{\partial \mathcal{L}}{\partial U} = \sum_{t=1}^{\tau} D_t \frac{\partial \mathcal{L}}{\partial h_t} x_t^T \quad (2.9)$$

where T represents the transpose operator and $D_t \in \mathbb{R}^{m \times m}$ is a diagonal matrix with entries consisting of the derivative of the activation values as follows:

$$D_t = \begin{bmatrix} \sigma'(a_t)_1 & & & & \\ & \sigma'(a_t)_2 & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & \sigma'(a_t)_m \end{bmatrix}$$

2.2 Vanishing/Exploding Gradients

A major obstacle for training RNNs is the **vanishing/exploding gradient problem** as discussed in [1], [40], and [2]. The vanishing gradient problem occurs when the gradients tend towards zero. When this occurs, the gradients become negligible in size and so the resulting gradient descent step will result in little to no update in the network parameters. In this case, no training occurs. On the other hand, the exploding gradient problem occurs when the size of the gradients become unbounded. This can result in the values of the parameters changing drastically, causing the network to overstep local minimums. In some cases, overflow issues can occur.

To see how vanishing/exploding gradients are a problem for RNNs, we examine (2.5) through (2.9). The gradients for each recurrent parameter W , U , and b , contain the term $\frac{\partial \mathcal{L}}{\partial h_t}$. Expanding on this term, we have

$$\frac{\partial \mathcal{L}}{\partial h_t} = \left[\frac{\partial \mathcal{L}}{\partial h_\tau} \frac{\partial h_\tau}{\partial h_{\tau-1}} \cdots \frac{\partial h_{t+2}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_t} \right]^T \quad (2.10)$$

$$= \left[\frac{\partial \mathcal{L}}{\partial h_\tau} \left(\prod_{k=\tau}^{t+1} \frac{\partial h_k}{\partial h_{k-1}} \right) \right]^T \quad (2.11)$$

$$= \left[\frac{\partial \mathcal{L}}{\partial h_\tau} \left(\prod_{k=\tau}^{t+1} D_k W \right) \right]^T \quad (2.12)$$

Taking the Euclidean ℓ_2 -norm to both sides of (2.12) and using the fact that W is a square matrix, we have:

$$\left\| \frac{\partial \mathcal{L}}{\partial h_t} \right\|_2 \leq \left(\prod_{k=t+1}^{\tau} \|D_k W\|_2 \right) \left\| \frac{\partial \mathcal{L}}{\partial h_\tau} \right\|_2 \quad (2.13)$$

$$\leq \left(\prod_{k=t+1}^{\tau} \|W\|_2 \|D_k\|_2 \right) \left\| \frac{\partial \mathcal{L}}{\partial h_\tau} \right\|_2 \quad (2.14)$$

Now if we assume that the pointwise nonlinearity σ is a ReLU function [37] that has a derivative of either zero or one and at least one entry of $a_k > 0$ for all k we have $\|D_k\| = 1$ and

$$\left\| \frac{\partial \mathcal{L}}{\partial h_t} \right\|_2 \leq \left(\prod_{k=t+1}^{\tau} \|W\|_2 \right) \left\| \frac{\partial \mathcal{L}}{\partial h_\tau} \right\|_2 \quad (2.15)$$

$$= \|W\|_2^{\tau-t} \left\| \frac{\partial \mathcal{L}}{\partial h_\tau} \right\|_2 \quad (2.16)$$

If the largest singular value of W is greater than 1, i.e. $\|W\|_2 > 1$, then

$$\|W\|_2^{\tau-t+1} \rightarrow \infty \text{ as } \tau \rightarrow \infty \quad (2.17)$$

and the upper bound in (2.16) goes to infinity. Although this doesn't necessarily imply that $\left\| \frac{\partial \mathcal{L}}{\partial h_t} \right\|_2$ is unbounded, it has the potential to become unbounded. This is what is known as the exploding gradient problem. On the other hand, if the largest singular value of W is less than 1, i.e. $\|W\|_2 < 1$, then

$$\|W\|_2^{\tau-t+1} \rightarrow 0 \text{ as } \tau \rightarrow \infty \quad (2.18)$$

and the upper bound in (2.16) becomes zero and so $\left\| \frac{\partial \mathcal{L}}{\partial h_t} \right\|_2$ goes to zero as well. This is known as the vanishing gradient problem.

2.3 Gated Recurrent Neural Networks

As discussed in Section 2.2, a major obstacle for RNNs to overcome is the vanishing/exploding gradient problem. Several different architectures have been proposed to help mitigate these effects with the most common type being gated RNNs. This class of RNNs use gating mechanisms to control when information is passed or discarded from one time step to another and provide a different path for gradients to pass through when performing BPTT. The two most common types are the **Long Short-Term Memory RNN (LSTM)** [18] and the **Gated Recurrent Unit (GRU)** [8].

LSTM

By far the most popular RNN is the LSTM. The LSTM is designed with three gating mechanisms: the input gate; forget gate; and output gate. The input gate, i_t , is designed to extract only relevant data from the input sequence to pass into the network. The forget gate, f_t , is designed to control what information from the previous LSTM cell state is passed onto the current LSTM cell state. Finally, the output gate, o_t , controls what information is output from the network. In order to determine what information should be discarded or kept, each gate performs an affine transformation on the input, x_t , and output, h_t , of the network with a elementwise sigmoid nonlinearity, $\sigma(\cdot)$. The equations for a basic architecture are below:

$$i_t = \sigma(U^i x_t + W^i h_{t-1} + b^i) \quad (2.19)$$

$$f_t = \sigma(U^f x_t + W^f h_{t-1} + b^f) \quad (2.20)$$

$$o_t = \sigma(U^o x_t + W^o h_{t-1} + b^o) \quad (2.21)$$

$$s_t = f_t \odot s_{t-1} + i_t \odot \sigma(U^s x_t + W^s h_{t-1} + b^s) \quad (2.22)$$

$$h_t = o_t \odot \tanh(s_t) \quad (2.23)$$

Although quite successful in practice, the LSTM requires roughly four times as many trainable weights per hidden size as a standard RNN and is still prone to exploding gradients. In many cases, gradient clipping is still required.

GRU

The GRU is similar to an LSTM, but is designed to have fewer trainable variables per hidden unit. This is done by incorporating the input gate and forget gate into a single gate, called the update gate, z_t . Instead of an output gate, the network incorporates a reset gate, r_t . Similar to an LSTM, each gate uses a sigmoid nonlinearity, $\sigma(\cdot)$. The equations for the basic architecture are given below.

$$z_t = \sigma(U^z x_t + W^z h_{t-1} + b^z) \quad (2.24)$$

$$r_t = \sigma(U^r x_t + W^r h_{t-1} + b^r) \quad (2.25)$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tanh(U x_t + W (r_t \odot h_{t-1})) \quad (2.26)$$

where $\mathbf{1}$ is a vector consisting of all ones. In practice, the LSTM is more common than the GRU.

2.4 Orthogonal/Unitary Recurrent Neural Networks

Recently there has been a surge in RNNs that maintain a strict orthogonal or unitary recurrent weight matrix. An orthogonal matrix is a matrix $W \in \mathbb{R}^{m \times m}$ such that $W^T W = W W^T = I$ and a unitary matrix is a matrix $W \in \mathbb{C}^{m \times m}$ such that $W^* W = W W^* = I$ where $*$ is the conjugate transpose. A desirable property for these matrices is that

$$\|W\|_2 = 1 \quad (2.27)$$

Thus we can rewrite (2.14) as

$$\left\| \frac{\partial \mathcal{L}}{\partial h_t} \right\|_2 \leq \left(\prod_{k=t+1}^{\tau} \|D_k\|_2 \right) \left\| \frac{\partial \mathcal{L}}{\partial h_\tau} \right\|_2 \quad (2.28)$$

Using the same assumption on D_k that there is at least one nonzero activation value and the nonlinearity is ReLU, (2.28) becomes

$$\left\| \frac{\partial \mathcal{L}}{\partial h_t} \right\|_2 \leq \left\| \frac{\partial \mathcal{L}}{\partial h_\tau} \right\|_2 \quad (2.29)$$

Although this doesn't necessarily guarantee the avoidance of vanishing/exploding gradients, it can help mitigate the possibility of such occurrences by eliminating the negative effects of the repeated multiplication of the recurrent weight matrix shown in (2.12). In this section, we explore several different architectures that either maintain an orthogonal or unitary recurrent weight matrix.

Unitary Recurrent Neural Networks

For networks that maintain a unitary recurrent weight matrix, $W \in \mathbb{C}^{m \times m}$, additional architecture complexity is required due to implementation issues. The first issue is the need to multiply a complex valued matrix and a complex valued vector together since most machine learning programs are currently not designed to handle complex valued entries. As shown in [48], many unitary RNNs overcome this issue by splitting complex matrices and vectors into the real and imaginary components. For instance, given $W = R + iC$ where $R, C \in \mathbb{R}^{m \times m}$ and $h_t = a_t + ib_t$ where $a_t, b_t \in \mathbb{R}^m$ we have $Wh_t = (Ra_t - Cb_t) + i(Rb_t + Ca_t)$. This operation can be rewritten as:

$$Wh_t = \begin{bmatrix} \text{Re}(Wh_t) \\ \text{Im}(Wh_t) \end{bmatrix} = \begin{bmatrix} R & -C \\ C & R \end{bmatrix} \begin{bmatrix} \text{Re}(h_t) \\ \text{Im}(h_t) \end{bmatrix} \quad (2.30)$$

A second issue is the selection of a nonlinear activation function. Since activation functions are generally real-valued functions, it is not clear how they should be applied to complex entries that are split into real and complex components. One approach is to apply the function to the real and complex components separately, but as discussed in [48], this breaks the relationship between each component and negatively affects training of the network. In many unitary RNNs, the preferred nonlinearity is the modReLU [48].

modReLU

The right selection of a nonlinear activation function plays a major role in avoiding the vanishing and exploding gradient problem. Inspired by the ReLU function, see Section 1.4, the **modReLU** was developed by [1] to handle complex-valued entries. The modReLU is quite popular in unitary RNNs and is used in architectures by

[21, 23, 48, 49] and has been analyzed in [45]. The modReLU is defined below:

$$\sigma_{\text{modReLU}}(z) = \begin{cases} (|z| + b) \frac{z}{|z|} & \text{if } |z| + b \geq 0 \\ 0 & \text{if } |z| + b < 0 \end{cases} \quad (2.31)$$

$$= \frac{z}{|z|} \sigma_{\text{ReLU}}(|z| + b), \quad (2.32)$$

Here b denotes a trainable bias and $z \in \mathbb{C}$. It should be noted that the modReLU can also be extended for use in the real case with (2.32) becoming

$$\sigma_{\text{modReLU}}(x) = \text{sgn}(x) \sigma_{\text{ReLU}}(|x| + b) \quad (2.33)$$

where $\text{sgn}(\cdot)$ is the sign operator and $x \in \mathbb{R}$. In this case, the modReLU behaves similar to the ReLU function with the desirable property that the norm of the Jacobian matrix will typically be one. Exceptions to this can occur in the case when $|x| < |b|$ for $b < 0$ and $x = 0$ for $b > 0$, see Figure 2.3 for details. It is also not differentiable at the points where $|x| = |b|$ for $b < 0$ and $x = 0$ for $b > 0$. In practice, these corner cases are rare and experimentally do not occur. They can also be avoided by numerically setting the derivative equal to one or zero at these points. Finally, the modReLU has the advantage over the ReLU function in that it can be nonzero for both positive and negative activation values.

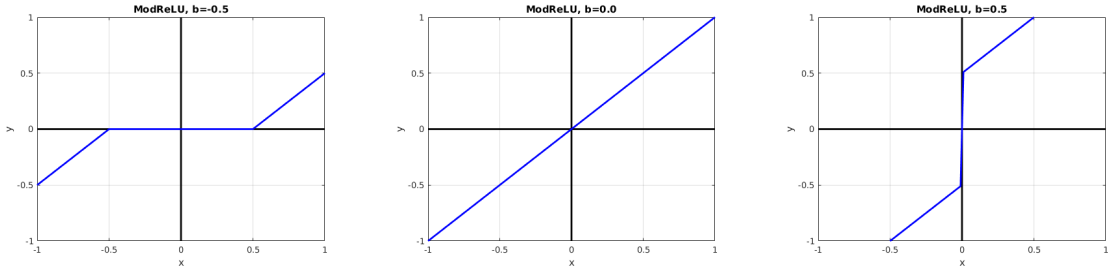


Figure 2.3: Plots of the modReLU activation function using real input, x , and $b=-0.5$ (left), $b=0$ (center), and $b=0.5$ (right).

In the complex case, the modReLU activation function in (2.32) does exhibit numerical instability. As noted in [33], the modReLU activation function has a discontinuity at $z = 0$ whenever $b > 0$ which results in the derivative having a singularity at this point. Using Wirtinger derivatives as defined in Section 1.6, the derivative of the modReLU is given in Theorem 2.4.1.

Theorem 2.4.1. *The Wirtinger derivatives of the $\sigma_{\text{modReLU}}(z)$ are given by*

$$\frac{\partial \sigma_{\text{modReLU}}(z)}{\partial z} = \begin{cases} 1 + \frac{b}{2|z|} & \text{if } |z| + b \geq 0 \\ 0 & \text{if } |z| + b < 0, \end{cases}$$

$$\frac{\partial \sigma_{\text{modReLU}}(z)}{\partial \bar{z}} = \begin{cases} \frac{1}{2} \left[\frac{-bz^2}{|z|^3} \right] & \text{if } |z| + b \geq 0 \\ 0 & \text{if } |z| + b < 0. \end{cases}$$

In particular, if $b > 0$, then $\frac{\partial \sigma_{\text{modReLU}}(z)}{\partial z}$ and $\frac{\partial \sigma_{\text{modReLU}}(z)}{\partial \bar{z}}$ tend to infinity as $z \mapsto 0$.

In numerical simulations, if $|z| \ll b$, then $|\frac{\partial \sigma_{\text{modReLU}}(z)}{\partial \bar{z}}|$ is extremely large and will cause floating point exceptions such as NaN during training. If $b \leq 0$, then the derivatives are well defined and bounded for all z and floating point exceptions do not occur.

In order to avoid division by zero during the forward and backward pass of the network, [1] and [48] implement an approximate modReLU of the form:

$$\sigma_\epsilon(z) = \frac{z}{\hat{z} + \epsilon} \sigma_{\text{ReLU}}(\hat{z} + b) \quad (2.34)$$

where $\epsilon = 10^{-5}$ and $\hat{z} = \sqrt{x^2 + y^2 + \epsilon}$ with $z = x + iy$. Unfortunately, the approximate modReLU still can have unbounded derivatives when $\hat{z} + \epsilon \ll b$ as indicated in Theorem 2.4.2 and Figure 2.4.

Theorem 2.4.2. *The Wirtinger derivatives of the approximate modReLU activation function are:*

$$\begin{aligned} \frac{\partial \sigma_\epsilon(z)}{\partial z} &= \begin{cases} \frac{\hat{z}+b}{\hat{z}+\epsilon} + \frac{|z|^2(\epsilon-b)}{2\hat{z}(\hat{z}+\epsilon)^2} & \text{if } \hat{z} + b \geq 0 \\ 0 & \text{if } \hat{z} + b < 0, \end{cases} \\ \frac{\partial \sigma_\epsilon(z)}{\partial \bar{z}} &= \begin{cases} \frac{z^2(\epsilon-b)}{2\hat{z}(\hat{z}+\epsilon)^2} & \text{if } \hat{z} + b \geq 0 \\ 0 & \text{if } \hat{z} + b < 0. \end{cases} \end{aligned}$$

Proof. First we note the following:

- $\frac{\partial \hat{z}}{\partial x} = \frac{x}{\hat{z}}$
- $\frac{\partial \hat{z}}{\partial y} = \frac{y}{\hat{z}}$
- $\frac{\partial \sigma_\epsilon(z)}{\partial x} = \frac{(\hat{z}+\epsilon)(\hat{z}+b) - \frac{xz(\hat{z}+b)}{\hat{z}}}{(\hat{z}+\epsilon)^2} + \frac{xz}{\hat{z}(\hat{z}+\epsilon)}$ if $\hat{z} + b \geq 0$
- $\frac{i(\hat{z}+\epsilon)(\hat{z}+b) - \frac{yz(\hat{z}+b)}{\hat{z}}}{(\hat{z}+\epsilon)^2} + \frac{yz}{\hat{z}(\hat{z}+\epsilon)}$ if $\hat{z} + b \geq 0$

Thus we have

$$\begin{aligned} \frac{\partial \sigma_\epsilon(z)}{\partial z} &= \frac{1}{2} \left[\frac{\partial \sigma_\epsilon(z)}{\partial x} - i \frac{\partial \sigma_\epsilon(z)}{\partial y} \right] \\ &= \frac{1}{2} \left[\frac{2(\hat{z} + \epsilon)(\hat{z} + b) - \frac{z(\hat{z}+b)}{\hat{z}}(x - iy)}{(\hat{z} + \epsilon)^2} + \frac{z}{\hat{z}(\hat{z} + \epsilon)}(x - iy) \right] \\ &= \frac{\hat{z} + b}{\hat{z} + \epsilon} - \frac{|z|^2(\hat{z} + b)}{2\hat{z}(\hat{z} + \epsilon)^2} + \frac{|z|^2}{2\hat{z}(\hat{z} + \epsilon)^2} \\ &= \frac{\hat{z} + b}{\hat{z} + \epsilon} + \frac{|z|^2(\epsilon - b)}{2\hat{z}(\hat{z} + \epsilon)^2} \end{aligned}$$

and

$$\begin{aligned}
 \frac{\partial \sigma_\epsilon(z)}{\partial z} &= \frac{1}{2} \left[\frac{\partial \sigma_\epsilon(z)}{\partial x} + i \frac{\partial \sigma_\epsilon(z)}{\partial y} \right] \\
 &= \frac{1}{2} \left[\frac{z(\hat{z} + b)(-x - iy)}{\hat{z}(\hat{z} + \epsilon)^2} + \frac{z(x + iy)}{\hat{z}(\hat{z} + \epsilon)} \right] \\
 &= \frac{1}{2} \left[\frac{-z^2(\hat{z} + b)}{\hat{z}(\hat{z} + \epsilon)^2} + \frac{z^2}{\hat{z}(\hat{z} + \epsilon)} \right] \\
 &= \frac{z^2(\epsilon - b)}{2\hat{z}(\hat{z} + \epsilon)^2}
 \end{aligned}$$

□

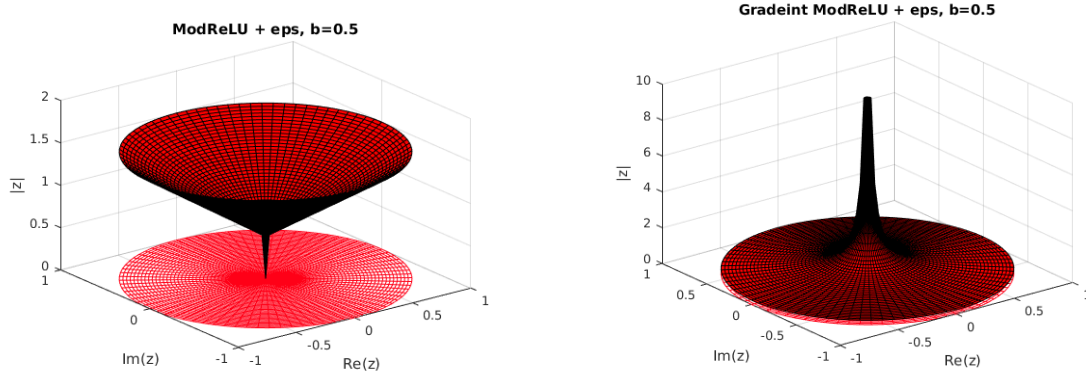


Figure 2.4: Surface plots of the modulus of the approximate modReLU activation function σ_ϵ (Left) and the modulus of the gradient of σ_ϵ with respect to \bar{z} (Right). Both plots use a bias of $b=0.5$.

To mitigate the issue of unbounded gradients when using modReLU, the initialization of the vector bias, b , and the hidden state, h_0 , becomes important. As noted in [33], for the MNIST experiment, see section 2.5, if the initial state is set to $h_0 = 0$ and non-trainable and b is initialized from $\mathcal{U}[-0.01, 0.01]$, gradient values can become NaN before the end of the first training epoch. These large gradients occur because some initial entries of b will be positive and can be much larger in magnitude than the corresponding entries of \hat{z} . In particular, the first several pixels of a given MNIST image will typically have zero pixel values which will cause \hat{z} to be very small. Thus for some entries, $\hat{z} + \epsilon \ll b$, and large gradients can occur or accumulate over many time-steps to cause overflow issues.

To avoid unbounded gradients, one can initialize h_0 away from zero so that entries of \hat{z} are larger with respect to b when the first several zero valued pixels are fed into the network. Another option is to constrain $b < 0$. Unfortunately, experimental results from [33] indicate that this tends to hinder performance. Finally, another option is to set h_0 to be trainable.

Unitary Evolution Recurrent Neural Network

The **unitary evolution recurrent neural network (uRNN)** was the first known RNN to maintain an orthogonal/unitary recurrent weight matrix [1]. To be consistent with [48], the uRNN will be referred to as the Restricted-Capacity uRNN in this paper. The Restricted-Capacity uRNN maintains a strict unitary recurrent weight matrix by parameterizing the matrix by a product of unitary matrices:

$$W = D_3 R_2 \mathcal{F}^{-1} D_2 \Pi R_1 \mathcal{F} D_1 \quad (2.35)$$

where each $D_i \in \mathbb{C}^{m \times m}$ is a diagonal matrix with diagonal entries $D_{j,j} = e^{iw_j}$ with $w_j \in \mathbb{R}$, each $R_i = I - 2 \frac{vv^*}{\|v\|^2} \in \mathbb{C}^{m \times m}$ is a Householder reflection matrix with $v \in \mathbb{C}^m$, the $\Pi \in \mathbb{R}^{m \times m}$ is a random fixed permutation matrix, and $\mathcal{F}, \mathcal{F}^{-1} \in \mathbb{C}^{m \times m}$ are Fourier and inverse Fourier transform matrices. Since the product of unitary matrices is also unitary, the resulting W in (2.35) is unitary. This parameterization requires the hidden state of the RNN to be complex valued and the approximate modReLU function is used, see (2.34).

A major drawback to the parameterization in (2.35) is that it can only be used to form a subset of unitary matrices in $\mathbb{C}^{m \times m}$ and is unable to represent all unitary matrices when $m > 7$ as shown in Theorem 2.4.4. To prove Theorem 2.4.4, we follow [48] and first define the Stiefel Manifold in Definition 2.4.1.

Definition 2.4.1 (Stiefel Manifold). *The set of unitary matrices is called the **Stiefel Manifold** and is denoted $\mathcal{V}_p(\mathbb{C}^n) = \{X \in \mathbb{C}^{n \times p} | X^* X = I\}$ where $*$ is the conjugate transpose.*

Note that for a single layer RNN, the hidden weight matrix is square, so we will consider the Stiefel Manifold $\mathcal{V}_n(\mathbb{C}^n)$ which is a manifold of dimension n^2 . The proof of Theorem 2.4.4 uses Sard's Theorem which is provided below.

Theorem 2.4.3 (Sard's Theorem). *Let $f : X \rightarrow Y$ be a smooth map of finite dimensional manifolds where $\dim(X) = q$ and $\dim(Y) = r$ and $C = \{x \in X : \text{rank}(J_f(x)) < r\}$ denote the set of critical points of f where J_f is the Jacobian of f . Then $f(C)$ has Lebesgue measure zero in Y .*

Theorem 2.4.4 (Wisdom et. al [48]). *The set of unitary matrices parameterized by $7n$ real-valued parameters is a proper subset of $\mathcal{V}_n(\mathbb{C}^n)$ for $7n < n^2$.*

Proof. Let g be the smooth map $g : \mathbb{R}^{7n} \rightarrow \mathcal{V}_n(\mathbb{C}^n)$ where $g(x) = W$ and W can be defined as in (2.35). We have $\text{rank}(J_{g(x)}) \leq 7n < n^2$. By Sard's Theorem, the set of unitary matrices formed by the Restricted-Capacity uRNN transformation has measure zero in $\mathcal{V}_n(\mathbb{C}^n)$ and so is a proper subset. \square

Now the parameterization in (2.35) requires $4m$ trainable parameters for the reflection matrices, since the vectors v are complex valued, and $3m$ trainable parameters for the diagonal matrices for a total of $7m$ trainable parameters. Thus when $7m < m^2$ or $m > 7$, the map from this parameterization to the manifold of unitary matrices will not be onto by Theorem 2.4.4 and so will only be able to parameterize a subset of unitary matrices.

Full-Capacity Unitary Recurrent Neural Network

The **Full-Capacity Unitary Recurrent Neural Network (Full-Capacity uRNN)** by [48] takes a slightly different approach than the Restricted-Capacity uRNN to maintain a unitary recurrent weight matrix. Instead of using a fixed parameterization, the Full-Capacity uRNN performs gradient descent along the manifold of unitary matrices or Stiefel manifold. The initial unitary recurrent weight matrix is initialized using (2.35). Since there is no restriction on what direction gradient descent will take along the manifold, the network is not constrained to a subset of unitary matrices but can optimize in any direction that minimizes the given loss function, $L := L(W)$. Note that we consider the loss function as a function of the recurrent weight W for the purpose of computing derivatives.

Using results from [44] and [46], the Full-Capacity uRNN follows the descent direction BW where

$$B = \frac{\partial L}{\partial W} W^* - W \left(\frac{\partial L}{\partial W} \right)^*.$$

Here BW is a representation of the derivative operator $DL(W)$ in the tangent space of the Stiefel manifold at W . The gradient direction given by BW is then reflected back down to the Stiefel manifold using a **Caley Transform**, see Definition 2.4.2.

Definition 2.4.2 (Caley Transform). *Given a square matrix $A \in \mathbb{C}^{n \times n}$ that does not have any -1 eigenvalues and $I \in \mathbb{R}^{n \times n}$ the identity matrix, the Caley transform is given by:*

$$W = (I + A)^{-1} (I - A) \tag{2.36}$$

The update of the recurrent matrix is thus given by:

$$W^{(k+1)} = \left(I + \frac{\lambda}{2} B^{(k)} \right)^{-1} \left(I - \frac{\lambda}{2} B^{(k)} \right) W^{(k)} \tag{2.37}$$

where λ is the learning rate and k is the current update step.

Unfortunately, the given update step in (2.37) is based on [44] and [46] which only pertains to the orthogonal case and not the unitary case. Since no proof is given that the proposed update step in (2.37) is indeed along the unitary Stiefel Manifold, we expand upon the results from [44] and [46] to the unitary case by showing that the update step is indeed along the Stiefel Manifold in Theorem 2.4.5. We also show it is a gradient descent direction in Theorem 2.4.6.

Theorem 2.4.5. *Given any skew-Hermitian matrix $A \in \mathbb{C}^{n \times n}$ and unitary matrix $W \in \mathbb{C}^{n \times n}$, the curve $Y(\lambda) = \left(I + \frac{\lambda}{2} A \right)^{-1} \left(I - \frac{\lambda}{2} A \right) W$ lies in the Stiefel manifold for all $\lambda \in \mathbb{R}$.*

Proof. Since A is skew-Hermitian, the eigenvalues of A are either purely imaginary or zero so the eigenvalues of $I \pm \frac{\lambda}{2} A$ are non-zero and $I \pm \frac{\lambda}{2} A$ is invertible. Calculating,

we have

$$\begin{aligned}
Y(\lambda)^*Y(\lambda) &= W^* (I - \frac{\lambda}{2}A)^* (I + \frac{\lambda}{2}A)^{-1*} (I + \frac{\lambda}{2}A)^{-1} (I - \frac{\lambda}{2}A) W \\
&= W^* (I - \frac{\lambda}{2}A)^* [(I + \frac{\lambda}{2}A) (I + \frac{\lambda}{2}A)^*]^{-1} (I - \frac{\lambda}{2}A) W \\
&= W^* (I + \frac{\lambda}{2}A) [(I + \frac{\lambda}{2}A) (I - \frac{\lambda}{2}A)]^{-1} (I - \frac{\lambda}{2}A) W \\
&= W^* (I + \frac{\lambda}{2}A) [(I - \frac{\lambda}{2}A) (I + \frac{\lambda}{2}A)]^{-1} (I - \frac{\lambda}{2}A) W \\
&= W^* (I + \frac{\lambda}{2}A) (I + \frac{\lambda}{2}A)^{-1} (I - \frac{\lambda}{2}A)^{-1} (I - \frac{\lambda}{2}A) W \\
&= I
\end{aligned}$$

Similarly,

$$\begin{aligned}
Y(\lambda)Y(\lambda)^* &= (I + \frac{\lambda}{2}A)^{-1} (I - \frac{\lambda}{2}A) WW^* (I - \frac{\lambda}{2}A)^* [(I + \frac{\lambda}{2}A)^{-1}]^* \\
&= (I + \frac{\lambda}{2}A)^{-1} (I - \frac{\lambda}{2}A) (I + \frac{\lambda}{2}A) [(I + \frac{\lambda}{2}A)^*]^{-1} \\
&= (I + \frac{\lambda}{2}A)^{-1} (I + \frac{\lambda}{2}A) (I - \frac{\lambda}{2}A) (I - \frac{\lambda}{2}A)^{-1} \\
&= I \quad \square
\end{aligned}$$

□

Theorem 2.4.6. *If $A = GW^* - WG^*$, where $G = \left[\frac{\partial L}{\partial W_{i,j}} \right]_{i,j=1}^n$ and $W \in \mathbb{C}^{n \times n}$ is unitary, then $Y(\lambda) = (I + \frac{\lambda}{2}A)^{-1} (I - \frac{\lambda}{2}A) W$ with $\lambda \in \mathbb{R}$ is a descent curve where the derivative of $L(Y(\lambda))$ with respect to λ for some loss function L satisfies:*

$$L'(Y(0)) = -\|A\|_F^2$$

Proof. By the chain rule,

$$\mathcal{L}'(Y(\lambda)) = \text{tr} \left(G^* Y'(\lambda) + G^T \overline{Y'(\lambda)} \right) \quad (2.38)$$

Note: Since L is not holomorphic, Wirtinger Calculus is needed to compute (2.38), see Section 1.6. We can rewrite $Y(\lambda) = W - \frac{\lambda}{2}A(W + Y(\lambda))$, and take the derivative with respect to λ and rearrange to get:

$$Y'(\lambda) = -\frac{1}{2} \left(I + \frac{\lambda}{2}A \right)^{-1} A (W + Y(\lambda)) \quad (2.39)$$

Now plugging (2.39) into (2.38) for $\lambda = 0$ we get:

$$L'(Y(0)) = -\text{tr} \left(G^* AW + G^T \overline{AW} \right) \quad (2.40)$$

By the invariance of the trace with respect to the transpose, we take the transpose of the second term to obtain:

$$L'(Y(0)) = -\text{tr} (G^* AW + W^* A^* G) \quad (2.41)$$

Substituting the definition of A and multiplying out:

$$L'(Y(0)) = -\text{tr}(G^*GW^*W - G^*WG^*W + W^*WG^*G - W^*GW^*G) \quad (2.42)$$

Using the cyclic property of traces:

$$L'(Y(0)) = -\text{tr}(GW^*WG^* - WG^*WG^* + WG^*GW^* - GW^*GW^*) \quad (2.43)$$

$$= -\text{tr}[(GW^* - WG^*)(WG^* - GW^*)] \quad (2.44)$$

$$= -\text{tr}(AA^*) \quad (2.45)$$

$$= -\|A\|_F^2 \quad \square \quad (2.46)$$

□

We should note that the Full-Capacity uRNN has two potential limitations that may negatively affect performance. The first is that the descent curve in (2.37) has been shown to guarantee a descent direction, but not necessarily the steepest descent direction. The second limitation is due to the repeat matrix multiplication required when implementing (2.37) iteratively. Due to the accumulation of rounding errors over a large number of multiplications, the recurrent weight matrix may not remain unitary throughout training.

Tunable Efficient Unitary Recurrent Neural Network

The **Tunable Efficient Unitary Recurrent Neural Network (EURNN)** [23] is designed to maintain a unitary recurrent weight matrix by using a long product of Givens rotation matrices and a diagonal matrix. In particular, any unitary matrix $W \in \mathbb{C}^{m \times m}$ can be parameterized as follows:

$$W = D \prod_{i=2}^m \prod_{j=1}^{i-1} R_{i,j} \quad (2.47)$$

where each $R_{i,j}$ is the identity matrix with the entries $R_{i,i}$, $R_{i,j}$, $R_{j,i}$, and $R_{j,j}$ consisting of:

$$\begin{bmatrix} R_{i,i} & R_{i,j} \\ R_{j,i} & R_{j,j} \end{bmatrix} = \begin{bmatrix} e^{i\phi_{i,j}} \cos \theta_{i,j} & -e^{i\phi_{i,j}} \sin \theta_{i,j} \\ \sin \theta_{i,j} & \cos \theta_{i,j} \end{bmatrix} \quad (2.48)$$

and D is a diagonal matrix of entries of the form $e^{iw_{i,j}}$. This parameterization requires $\frac{m(m-1)}{2} - 1$ matrix multiplications and a total of m^2 parameters. In order to reduce the large number of required matrix multiplications, the unitary recurrent weight matrix is represented by grouping various rotation matrices together as follows:

$$\begin{aligned} W &= D \left(R_{1,2}^{(1)} R_{3,4}^{(1)} \dots R_{m/2-1, m/2}^{(1)} \right) \left(R_{2,3}^{(2)} R_{4,5}^{(2)} \dots R_{m/2-2, m/2-1}^{(2)} \right) \dots \\ &:= DF_a^{(1)} F_b^{(2)} \dots F_b^{(L)} \end{aligned} \quad (2.49)$$

When $L = m$, this parameterization is able to represent any unitary matrix. In practice, $L < m$ which reduces the number of matrix multiplications but restricts the unitary matrix to a subset of unitary matrices. This version of implementation of the EURNN is called the tunable space implementation.

To further reduce computational costs, the EURNN can be implemented using a FFT-style implementation using the following parameterization:

$$W = DF_1F_2\dots F_{\log(m)} \quad (2.50)$$

Here each F_i consists of a subset of rotation matrices. In particular, rotation matrices with coordinate indices in the interval $(2pk + j, p(2k + 1) + j)$ where $p = \frac{m}{2^i}$, $k \in \{0, 1, \dots, 2^{i-1}\}$, and $j \in \{1, 2, \dots, p\}$. This parameterization requires a total of $m \log(m)/2$ rotation matrices and provides an approximation of any unitary matrix.

Finally, to reduce the computational overhead even further, the EURNN does not explicitly form the matrices used in (2.49) and (2.50). Instead the EURNN replaces required matrix multiplications with vector element-wise multiplications and rotations. The EURNN also uses the modReLU nonlinearity.

Efficient Orthogonal Parameterization Recurrent Neural Network

Unlike the Restricted-Capacity uRNN, Full-Capacity uRNN, and EURNN, the **Efficient Orthogonal Parameterization Recurrent Neural Network (oRNN)** [35] does not maintain a unitary recurrent weight matrix but an orthogonal recurrent weight matrix by using a long product of Householder reflection matrices. Let $W \in \mathbb{R}^{m \times m}$ be the orthogonal recurrent weight matrix and $H_k \in \mathbb{R}^{m \times m}$ a Householder reflection matrix of the form:

$$H_k = \begin{bmatrix} I_{m-k} & 0 \\ 0 & I_k - 2 \frac{u_k u_k^T}{\|u_k\|_2^2} \end{bmatrix} \quad (2.51)$$

where $u_k \in \mathbb{R}^k$, $I_j \in \mathbb{R}^{j \times j}$ is the identity matrix, and $1 < k \leq m$. For $k = 1$, we define

$$H_1 = \begin{bmatrix} I_{m-1} & \\ & u_1 \end{bmatrix} \quad (2.52)$$

with $u_1 \in \mathbb{R}$. Using these matrices, the orthogonal matrix is parameterized by

$$W = H_m H_{m-1} \dots H_{m-l+1} \quad (2.53)$$

In the case of $l = m$, this parameterization can represent any orthogonal matrix as long as $u_1 \in \{-1, 1\}$. When $l < m$, the recurrent weight matrix is restricted to a subset of possible orthogonal matrices.

For implementation purposes, the oRNN is typically restricted to the case of $l < m$. The nonlinearity activation used is the Leaky-ReLU.

2.5 Scaled Cayley Orthogonal Recurrent Neural Network

In the previous sections, several models were presented that maintain either an orthogonal or unitary recurrent weight matrix to address the vanishing/exploding gradient issue. It was shown that the parameterization used in the Restricted-Capacity uRNN is unable to represent all unitary matrices. The EURNN and oRNN are able to parameterize any unitary/orthogonal matrix but require a long product of matrix multiplications. To reduce the number of matrix multiplications, the EURNN and oRNN are implemented by restricting the number of matrix multiplications which prevents these models from parameterizing any arbitrary unitary/orthogonal weight matrix. To further increase efficiency, the EURNN uses many complicated simplifications that pose some difficulty in implementation. Finally, the Full-Capacity uRNN does not require a long product of matrices but updates the unitary recurrent weight matrix along the Stiefel manifold using a direction that may not necessarily be the steepest descent direction. The update step also involves a multiplicative update that can result in a recurrent matrix that is no longer unitary.

To eliminate the need of a long product of matrices, complicated implementation schemes, and a potential loss of orthogonality, the **scaled Cayley orthogonal Recurrent Neural Network (scoRNN)** was developed by [14]. The scoRNN parameterizes an orthogonal recurrent weight matrix by a skew-symmetric matrix through a scaled Cayley transform.

$$W = (I + A)^{-1} (I - A) D \quad (2.54)$$

where $W \in \mathbb{R}^{n \times n}$ is orthogonal, $A \in \mathbb{R}^{n \times n}$ is skew-symmetric, and $D \in \mathbb{R}^{n \times n}$ is a diagonal matrix with entries consisting of ± 1 .

The idea behind (2.54) is motivated by the well known observation that the Cayley transform forms a bijection between the set of all orthogonal matrices without -1 eigenvalues and the set of all skew-symmetric matrices as formalized in Theorem 2.5.1.

Theorem 2.5.1. *Let \mathcal{W} be the set of all orthogonal matrices without -1 eigenvalues and \mathcal{A} the set of all skew-symmetric matrices. Then the mapping $\mathcal{K} : \mathcal{A} \rightarrow \mathcal{W}$ defined by: $\mathcal{K}(A) = (I + A)^{-1} (I - A)$ forms a bijection.*

Proof. From the proof of Theorem 2.4.5, it follows that $\mathcal{K}(A)$ is indeed orthogonal. To show that the mapping is injective, assume $A, B \in \mathcal{A}$ such that $\mathcal{K}(A) = \mathcal{K}(B)$. It follows that:

$$\begin{aligned} (I + A)^{-1} (I - A) &= (I + B)^{-1} (I - B) \\ &= (I + B)^{-1} (I - B) (I + B) (I + B)^{-1} \\ &= (I + B)^{-1} (I + B) (I - B) (I + B)^{-1} \\ &= (I - B) (I + B)^{-1} \end{aligned} \quad (2.55)$$

Now rearranging (2.55),

$$\begin{aligned}
(I - A)(I + B) &= (I + A)(I - B) \\
I + B - A - AB &= I + A - B - AB \\
B &= A
\end{aligned}$$

as desired. To show surjectivity, let $W \in \mathcal{W}$ and define $A := (I + W)^{-1}(I - W)$. We note that $A \in \mathcal{A}$ as shown:

$$\begin{aligned}
A^T &= (I - W)^T [(I + W)^{-1}]^T \\
&= (I - W^T) (I + W^T)^{-1} \\
&= (WW^T - W^T) (WW^T + W^T)^{-1} \\
&= (W - I) W^T W (W + I)^{-1} \\
&= -(I - W) (I + W)^{-1} \\
&= -(I + W)^{-1} (I - W) \\
&= -A
\end{aligned}$$

Using the definition of A and multiplying both sides by $(I + W)$, it directly follows

$$\begin{aligned}
A(I + W) &= I - W \\
A + AW &= I - W \\
AW + W &= I - A \\
(I + A)W &= I - A \\
W &= (I + A)^{-1}(I - A)
\end{aligned}$$

Here it should be noted that $I + A$ is invertible since A is skew-symmetric and so only has zero or purely complex eigenvalues. To see this, suppose $\lambda \in \mathbb{C}$ is an eigenvalue of A with associated eigenvector $v \in \mathbb{C}^n$ that is normalized so that $v^*v = 1$. Thus

$$\begin{aligned}
Av &= \lambda v \\
v^*Av &= \lambda
\end{aligned} \tag{2.56}$$

Similarly, taking the complex conjugate of (2.56),

$$\begin{aligned}
v^*A^T &= \bar{\lambda}v^* \\
-v^*A &= \bar{\lambda}v^* \\
v^*Av &= -\bar{\lambda}
\end{aligned} \tag{2.57}$$

Combining (2.56) and (2.57) we have $\lambda = -\bar{\lambda}$ which holds if λ is either zero or purely imaginary.

□

Parameterizing an orthogonal recurrent weight matrix by a skew-symmetric matrix through the Cayley transform is attractive because it requires only $\frac{n(n-1)}{2}$ trainable parameters to represent the skew-symmetric matrix as opposed to n^2 for a complete orthogonal matrix and because skew-symmetric matrices are closed under addition or subtraction. Thus gradient descent algorithms like RMSProp and Adam, see Section 1.7, will maintain a skew-symmetric matrix after taking an update step.

Unfortunately, a major drawback to this parameterization is that it cannot represent orthogonal matrices with -1 eigenvalues because the resulting $I + W$ term will not be invertible. Although these matrices can be theoretically approximated by a matrix with eigenvalues arbitrarily close to -1 , in practice it can be a problem. In particular, consider a 2x2 orthogonal matrix W with eigenvalues $\approx -0.99999 \pm 0.00447i$. Such a matrix and the associated skew-symmetric matrix are given below:

$$W = \begin{bmatrix} -0.99999 & -\sqrt{1 - 0.99999^2} \\ \sqrt{1 - 0.99999^2} & -0.99999 \end{bmatrix} \quad A \approx \begin{bmatrix} 0 & 447.212 \\ -447.212 & 0 \end{bmatrix}$$

With A having such large entries, it will take a long time for gradient descent algorithms to reach such a matrix, if at all. Fortunately, work by [39] and [24] show that scaling the Cayley transform by a diagonal matrix with appropriate entries, orthogonal matrices with -1 eigenvalues can be parameterized. See Theorem 2.5.2 for the formal results.

Theorem 2.5.2. *Every orthogonal matrix W can be expressed as*

$$W = (I + A)^{-1}(I - A)D$$

where $A = [a_{ij}]$ is real-valued, skew-symmetric with $|a_{ij}| \leq 1$, and D is diagonal with all nonzero entries equal to ± 1 . Similarly, every unitary matrix U can be expressed as

$$U = (I + S)^{-1}(I - S)\Phi$$

where $S = [s_{ij}]$ is skew-Hermitian with $|s_{i,j}| \leq 1$ and Φ is a unitary diagonal matrix.

Proof. See [39] for the orthogonal case, and [24] for the unitary case. □

The transform in Theorem 2.5.2 will be referred to as the **scaled Cayley transform**. It should be noted that these results also indicate that a skew-symmetric matrix with bounded entries can be used in the scaled Cayley transform.

Architecture Details

The scoRNN model operates similarly to a standard RNN, see (2.1), (2.2), and (2.3), except the modReLU activation function is used, see (2.33), and the recurrent weight matrix W is parameterized through the scaled Cayley Transform. In particular, (2.1) and (2.2) become

$$z_i = Ux_i + Wh_{i-1} \tag{2.58}$$

$$h_i = \text{sgn}(z_i)\sigma_{\text{ReLU}}(|z_i| + b) \tag{2.59}$$

respectively where $W = (I + A)^{-1} (I - A) D$.

During training, D is kept fixed with the number of -1 s considered a hyperparameter which we denote as ρ . In order to update A , the gradients must pass through the scaled Cayley transform which are given in Theorem 2.5.3. All other trainable parameters are updated using gradient descent or a related algorithm. In particular, the standard backpropagation algorithm is first used to compute $\frac{\partial L}{\partial W}$ and to update all other trainable weights. The skew-symmetric matrix A is then updated using the gradients computed in Theorem 2.5.3 with a gradient descent optimization method, and W is reconstructed as follows:

$$A^{(k+1)} = A^{(k)} - \lambda \frac{\partial L(W(A^{(k)}))}{\partial A^{(k)}}$$

$$W^{(k+1)} = (I + A^{(k+1)})^{-1} (I - A^{(k+1)}) D$$

It should be noted that the skew-symmetry of $\frac{\partial L}{\partial A}$ ensures that $A^{(k+1)}$ will be skew-symmetric and, in turn, $W^{(k+1)}$ will be orthogonal to the order of machine precision. The scoRNN model also maintains stable hidden state gradients in the sense that the gradient norm does not change significantly in time. The scoRNN model also has small overhead computational costs over the standard RNN.

Similar to the EURNN and oRNN, scoRNN can be implemented by restricting the A matrix to a banded skew-symmetric matrix with bandwidth ℓ to reduce the number of trainable parameters. Although this may work well for particular tasks, such a modification introduces an additional hyperparameter and reduces representational capacity for the recurrent weight matrix without any reduction in the dimension of the hidden state.

Theorem 2.5.3. *Let $L = L(W) : \mathbb{R}^{n \times n} \rightarrow \mathbb{R}$ be some differentiable loss function for an RNN with the recurrent weight matrix W . Let $W = W(A) := (I + A)^{-1} (I - A) D$ where $A \in \mathbb{R}^{n \times n}$ is skew-symmetric and $D \in \mathbb{R}^{n \times n}$ is a fixed diagonal matrix consisting of -1 and 1 entries. Then the gradient of $L = L(W(A))$ with respect to A is*

$$\frac{\partial L}{\partial A} = V^T - V \tag{2.60}$$

where $V := (I + A)^{-T} \frac{\partial L}{\partial W} (D + W^T)$, $\frac{\partial L}{\partial A} = \left[\frac{\partial L}{\partial A_{i,j}} \right] \in \mathbb{R}^{n \times n}$, and $\frac{\partial L}{\partial W} = \left[\frac{\partial L}{\partial W_{i,j}} \right] \in \mathbb{R}^{n \times n}$.

Proof. Let $Z := (I + A)^{-1} (I - A)$. We consider the (i, j) entry of $\frac{\partial L}{\partial A}$. Taking the derivative with respect to $A_{i,j}$ where $i \neq j$ we obtain:

$$\begin{aligned} \frac{\partial L}{\partial A_{i,j}} &= \sum_{k,l=1}^n \frac{\partial L}{\partial W_{k,l}} \frac{\partial W_{k,l}}{\partial A_{i,j}} = \sum_{k,l=1}^n \frac{\partial L}{\partial W_{k,l}} D_{l,l} \frac{\partial Z_{k,l}}{\partial A_{i,j}} \\ &= \text{tr} \left[\left(\frac{\partial L}{\partial W} D \right)^T \frac{\partial Z}{\partial A_{i,j}} \right] \end{aligned}$$

Using the identity $(I + A) Z = I - A$ and taking the derivative with respect to $A_{i,j}$ to both sides we obtain:

$$\frac{\partial Z}{\partial A_{i,j}} + \frac{\partial A}{\partial A_{i,j}} Z + A \frac{\partial Z}{\partial A_{i,j}} = -\frac{\partial A}{\partial A_{i,j}}$$

and rearranging we get:

$$\frac{\partial Z}{\partial A_{i,j}} = -(I + A)^{-1} \left(\frac{\partial A}{\partial A_{i,j}} + \frac{\partial A}{\partial A_{i,j}} Z \right)$$

Let $E_{i,j}$ denote the matrix whose (i, j) entry is 1 with all others being 0. Since A is skew-symmetric, we have $\frac{\partial A}{\partial A_{i,j}} = E_{i,j} - E_{j,i}$. Combining everything, we have:

$$\frac{\partial L}{\partial A_{i,j}} = -\text{tr} \left[\left(\frac{\partial L}{\partial W} D \right)^T (I + A)^{-1} (E_{i,j} - E_{j,i} + E_{i,j} Z - E_{j,i} Z) \right] \quad (2.61)$$

$$= -\text{tr} \left[\left(\frac{\partial L}{\partial W} D \right)^T (I + A)^{-1} E_{i,j} \right] + \text{tr} \left[\left(\frac{\partial L}{\partial W} D \right)^T (I + A)^{-1} E_{j,i} \right] \quad (2.62)$$

$$- \text{tr} \left[\left(\frac{\partial L}{\partial W} D \right)^T (I + A)^{-1} E_{i,j} Z \right] + \text{tr} \left[\left(\frac{\partial L}{\partial W} D \right)^T (I + A)^{-1} E_{j,i} Z \right] \quad (2.63)$$

$$= - \left[\left(\left(\frac{\partial L}{\partial W} D \right)^T (I + A)^{-1} \right)^T \right]_{i,j} + \left[\left(\frac{\partial L}{\partial W} D \right)^T (I + A)^{-1} \right]_{i,j} \quad (2.64)$$

$$- \left[\left(\left(\frac{\partial L}{\partial W} D \right)^T (I + A)^{-1} \right)^T Z^T \right]_{i,j} + \left[Z \left(\frac{\partial L}{\partial W} D \right)^T (I + A)^{-1} \right]_{i,j} \quad (2.65)$$

$$= \left[(I + Z) \left(\frac{\partial L}{\partial W} D \right)^T (I + A)^{-1} \right]_{i,j} - \left[\left(\left(\frac{\partial L}{\partial W} D \right)^T (I + A)^{-1} \right)^T (I + Z^T) \right]_{i,j} \quad (2.66)$$

$$= \left[(D + W) \left(\frac{\partial L}{\partial W} \right)^T (I + A)^{-1} \right]_{i,j} - \left[(I + A)^{-T} \frac{\partial L}{\partial W} (D + W^T) \right]_{i,j} \quad (2.67)$$

Using the above formulation, $\frac{\partial L}{\partial A_{j,j}} = 0$ and $\frac{\partial L}{\partial A_{i,j}} = -\frac{\partial L}{\partial A_{j,i}}$ so that $\frac{\partial L}{\partial A}$ is a skew-symmetric matrix. Finally, by the definition of V we get the desired result. \square

Initialization

As is common with RNNs, initialization of the trainable parameters affects the performance of scoRNN. Of particular note is the initialization of the skew-symmetric matrix A . Inspired by [15], A is initialized so that the resulting recurrent matrix W has eigenvalues distributed on the complex unit circle. This is done by initializing entries of A to be 0 except for 2x2 blocks along the diagonal.

$$A = \begin{bmatrix} B_1 & & \\ & \ddots & \\ & & B_{\lfloor n/2 \rfloor} \end{bmatrix} \quad \text{where} \quad B_j = \begin{bmatrix} 0 & s_j \\ -s_j & 0 \end{bmatrix}$$

with $s_j = \sqrt{\frac{1-\cos(t_j)}{1+\cos(t_j)}}$ and t_j is sampled uniformly from $[0, \frac{\pi}{2}]$. The Cayley transform of this matrix A will have eigenvalues of the form $\pm e^{it_j}$ for each j which will be distributed uniformly along the right unit half-circle. Multiplication by the scaling matrix D will reflect ρ of these eigenvalues across the imaginary axis.

Experiments

In this section, we perform experiments on several standard sequential tasks in order to compare the scoRNN model with a standard LSTM and other recently proposed orthogonal/unitary RNNs. For comparison purposes, single layer models were used. Code for these experiments can be found at <https://github.com/SpartinStuff/scoRNN>.

For each experiment, we found optimal hyperparameters for scoRNN using a grid search. For other models, we used the best hyperparameter settings as reported for the same testing problems, when applicable, or performed a grid search to find hyperparameters. For the LSTM, the forget gate bias was tuned over the integers -4 to 4 with it set to 1.0 unless otherwise noted.

Adding Problem

We examined a variation of the adding problem as proposed by [1] which is based on the work of [18]. This variation involves passing two sequences concurrently into the RNN, each of length T . The first sequence is a sequence of digits sampled uniformly with values ranging in a half-open interval, $\mathcal{U}[0, 1)$. The second sequence is a marker sequence consisting of all zeros except for two entries that are marked by one. The first 1 is located uniformly within the interval $[1, \frac{T}{2})$ of the sequence and the second 1 is located uniformly within the interval $[\frac{T}{2}, T)$ of the sequence. The label for each pair of sequences is the sum of the two entries that are marked by one, which forces the machine to identify relevant information in the first sequence among noise, see Figure 2.5. As the sequence length increases, it becomes more crucial to avoid vanishing/exploding gradients. Naively predicting one regardless of the sequence gives an expected mean squared error (MSE) of approximately 0.167. This will be considered as the baseline.

Sequence lengths of $T = 200, 400,$ and 750 were used in this experiment. For each sequence length, a training set size of $100,000$ and a testing set size of $10,000$ were used. The number of hidden units for each model were also adjusted so they each had approximately 15k trainable parameters. This results in hidden sizes of $n = 170$ for scoRNN, $n = 60$ for LSTM, $n = 120$ for the Full-Capacity uRNN and $n = 950$ for the Restricted-Capacity uRNN.

0.58	0.23	0.84	0.06	0.71	0.35	0.22	0.63	0.14	0.97
0	0	1	0	0		0	1	0	0	0

Figure 2.5: An illustration of the Adding Problem. The goal of the machine is to output the sum of the entries marked by one, in this case $0.84+0.22 = 1.06$

Hyperparameters for the scoRNN model were $\rho = 85$ for sequence length $T = 200$ and $\rho = 119$ for sequence lengths $T = 400$ and $T = 750$. Optimizer and learning rates were RMSProp 10^{-4} to update the skew-symmetric matrix and Adam 10^{-3} for all other weights. For the LSTM, the forget bias was initialized to be 2 for $T = 200$, 4 for $T = 400$, and 0 for $T = 750$. All trainable parameters were updated using an Adam optimizer with learning rate 10^{-2} . The hyperparameters used for the Restricted-Capacity uRNN were in accordance with [1] and [48] with an RMSProp optimizer with learning rate 10^{-4} for all sequences. The hyperparameter settings for the Full-Capacity uRNN were in accordance with [48] with an RMSProp optimizer and learning rate of 10^{-5} . For the oRNN, the best hyperparameters were in accordance with [35] which was $n = 128$ with 16 reflections with an Adam optimizer and learning rate of 0.01, $\approx 2.6k$ parameters. We found that performance decreased when matching the number of reflections to the hidden size to increase the number of parameters. For the EURNN, both the tunable-style and FFT-style EURNN with $n = 512$ were tested. We found better results from the tunable-style ($\approx 3k$ parameters) for sequence lengths $T = 200$ and $T = 400$, and from the FFT-style ($\approx 7k$ parameters) for sequence length $T = 750$.

The test set MSE results for sequence lengths $T = 200$, $T = 400$, and $T = 750$ can be found in Figure 2.6. For each case, the networks start at or near the baseline MSE, except for the EURNN for $T = 750$, and drop towards zero after a few epochs. As the sequence length increases, the number of epochs before the drop increases. As can be seen, the LSTM error drops precipitously across the board. Although the oRNN begins to drop below the baseline before scoRNN, it has a much more irregular descent curve, and jumps back to the baseline after several epochs.

MNIST

The **Mixed National Institute for Standards and Technology (MNIST)** [30] database is a database consisting of gray scale 28×28 pixel images of handwritten integer digits ranging from 0 to 9, see Figure 2.7 for examples. Following the implementation of [29], each pixel of the image is fed into the RNN sequentially, resulting in a single pixel sequence length of 784. In the first experiment, which we refer to as unpermuted MNIST, pixels are arranged in the sequence row-by-row. In the second, which we call permuted MNIST, a fixed permutation is applied to training and testing sequences.

Each experiment used a training set of 55,000 images and a test set of 10,000 images. The machines were trained for 70 epochs, and test set accuracy, the percentage

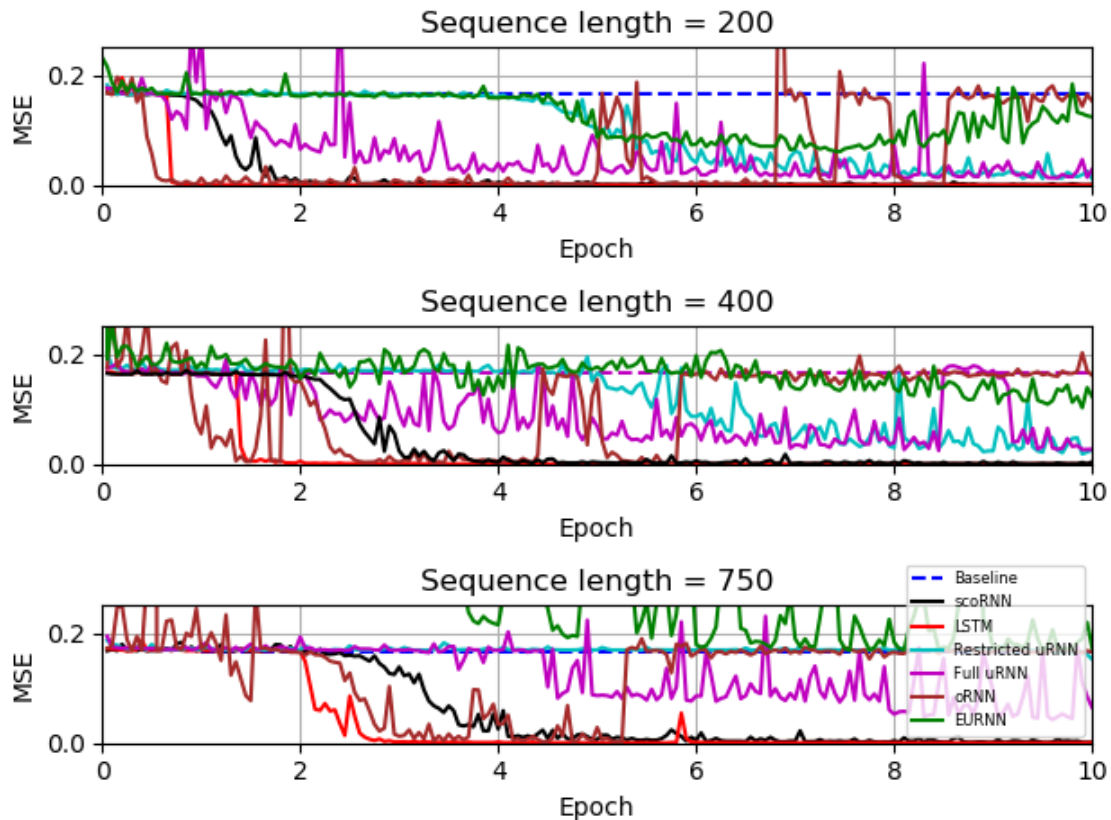


Figure 2.6: Test set MSE for each machine on the adding problem with sequence lengths of $T = 200$ (top), $T = 400$ (middle), and $T = 750$ (bottom).

of test images classified correctly, were evaluated at the conclusion of each epoch. Figures 2.8 and 2.9 shows test set accuracy over time, and the best performance over all epochs by each machine is given in Table 2.1.

All scoRNN machines were trained with the RMSProp optimization algorithm. Input and output weights used a learning rate of 10^{-3} , while the recurrent parameters used a learning rate of 10^{-4} (for $n = 170$) or 10^{-5} (for $n = 360$ and $n = 512$). For unpermuted MNIST, we found the number of negative ones used in D to be optimal at $n/10$ and for permuted MNIST to be $n/2$. The difference between these two values is expected to come from the different types of dependencies in each problem: unpermuted MNIST has mostly local dependencies, while permuted MNIST requires learning many long-term dependencies, which appear to be more easily modeled when the diagonal of D has a higher proportion of -1 s.

The LSTM used an RMSProp optimization algorithm with learning rate of 10^{-3} for hidden sizes $n = 128$ and $n = 256$ and learning rate of 10^{-4} for hidden size $n = 512$. The forget gate bias was initialized to be one. The restricted-capacity uRNN used an RMSProp optimization algorithm with learning rate of 10^{-4} . The Full-Capacity uRNN also used an RMSProp optimization algorithm with learning

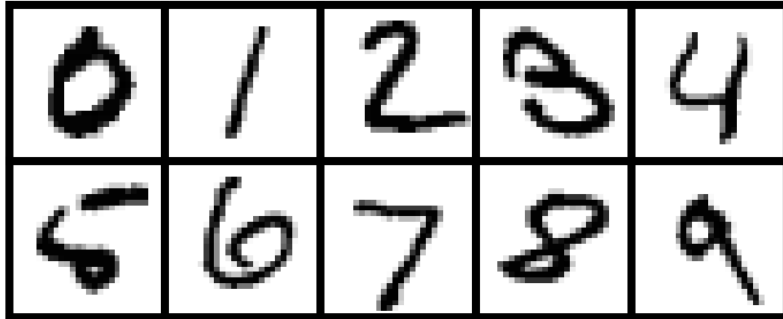


Figure 2.7: Example images of handwritten digits from the MNIST data set.

Table 2.1: Results for unpermuted and permuted pixel-by-pixel MNIST experiments. Evaluation accuracies are based on the best test accuracy at the end of every epoch. Asterisks indicate reported results from [23] and [35].

Model	n	# params	Permuted	
			MNIST Test Acc.	MNIST Test Acc.
scoRNN	170	$\approx 16k$	0.973	0.943
scoRNN	360	$\approx 69k$	0.983	0.962
scoRNN	512	$\approx 137k$	0.985	0.966
LSTM	128	$\approx 68k$	0.987	0.920
LSTM	256	$\approx 270k$	0.989	0.929
LSTM	512	$\approx 1,058k$	0.985	0.920
Restricted uRNN	512	$\approx 16k$	0.976	0.945
Restricted uRNN	2170	$\approx 69k$	0.984	0.953
Full uRNN	116	$\approx 16k$	0.947	0.925
Full uRNN	512	$\approx 270k$	0.974	0.947
EURNN	512	$\approx 9k$	-	0.937*
oRNN	256	$\approx 11k$	0.972*	-

rate of 10^{-3} for $n = 116$ and learning rate of 10^{-4} for $n = 512$.

In both experiments, the 170 hidden unit scoRNN gives similar performance to both of the 512 hidden unit uRNNs using a much smaller hidden dimension and, in the case of the Full-Capacity uRNN, an order of magnitude fewer parameters. Matching the number of parameters ($\approx 69k$), the 2170 Restricted-Capacity uRNN performance was comparable to the 360 hidden unit scoRNN for unpermuted MNIST, but performed worse for permuted MNIST, and required a much larger hidden size and a significantly longer run time. As in experiments presented in [1] and [48], orthogonal and unitary RNNs are unable to outperform the LSTM in the unpermuted case. However, the 360 and 512 hidden unit scoRNNs outperform all other unitary RNNs.

On permuted MNIST, the 512 hidden unit scoRNN achieves a test-set accuracy of 96.6%, outperforming all other architectures.

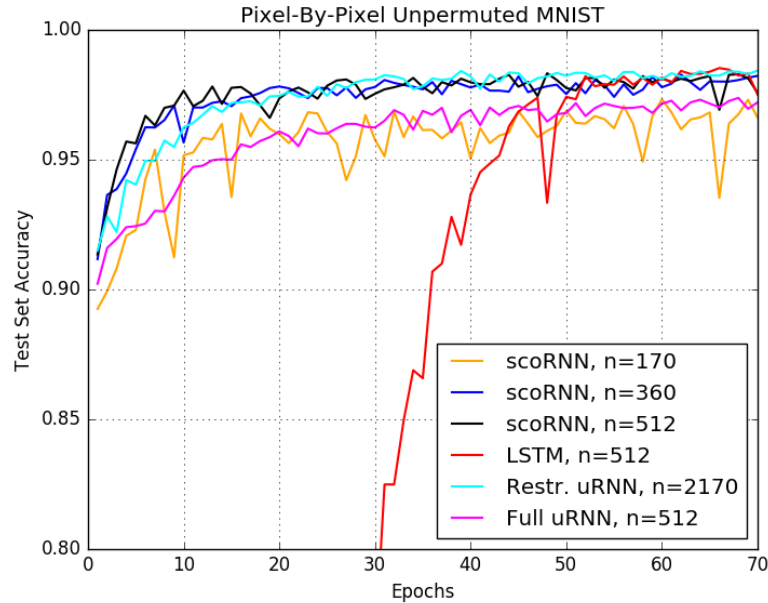


Figure 2.8: Test accuracy for unpermuted MNIST over time. All scoRNN models and the best performing LSTM, Restricted-Capacity uRNN, and Full-Capacity uRNN are shown.

TIMIT Speech Dataset

The scoRNN architecture was also tested on audio data using the TIMIT dataset [10]. The TIMIT dataset is a collection of real-world speech recordings of American English from 8 dialect regions of the United States. Following the experimental settings in [48], only the core test set was used which excludes the SA dialects. The resulting dataset consisted of 3,696 training and 192 testing audio files. Each audio file was downsampled from 16kHz to 8kHz and a **short-time Fourier transform (STFT)** was applied with a Hann window of 256 samples and a window hop of 128 samples (16 milliseconds). The result is a set of frames, each with 129 complex-valued Fourier amplitudes. The log magnitude of these amplitudes is used as the input data for the networks with the goal of predicting the next log magnitude in the sequence. Each sequence is padded with zeros to make uniform lengths. A batch size of 28 was used.

For each model, the hidden layer sizes were adjusted so each had approximately the same number of trainable parameters. For scoRNN, we used the Adam optimizer with learning rate 10^{-3} to train input and output parameters, and RMSProp with a learning rate of 10^{-3} (for $n = 224$) or 10^{-4} (for $n = 322, 425$) to train the recurrent weight matrix. The number of negative eigenvalues used was $\rho = n/10$. The LSTM forget gate bias was initialized to -4 and an RMSProp optimizer with learning rate

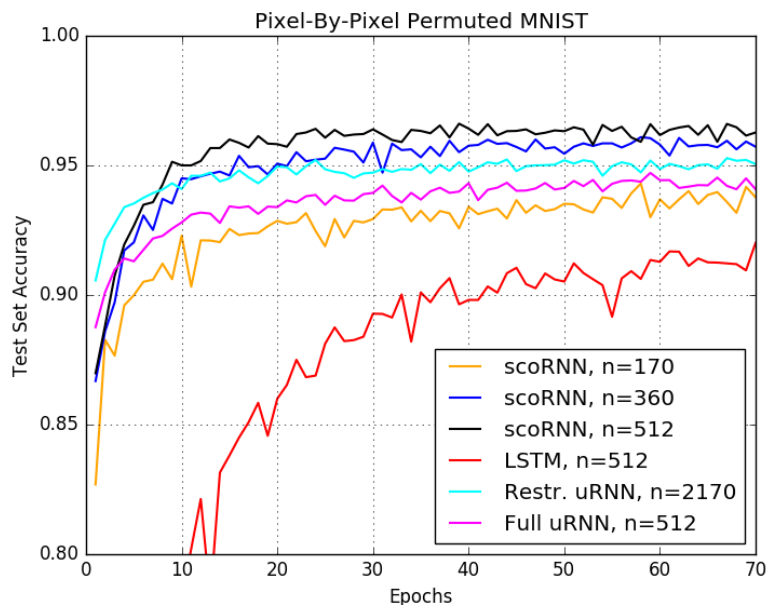


Figure 2.9: Test accuracy for permuted MNIST over time. All scoRNN models and the bets performing LSTM, Restricted-Capacity uRNN, and Full-Capacity uRNN are shown.

10^{-3} was used. The Restricted-Capacity and Full-Capacity uRNN were trained with an RMSProp optimizer with learning rate 10^{-3} .

The loss function was the mean squared error (MSE) between the predicted and actual log-magnitudes of the next time frame over the entire sequence. Table 2.2 contains the MSE on the validation and testing sets for 200 epochs based on the lowest obtain validation MSE. We note the MSE is computed slightly different for the LSTM and scoRNN models from the Restricted and Full-Capacity uRNNs. The Restricted and Full-Capacity uRNNs compute the MSE by taking the squared difference between the predicted and actual log magnitudes and applying a mask to zero out padded entries before computing the batch mean. The LSTM and scoRNN models compute the MSE by taking the squared difference between the predicted and actual log magnitudes but do not apply a mask to zero out padded entries before computing the batch mean. The method used for the LSTM and scoRNN models to compute the MSE will result in slightly higher MSE scores than the method used for the Restricted and Full-Capacity uRNNs. Despite this disadvantage, all scoRNN models achieve a smaller MSE than all LSTM and unitary RNN models. Similar to [48], we reconstruct the audio files using the predicted log magnitudes from each machine and evaluated them on several audio metrics. The signal-to-noise ratio metric (SegSNR) [5] is the ratio of the signal to background noise measured in decibels. A score greater than one indicates more signal than noise with a higher score being better than a lower score. The Short-Time Objective Intelligibility (STOI) [43] score ranges from 0 to 1 with a higher score representing a higher speech intelligibility and

Table 2.2: Results for the TIMIT speech dataset based on the best validation MSE.

Model	n	# params	Valid. MSE	Test MSE
scoRNN	224	\approx 83k	9.26	8.50
scoRNN	322	\approx 135k	8.48	7.82
scoRNN	425	\approx 200k	7.97	7.36
LSTM	84	\approx 83k	15.42	14.30
LSTM	120	\approx 135k	13.93	12.95
LSTM	158	\approx 200k	13.66	12.62
Rest. uRNN	158	\approx 83k	15.57	18.51
Rest. uRNN	256	\approx 135k	15.90	15.31
Rest. uRNN	378	\approx 200k	16.00	15.15
Full uRNN	128	\approx 83k	15.07	14.58
Full uRNN	192	\approx 135k	15.10	14.50
Full uRNN	256	\approx 200k	14.96	14.69

compares the discrete Fourier-transform bins from the original and predicted signal to measure human intelligibility. The Perceptual Evaluation of Speech Quality (PESQ) [41] score ranges from 1 to 5 with higher score indicated a higher speech quality. We found that the scoRNN predictions achieved better scores on SegSNR but performed slightly worse than the Full-Capacity uRNN predictions on STOI and PESQ.

Loss of Unitarity

In the scoRNN architecture, the recurrent weight matrix is parameterized by a skew-symmetric matrix through the Cayley transform. This ensures the recurrent weight matrix is orthogonal within the order of machine precision. Unlike scoRNN, the Full-Capacity uRNN maintains a unitary recurrent weight matrix through a multiplicative update scheme. After many iterations, rounding errors will accumulate because of the repeated multiplication of matrices. Eventually the rounding errors may result in a recurrent weight matrix that is no longer unitary. To analyze this, we check the orthogonality of the recurrent weight matrices in the scoRNN and Full-Capacity uRNN architectures. This is done at the end of every epoch on the unpermuted MNIST task. Each model has an equal hidden unit size of $n = 512$. Results of this experiment are shown in Figure 2.10. As can be seen, the recurrent weight matrix for the Full-Capacity uRNN becomes less unitary over time, but the recurrent weight matrix for scoRNN maintains orthogonality and is not affected by roundoff errors. This issue is of particular concern when using a graphics processing unit (GPU) which have a much lower machine precision than a standard computer processing unit (CPU).

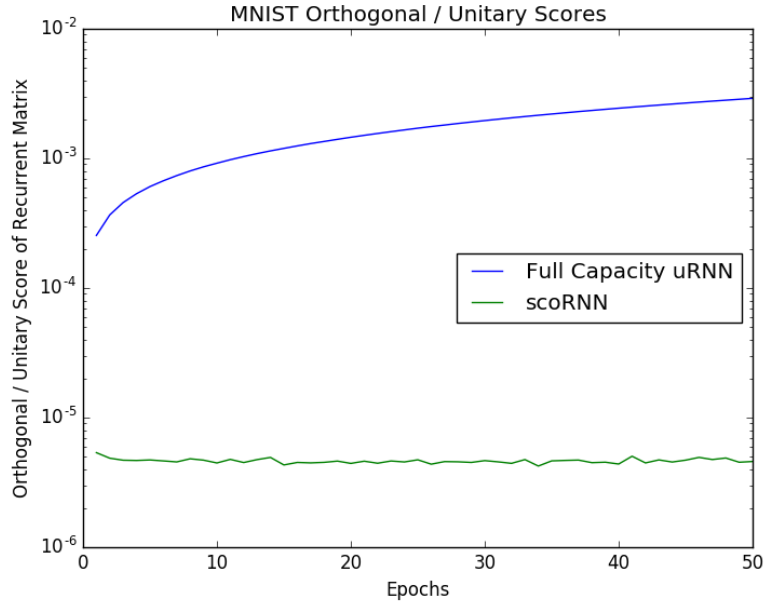


Figure 2.10: Unitary scores ($\|W^*W - I\|_F$) for the Full-Capacity uRNN recurrent weight matrix and orthogonality scores ($\|W^TW - I\|_F$) for the scoRNN recurrent weight matrix using a GPU on the pixel-by-pixel MNIST experiment.

Gradient Stability

As discussed in Section 2.2, the vanishing/exploding gradient problem occurs when the gradient of the hidden state $\frac{\partial L}{\partial h_t}$ either goes to zero or becomes unbounded as the gradients propagate back through the sequence. To see if the scoRNN architecture is still affected by vanishing/exploding gradients, we examine the hidden state gradients on the adding problem with sequence length $T = 500$. See Figure 2.11 for a plot of the gradient norms at the beginning of training and after 300 training iterations. As can be seen, the scoRNN gradient norms have a slight decay by less than an order of magnitude at the beginning of training with a value around 10^{-2} . Even after 300 iterations of training, the scoRNN gradient norms decay from 10^{-3} at $t = 500$ to 10^{-4} at $t = 0$. In contrast, the LSTM gradient norms decrease quickly as the gradients propagate backwards through time. This decay of gradient norms becomes more pronounced after 300 epochs. These results indicate that information can more easily propagate from the beginning of the sequence towards the end when the recurrent weight matrix is orthogonal.

Complexity and Speed

The scoRNN architecture has comparable complexity and memory usage as a standard RNN except for the additional memory requirement of storing the $n(n - 1)/2$ entries of the skew-symmetric matrix A and the additional $\mathcal{O}(n^3)$ complexity of forming the recurrent weight matrix W from A through the scaled Cayley transform. It should be noted that W is generated only once per training iteration. In contrast, if

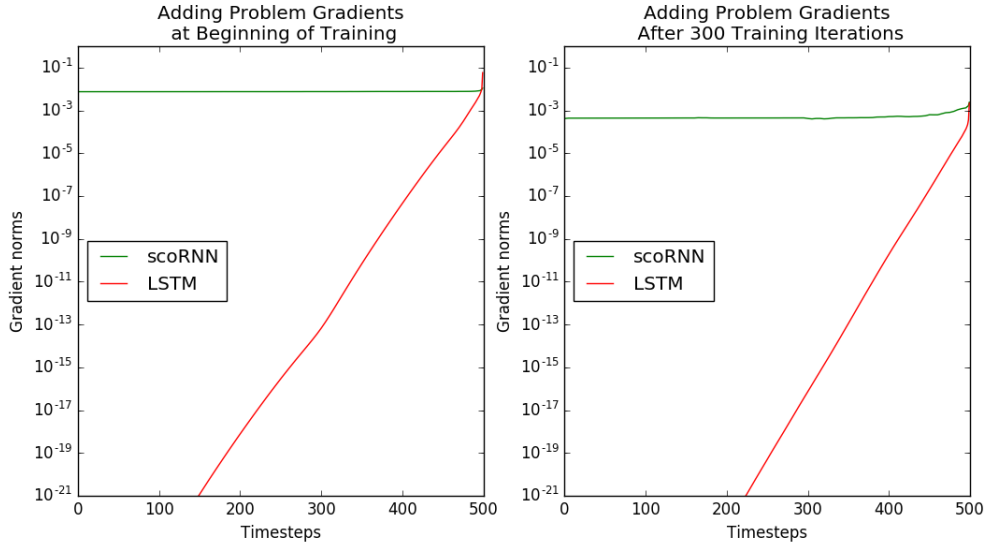


Figure 2.11: Gradient norms $\|\frac{\partial L}{\partial h_t}\|$ for scoRNN and LSTM models during training on the adding problem. The x -axis shows different values of t . The left plot shows gradients at the beginning of training, and the right shows gradients after 300 training iterations.

we let T be the sequence length and B the batch size, then a standard RNN requires a complexity of $\mathcal{O}(BTn^2)$. When BT is comparable to n , which is normally the case in practice, then a standard RNN has the same order of magnitude of complexity as the scoRNN architecture.

To experimentally quantify potential differences between scoRNN and other models, the real run-time for the unpermuted MNIST experiment were recorded and are included in Table 2.3. All models were run on the same machine, which has an Intel Core i5-7400 processor and an nVidia GeForce GTX 1080 GPU. The scoRNN and LSTM models were run in Tensorflow, while the full and restricted capacity uRNNs were run using code provided in [48].

The RNN and LSTM models were the fastest with negligible differences in run time per epoch for larger hidden sizes. It is suspected that this is because these models were built in to Tensorflow which has been optimized for efficiency. The LSTMs are of similar speed to the $n = 170$ scoRNN, while they are approximately twice as fast as the $n = 512$ scoRNN. Matching the number of hidden parameters, the scoRNN model with $n = 170$ is approximately 1.5 times faster than the Restricted-Capacity uRNN with $n = 512$, and twice as fast as the Full-Capacity uRNN with $n = 116$. This relationship can also be seen in the scoRNN and Full-Capacity uRNN models with $\approx 137k$ parameters, where the scoRNN takes 11.2 minutes per epoch as compared to 25.8 minutes for the Full-Capacity uRNN.

Table 2.3: Timing results for the unpermuted MNIST dataset.

Model	n	# params	Minutes Per Epoch
RNN	116	$\approx 16\text{k}$	2.3
scoRNN	170	$\approx 16\text{k}$	5.3
Rest. uRNN	512	$\approx 16\text{k}$	8.2
Full uRNN	116	$\approx 16\text{k}$	10.8
LSTM	128	$\approx 68\text{k}$	5.0
scoRNN	360	$\approx 69\text{k}$	7.4
Rest. uRNN	2,170	$\approx 69\text{k}$	50.1
RNN	360	$\approx 137\text{k}$	2.3
scoRNN	512	$\approx 137\text{k}$	11.2
Full uRNN	360	$\approx 137\text{k}$	25.8
RNN	512	$\approx 270\text{k}$	2.4
LSTM	256	$\approx 270\text{k}$	5.2
Full uRNN	512	$\approx 270\text{k}$	27.9
LSTM	512	$\approx 1,058\text{k}$	5.6

2.6 Scaled Cayley Unitary Recurrent Neural Network

As discussed in Section 2.5, the scoRNN architecture was developed to address the vanishing/exploding gradient problem by maintaining an orthogonal recurrent weight matrix. This is done by parameterizing the matrix with a skew-symmetric matrix through the scaled Cayley transform. This parameterization requires a diagonal scaling matrix D consisting of ± 1 entries. These parameters are discrete and can not be optimized by gradient descent. Since it is not known apriori what the optimal number of negative ones should be for a particular task, a tunable hyperparameter is introduced in the scoRNN architecture that controls the number of negative ones. Since this hyperparameter is set throughout training, D is fixed and so the possible set of orthogonal recurrent weight matrices is restricted to a subset of all orthogonal matrices. To overcome this restriction, the **scaled Cayley unitary RNN (scuRNN)** was developed.

Similar to scoRNN, the scuRNN architecture is based on a scaled Cayley transform but uses the unitary version of Theorem 2.5.2 to maintain a unitary recurrent weight matrix. In other words,

$$W = (I + A)^{-1} (I - A) D \quad (2.68)$$

where $W \in \mathbb{C}^{n \times n}$ is unitary, $A \in \mathbb{C}^{n \times n}$ is skew-Hermitian, and $D \in \mathbb{C}^{n \times n}$ is a diagonal matrix with entries lying on the complex unit circle of the form $e^{i\theta}$. Unlike scoRNN, this parameterization is differentiable with respect to the continuous θ variable and can be optimized using gradient descent. This eliminates the need for tuning a hyperparameter and having a fixed scaling matrix during training.

Architecture Details

Since most machine learning programs are designed to handle only real valued matrices and scuRNN requires matrices that are complex valued, careful consideration must be given on how to implement the network. It first should be noted that the scuRNN model operates similarly to a standard RNN, see (2.1), (2.2), and (2.3), except the modReLU activation function is used, see (2.33), and all parameters are complex valued. In particular, (2.1) and (2.2) become

$$a_i = Ux_i + Wh_{i-1} \quad (2.69)$$

$$h_i = \frac{a_i}{|a_i|} \sigma_{\text{ReLU}}(|a_i| + b) \quad (2.70)$$

In order to perform complex matrix and vector multiplication as discussed in Section 2.4, all complex parameters associated with scuRNN are separated into their real and imaginary components. Since the input of the network is real valued and the hidden state of the network is complex valued, we simply take the input x_i and multiply it by the real and complex components of the input matrix U or

$$Ux_i = \begin{bmatrix} \text{Re}(U) \\ \text{Im}(U) \end{bmatrix} x_i \quad (2.71)$$

Similarly, since the output of the network is real valued, we simply make V the appropriate shape so that

$$Vh_i = V \begin{bmatrix} \text{Re}(h_i) \\ \text{Im}(h_i) \end{bmatrix} \quad (2.72)$$

Finally, in order to train the skew-Hermitian matrix A and scaling matrix D that are used to parameterize the unitary recurrent weight matrix in scuRNN, complex derivatives must be used. When we consider the loss function as a function of the complex matrix A or scaling matrix D with a range on the real-line, the loss function is nonholomorphic and thus not complex differentiable. To compute the necessary gradients, Wirtinger calculus [47] is required. As discussed in Section 1.6, the steepest descent direction is given by

$$\frac{\partial f(z)}{\partial \bar{z}} \quad (2.73)$$

Using the Wirtinger derivatives in (1.43) and the steepest descent direction in (2.73), we update the unitary recurrent weight matrix W by performing gradient descent on the associated skew-Hermitian parameterization matrix A and scaling matrix D . We note that in order to compute gradients with respect to A , we must pass the gradients through the scaled Cayley transform. The desired gradients for A and the diagonal arguments of D are given in (2.74) and (2.75)

$$\frac{\partial L}{\partial A} = C^T - \bar{C} \quad (2.74)$$

$$\frac{\partial L}{\partial \theta} = 2\text{Re} \left(i \left(\left(\frac{\partial L}{\partial W} \right)^T K \right) \odot I \right) d \quad (2.75)$$

where L is a differentiable loss function, $C = (I + A)^{-T} \frac{\partial L}{\partial W} (D + W^T)$, $K = (I + A)^{-1} (I - A)$, $d = [e^{i\theta_1}, e^{i\theta_2}, \dots, e^{i\theta_n}]^T$, and $\theta = [\theta_1, \theta_2, \dots, \theta_n]^T$. See [33] for more details.

To update the unitary matrix W , we first compute $\frac{\partial \mathcal{L}}{\partial W}$ using the given machine learning program. We then compute $\frac{\partial \mathcal{L}}{\partial \theta}$ and $\frac{\partial \mathcal{L}}{\partial A}$ using (2.74) and (2.75). A gradient descent optimizer is used to update the θ entries of the scaling matrix D . These updated values are then used to reform D . We next update A using a gradient descent algorithm and finally reform W through the scaled Cayley transform. Denoting λ and γ as learning rates, the order of updates is as follows

$$\theta^{(k+1)} \leftarrow \theta^{(k)} - \lambda \frac{\partial L}{\partial \theta^{(k)}} \quad (2.76)$$

$$D^{(k+1)} \leftarrow \text{diag} \left(e^{i\theta_1^{(k+1)}}, e^{i\theta_2^{(k+1)}}, \dots, e^{i\theta_n^{(k+1)}} \right) \quad (2.77)$$

$$A^{(k+1)} \leftarrow A^{(k)} - \gamma \frac{\partial L}{\partial A} \quad (2.78)$$

$$W^{(k+1)} \leftarrow (I + A^{(k+1)})^{-1} (I - A^{(k+1)}) D^{(k+1)} \quad (2.79)$$

It should be noted that $\frac{\partial \mathcal{L}}{\partial A}$ is skew-Hermitian. Since the addition of any skew-Hermitian matrix is again skew-Hermitian, the SGD optimizer will ensure $A^{(k+1)}$ will be skew-Hermitian. Thus $W^{(k+1)}$ will be unitary to the order of machine precision. For optimizers that require squaring of the gradients elementwise such as RMSProp and Adam, the real and imaginary components are updated separately in order to ensure that $A^{(k+1)}$ is still skew-Hermitian. All other trainable parameters are updated using gradient descent or a related algorithm.

Experiments

The experiments conducted in [33] were the same as in [14] and consisted of MNIST, copying, adding, and TIMIT problems. The scuRNN model had slightly better performance than the scoRNN model on the copying problem with sequence length $T = 2,000$ and the TIMIT problem. Performance was slightly worse on the adding problem of sequence length $T = 750$. For the remaining experiments, performance of the scuRNN model was comparable to the scoRNN model. It should be noted that the scuRNN model does not require the tuning of an additional hyperparameter to control the number of negative ones in the scaling matrix like the scoRNN model. See [33] for more details.

2.7 Eigenvalue Normalized Recurrent Neural Network

Several variants of RNNs with orthogonal or unitary recurrent matrices have recently been developed to mitigate the vanishing/exploding gradient problem and to model long-term dependencies of sequences. In spite of the promises shown in recent work, orthogonal/unitary RNNs still have some shortcomings. An orthogonal/unitary recurrent weight matrix can allow an input to affect an output over long sequences,

but unlike gated architectures, orthogonal/unitary RNNs lack “forget” mechanisms [21] to discard information that is no longer needed. In sequences where certain input information is only used for the states or outputs locally, the state may be consumed with such information, reducing its capacity for carrying other information. This makes it difficult to efficiently model sequences with both long and short-term dependencies.

The **Eigenvalue Normalized RNN** expands upon the orthogonal/unitary RNN architecture by incorporating a dissipative state to model short-term dependencies. The ENRNN forms a recurrent matrix with a spectral radius (i.e. the largest absolute value of the eigenvalues) less than 1 by normalizing another parametric matrix by its spectral radius. Any input to this state will dissipate in time with repeat multiplication by the recurrent matrix and will be replaced with new input information. This state can be concatenated with another state with an orthogonal/unitary recurrent matrix to form an RNN that has a long and short-term memory component to efficiently model long sequences. The resulting architecture falls within the existing framework of the basic RNN.

The ENRNN is inspired by work on the **Spectral Normalized Generative Adversarial Network (SN-GAN)** [36]. The SN-GAN normalizes the discriminator weight matrix by its spectral norm, i.e. its largest singular value. Noting that the spectral radius is bounded by any matrix norm including the spectral norm, normalization by the spectral norm is expected to make the spectral radius of the matrix much less than 1. The importance of constraining the eigenvalues of the recurrent matrix rather than its singular values should be emphasized because the eigenvalues affect the dynamical behavior of an RNN but the singular values do not, see also [3]. For example, all orthogonal matrices have singular values equal to 1, but may define very different RNNs.

Architecture

In order to improve the capacity of orthogonal/unitary RNNs to capture short-term dependencies, the ENRNN introduces a dissipative state. Let $h_t \in \mathbb{R}^n$ be the hidden state consisting of two components: $h_t^{(L)} \in \mathbb{R}^q$ that captures long-term dependencies and $h_t^{(S)} \in \mathbb{R}^{n-q}$ that captures short-term dependencies. In this scheme, q is considered a hyperparameter. Now let $W^{(L)} \in \mathbb{R}^{q \times q}$ be an orthogonal matrix used as the recurrent matrix for $h_t^{(L)}$ that is designed to propagate information over many time steps, and $W^{(S)} \in \mathbb{R}^{(n-q) \times (n-q)}$ which has a spectral radius less than one by normalizing with the spectral radius. If we consider a recurrent weight matrix $W \in \mathbb{R}^{n \times n}$ of the form $W = \text{diag}(W^{(L)}, W^{(S)})$, then a forward pass of the RNN will be:

$$\begin{cases} h_t^{(L)} = \sigma \left(U^{(L)} x_t + W^{(L)} h_{t-1}^{(L)} + b^{(L)} \right) \\ h_t^{(S)} = \sigma \left(U^{(S)} x_t + W^{(S)} h_{t-1}^{(S)} + b^{(S)} \right) \\ y_t = V^{(L)} h_t^{(L)} + V^{(S)} h_t^{(S)} + c \end{cases} \quad (2.80)$$

Since $W^{(S)}$ has a spectral radius less than 1, the effect of any input on $h^{(S)}$ will decay quickly from repeat multiplication by $W^{(S)}$ with the rate of decay controlled by the

magnitude of the eigenvalues of $W^{(S)}$. Different eigenvalues with different magnitudes will then decay at different rates, emulating different lengths of memory.

In this model, the output y_t is determined from a combination of $h_t^{(L)}$ and $h_t^{(S)}$ where $h_t^{(S)}$ contains information of recent input data, see (2.80). In this way, short-term memory that is needed to determine y_t is stored in $h_t^{(S)}$, but once y_t is computed, $h_t^{(S)}$ will be gradually replaced by information from new inputs. This allows $h_t^{(L)}$ to store and carry only long-term memory information needed for the output.

In the architecture (2.80), the hidden states $h^{(L)}$ and $h^{(S)}$ are separate. They carry the long and short-term memory in parallel and the short-term state is directly used to determine output. If the task is to determine a single output from a sequence at the end of the entire sequence, then $h^{(S)}$ does not affect the output until near the end of the sequence. In this case, it may still be beneficial to have $h^{(S)}$ accumulate short-term memory but to feed it into $h^{(L)}$ to indirectly affect the final output. This can be done by adding a coupling block to the recurrent matrix,

$$W = \left[\begin{array}{c|c} W^{(L)} & W^{(C)} \\ \hline & W^{(S)} \end{array} \right] \quad (2.81)$$

where $W^{(C)} \in \mathbb{R}^{q \times (n-q)}$ is called a coupling matrix. Applying the recurrent matrix in (2.81) to a forward pass of the RNN, we obtain:

$$\begin{cases} h_t^{(L)} &= \sigma \left(U^{(L)} x_t + W^{(L)} h_{t-1}^{(L)} + W^{(C)} h_{t-1}^{(S)} + b^{(L)} \right) \\ h_t^{(S)} &= \sigma \left(U^{(S)} x_t + W^{(S)} h_{t-1}^{(S)} + b^{(S)} \right) \\ y_t &= V^{(L)} h_t^{(L)} + V^{(S)} h_t^{(S)} + c \end{cases} \quad (2.82)$$

In this case, $h^{(S)}$ is generated by the same recurrence as before and stores short-term information of the inputs. However, with the coupling block, $h^{(L)}$ is determined from the current input, the short-term hidden state $h^{(S)}$, and $h^{(L)}$. This interaction is similar to the update of the internal state of an LSTM. In particular, $h^{(S)}$ can be regarded as a preprocessing of several consecutive inputs designed to extract information to be used to update the long-term memory state $h^{(L)}$. As an example, one can think of character inputs in a language processing problem. The short-term memory state may process the character inputs to produce word or phrase information to be used in the long-term state $h^{(L)}$ so that $h^{(L)}$ can be devoted to processing the information at a higher level. We believe this separation of the processing of characters from the processing at a higher level of sentences or concepts will be more effective and efficient.

We note that since W is an upper triangular matrix, the eigenvalues of W consist of the eigenvalues of both $W^{(L)}$ and $W^{(S)}$ and so has a spectral radius of at most one and this coupling does not alter the spectral properties of the recurrent matrix. For this reason, we do not allow a coupling from $h^{(L)}$ to $h^{(S)}$ because the fully dense recurrent matrix would not preserve the desired spectral properties.

To illustrate how $h^{(S)}$ can simulate a short-term memory state, we note that since $\rho(W^{(S)}) < 1$, there exists some norm $\|\cdot\|$ such that $\|W^{(S)}\| < 1$. If we assume that this holds for the 2-norm, i.e. $\|W^{(S)}\|_2 < 1$, we formulate the following theorem.

Theorem 2.7.1. For an RNN as defined in (2.80) and (2.82) with a ReLU nonlinearity, if $\|W^{(S)}\|_2 < 1$ then

$$\left\| \frac{\partial h_{t+\tau}^{(S)}}{\partial h_t^{(S)}} \right\| \leq \|W^{(S)}\|^\tau \text{ and } \left\| \frac{\partial h_{t+\tau}^{(S)}}{\partial x_t} \right\| \leq \|W^{(S)}\|_2^\tau \|U^{(S)}\| \quad (2.83)$$

where $\|\cdot\|$ is the 2-norm.

Proof. By the chain rule, we obtain:

$$\frac{\partial h_{t+\tau}^{(S)}}{\partial h_t^{(S)}} = \frac{\partial h_{t+\tau}^{(S)}}{\partial h_{t+\tau-1}^{(S)}} \frac{\partial h_{t+\tau-1}^{(S)}}{\partial h_{t+\tau-2}^{(S)}} \cdots \frac{\partial h_{t+1}^{(S)}}{\partial h_t^{(S)}} \quad (2.84)$$

$$= \prod_{k=t+\tau}^{t+1} \frac{\partial h_k^{(S)}}{\partial h_{k-1}^{(S)}} \quad (2.85)$$

$$= \prod_{k=t+\tau}^{t+1} G_k W^{(S)} \quad (2.86)$$

where $G_k = \text{diag}(\sigma'(a_j))$ is a diagonal matrix consisting of the derivative of the nonlinearity function for each activation a_j at time step k . Now taking the two-norm to both sides,

$$\left\| \frac{\partial h_{t+\tau}^{(S)}}{\partial h_t^{(S)}} \right\|_2 \leq \prod_{k=t+\tau}^{t+1} \|G_k\|_2 \|W^{(S)}\|_2 \leq \|W^{(S)}\|_2^\tau \quad (2.87)$$

Similarly, we have

$$\frac{\partial h_{t+\tau}^{(S)}}{\partial x_t} = \frac{\partial h_{t+\tau}^{(S)}}{dh_t^{(S)}} \frac{\partial h_t^{(S)}}{\partial x_t} = \frac{\partial h_{t+\tau}^{(S)}}{dh_t^{(S)}} G_t U^{(S)} \quad (2.88)$$

and taking the two-norm to both sides we obtain:

$$\left\| \frac{\partial h_{t+\tau}^{(S)}}{\partial x_t} \right\|_2 \leq \|W^{(S)}\|_2^\tau \|U^{(S)}\|_2 \quad (2.89)$$

□

We remark that as τ increases, the derivative bounds in Theorem 2.7.1 go to zero, indicating the dependence of $h_{t+\tau}^{(S)}$ on $h_t^{(S)}$ and x_t goes to zero.

ENRNN Gradient Descent

The training of $W^{(S)}$ by gradient descent can easily lead to a matrix with spectral radius greater than 1. To maintain $W^{(S)}$ with spectral radius less than 1, we parameterize it by another matrix $T \in \mathbb{R}^{(n-q) \times (n-q)}$ through the normalization

$$W^{(S)} = W^{(S)}(T) := \frac{T}{\rho(T) + \epsilon} \quad (2.90)$$

for some small $\epsilon > 0$, where $\rho(T) \in \mathbb{R}$ is the spectral radius of T . In this way, $W^{(S)}$ has eigenvalues with modulus less than one and the training of $W^{(S)}$ is carried out in T . Namely, for an RNN loss function $L = L(W^{(S)})$ in terms of $W^{(S)}$, we regard it as a function $L = L(W^{(S)}(T))$ of T . Instead of optimizing with respect to $W^{(S)}$, we optimize $L = L(W^{(S)}(T))$ with respect to T . The gradients of such a parameterization are given below in Proposition 2.7.1

Proposition 2.7.1. *Let $L = L(W) : \mathbb{R}^{m \times m} \rightarrow \mathbb{R}$ be some differentiable loss function for an RNN with a recurrent weight matrix W and let $\frac{\partial L}{\partial W} := \left[\frac{\partial L}{\partial W_{i,j}} \right] \in \mathbb{R}^{m \times m}$. Let W be parameterized by another matrix $T \in \mathbb{R}^{m \times m}$ as $W = \frac{T}{\rho(T) + \epsilon}$, where $\rho(T) \in \mathbb{R}$ is the spectral radius of T and $\epsilon > 0$ is a small positive number. If $\lambda = \alpha + i\beta$ (with $\alpha, \beta \in \mathbb{R}$) is a simple eigenvalue of T with $|\lambda| = \rho(T)$ and if $u \in \mathbb{C}^n$ and $v \in \mathbb{C}^n$ are corresponding right and left eigenvectors, i.e. $Tu = \lambda u$ and $v^*T = \lambda v^*$, then the gradient of $L = L(T)$ as a function of T is given by:*

$$\frac{\partial L}{\partial T} = \frac{1}{\tilde{\rho}(T)} \left[\frac{\partial L}{\partial W} - \frac{1}{\tilde{\rho}(T)} 1_m^T \left(\frac{\partial L}{\partial W} \odot W \right) 1_m C \right]$$

where $C = \alpha \operatorname{Re}(S) + \beta \operatorname{Im}(S)$ with $S = \frac{\bar{v}u^T}{v^*u} \in \mathbb{C}^{m \times m}$, $1_m \in \mathbb{R}^m$ is a vector consisting of all ones, $\tilde{\rho}(T) = \rho(T) + \epsilon$, $*$ is the conjugate transpose operator, and \odot is the Hadamard product.

Proof. To find the derivative with respect to the (i, j) entry of T , we obtain by the chain rule:

$$\begin{aligned} \frac{\partial L}{\partial T_{i,j}} &= \sum_{k,l=1}^m \frac{\partial L}{\partial W_{k,l}} \frac{\partial W_{k,l}}{\partial T_{i,j}} \\ &= \sum_{k,l=1}^m \frac{\partial L}{\partial W_{k,l}} \left(\frac{1}{\rho(T) + \epsilon} \frac{\partial T_{k,l}}{\partial T_{i,j}} + T_{k,l} \frac{\partial (r^{-1})}{\partial T_{i,j}} \right) \end{aligned} \quad (2.91)$$

where $r = \rho(T) + \epsilon$. Looking at the second term in (2.91) and using $\rho(T) = (\alpha^2 + \beta^2)^{\frac{1}{2}}$, we have

$$\frac{\partial (r^{-1})}{\partial T_{i,j}} = -\frac{1}{\rho(T) (\rho(T) + \epsilon)^2} \left(\alpha \frac{\partial \alpha}{\partial T_{i,j}} + \beta \frac{\partial \beta}{\partial T_{i,j}} \right) \quad (2.92)$$

Since λ is a simple eigenvalue, we have

$$\frac{\partial \lambda}{\partial T_{i,j}} = \frac{v^* \frac{\partial T}{\partial T_{i,j}} u}{v^* u} = \frac{\bar{v}_i u_j}{v^* u} \quad (2.93)$$

and hence $\frac{\partial \lambda}{\partial T} = \frac{\bar{v}u^T}{v^*u} = S$. So

$$\frac{\partial \alpha}{\partial T} = \operatorname{Re}(S); \quad \frac{\partial \beta}{\partial T} = \operatorname{Im}(S) \quad (2.94)$$

Combining (2.91), (2.92), and (2.94) with the fact that $\frac{\partial T_{k,l}}{\partial T_{i,j}} = 1$ for $(k,l) = (i,j)$ and 0 otherwise, we obtain

$$\begin{aligned} \frac{\partial L}{\partial T_{i,j}} &= \frac{1}{\rho(T)} \frac{\partial L}{\partial W_{i,j}} - \frac{1}{\rho(T)\rho(T)^2} \sum_{k,l=1}^m \frac{\partial L}{\partial W_{k,l}} T_{k,l} C_{i,j} \\ &= \frac{1}{\rho(T)} \left[\frac{\partial L}{\partial W_{i,j}} - \frac{1}{\rho(T)} \mathbf{1}_m^T \left(\frac{\partial L}{\partial W} \odot W \right) \mathbf{1}_m C_{i,j} \right] \end{aligned} \quad (2.95)$$

as desired. \square

We remark that if W is a real matrix, then complex eigenvalues appear in conjugate pairs, both of which give the spectral radius $\rho(T)$. However, the formula in the above theorem is independent of which eigenvalue we use. Specifically, if λ in the theorem is a complex eigenvalue, $\bar{\lambda}$ is also an eigenvalue with \bar{u} and \bar{v} as right and left eigenvectors. Using $\bar{\lambda}$, \bar{u} and \bar{v} in the theorem, we obtain the same formula for $\frac{\partial L}{\partial T}$ because $\frac{\bar{v}\bar{u}^T}{\bar{v}^*\bar{u}} = \bar{S}$ and correspondingly $\alpha \operatorname{Re}(\bar{S}) + \beta \operatorname{Im}(\bar{S}) = C$. Thus selecting either λ or $\bar{\lambda}$ in Proposition 2.7.1 will result in an identical derivative. In addition, when λ is a multiple eigenvalue, the computation of S involves a division by 0 or a number nearly 0. This is a rare situation and can be remedied in practice. First, it is unlikely to occur as the set of matrices with multiple eigenvalues lie on a low dimensional manifold in the space of $n \times n$ matrices and has a Lebesgue measure 0. Thus the probability of a random matrix having multiple eigenvalue is zero. Second, if a multiple or nearly multiple eigenvalue occurs, we may train using usual gradient descent without eigenvalue normalization for a few steps and return to $\frac{\partial L}{\partial T}$ when the eigenvalues are separated. This situation never occurred in our experiments.

Using Proposition 2.7.1, an optimizer with learning rate ζ is used to first update T which is then used to update $W^{(S)}$:

$$T_k \leftarrow T_{k-1} - \zeta \frac{\partial L}{\partial T_{k-1}}; \quad W_k^{(S)} \leftarrow \frac{T_k}{\rho(T_k) + \epsilon} \quad (2.96)$$

A naive approach may be to simply apply gradient descent on $W^{(S)}$ and then re-normalize $W^{(S)}$ by its spectral radius. The problem is that the computed gradients $\frac{\partial L}{\partial W^{(S)}}$ do not take into account the effects of the normalization. Thus a steepest descent step on $W^{(S)}$ will reduce the loss function, but it may not be the case after $W^{(S)}$ is re-normalized by the spectral radius. In contrast, our approach takes a gradient descent step on T , which decreases the loss function with the new W . Namely, the steepest descent direction $\frac{\partial L}{\partial T}$ has taken the eigenvalue normalization into account.

Initialization

We initialize T to be a random matrix with eigenvalues uniformly distributed on the complex unit disc. This is done in a way similar to scoRNN as

$$T = \operatorname{diag}(B_1, \dots, B_{\lfloor n/2 \rfloor}) \quad B_j = \gamma_j \begin{bmatrix} \cos t_j & -\sin t_j \\ \sin t_j & \cos t_j \end{bmatrix} \quad (2.97)$$

where each t_j is sampled from $\mathcal{U}[0, \frac{\pi}{2})$ and each γ_j is sampled from $\mathcal{U}[-1.0, 1.0)$. This results in eigenvalues of the form $\gamma_j e^{\pm i t_j}$ which are uniformly distributed on the complex unit disc. For the coupling matrix, $W^{(C)}$, initialization is Glorot Uniform [11] unless indicated otherwise. The initial states of $h_0^{(L)}$ and $h_0^{(S)}$ are set to zero and are non-trainable.

It is unknown before hand if the largest eigenvalue should have a modulus near one, so we start by setting $W^{(S)} = T$ without eigenvalue normalization and train until $\rho(T) > 1$, at which point eigenvalue normalization is implemented. Namely, if $\rho(T) \leq 1$, then a standard gradient descent step is taken with $W^{(S)} = T$. Once an update step results in a $\rho(T) > 1$, (2.96) is used for all subsequent training steps.

Experiments

In this section, we present four experiments to compare ENRNN with LSTM and several orthogonal/unitary RNNs. Code for the experiments and hyperparameter settings for ENRNN are available at <https://github.com/KHelfrich1/ENRNN>. We compare models using single layer networks because implementation of multi-layer networks in the literature typically involves dropout, learning rate decay, and other multi-layer hyperparameters that make comparisons difficult. This is also the setting used in prior work on orthogonal/unitary RNNs. Each hidden state dimension is adjusted to match the number of trainable parameters, but ENRNN can be stacked in multiple layers. For ENRNN, the long-term recurrent matrix $W^{(L)}$ is parameterized using scoRNN [14]. For the short-term component state, we use $\epsilon = 0$ in Proposition 2.7.1. Unless noted otherwise, the activation function used was modReLU. For each method, the hyperparameters tuned included the optimizer {Adam, RMSProp, Adagrad}, and learning rates $\{10^{-3}, 10^{-4}, 10^{-5}\}$. For scoRNN, the number of negative ones used in the parameterization of the recurrent matrix is tuned in multiplies of 10% of the hidden size. For ENRNN, the size of the short-term state $W^{(S)}$ is tuned in multiplies of 10% of the entire hidden size up to 60%. For the LSTM, the forget gate bias initialization and gradient clipping threshold are tuned using integers in $[-4, 4]$ and in $[1, 10]$ respectively. These hyperparameters were selected using a gridsearch method. Experiments were run using Python3, Tensorflow, and CUDA9.0 on GPUs.

Adding Problem

The setup for the adding problem is the same as described in Section 2.5. The hidden sizes for each model were adjusted so they each had $\approx 15k$ trainable parameters which results in a total hidden size of $n = 160, 60, 60, 170, 128$, and 120 for the ENRNN, LSTM, Spectral RNN, scoRNN, oRNN, and Full-Capacity uRNN respectively. The ENRNN was comprised of an $h^{(L)}$ and an $h^{(S)}$ of respective sizes 96 and 64 with a coupling matrix, see (2.82). An RMSProp optimizer with learning rate 10^{-4} was used for all weights. The $W^{(L)}$ was parameterized with 29 negative ones. The LSTM used an Adam optimizer with learning rate 10^{-2} and an initial forget gate bias of 0. The Spectral RNN had an initial learning rate of 10^{-2} with learning rate decay of 0.99, r size of 16, and r margin of 0.01, similar to the settings in [51]. As per [14], the scoRNN

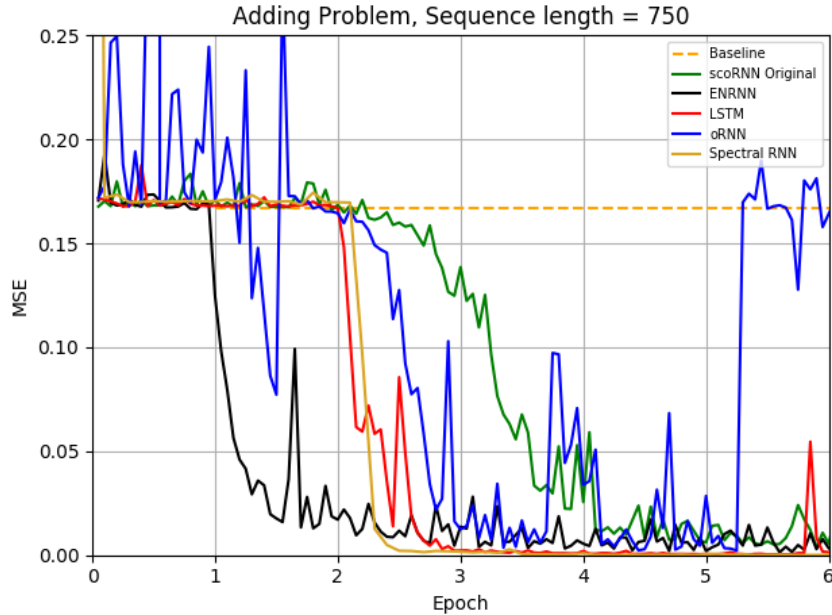


Figure 2.12: Test set MSE for the adding problem with sequence length of $T = 750$.

model used an RMSProp optimizer with learning rate 10^{-4} for the recurrent weight and an Adam optimizer with learning rate 10^{-3} for all other weights and 119 negative ones. The best hyperparameters for the oRNN were in accordance with [35] with 16 reflections and an Adam optimizer with learning rate 10^{-2} , $\approx 2.6k$ parameters. The Full-Capacity uRNN used an RMSProp optimizer with learning rate 10^{-5} . Figure 2.12 presents the convergence plots for 6 epochs. ENRNN converges towards 0 MSE before all other models with Spectral RNN asymptotically achieving a slightly lower MSE with learning rate decay.

Copying Problem

The copying problem has also been used to test many orthogonal/unitary RNNs [1, 14, 22, 35, 48]. In this experiment, a sequence of digits is passed into the RNN with the first 10 digits uniformly sampled from the digits 1 through 8 followed by the marker digit 9, a sequence of T zeros, and another marker digit 9. The network is to output the first 10 digits in the sequence once it sees the second marker 9, forcing the network to remember the original digits over the entire sequence. The total sequence length is $T + 20$. The cross-entropy loss function is used. The training and test sets were 20,000 and 1,000 sequences, respectively. Each model was trained for 4,000 iterations with batch size 20. The baseline for this task is the expected cross-entropy of randomly selecting digits 1-8 after the last marker 9, $\frac{10 \log(8)}{T+20}$.

The setup of this experiment is the same as described in Section 2.5. The hidden sizes for each model were adjusted so that they each had $\approx 22k$ trainable parameters. This resulted in a hidden size of $n = 192, 68, 190$, and 128 for the ENRNN, LSTM, scoRNN, and Full-Capacity uRNN respectively. The ENRNN had an $h^{(L)}$ and $h^{(S)}$

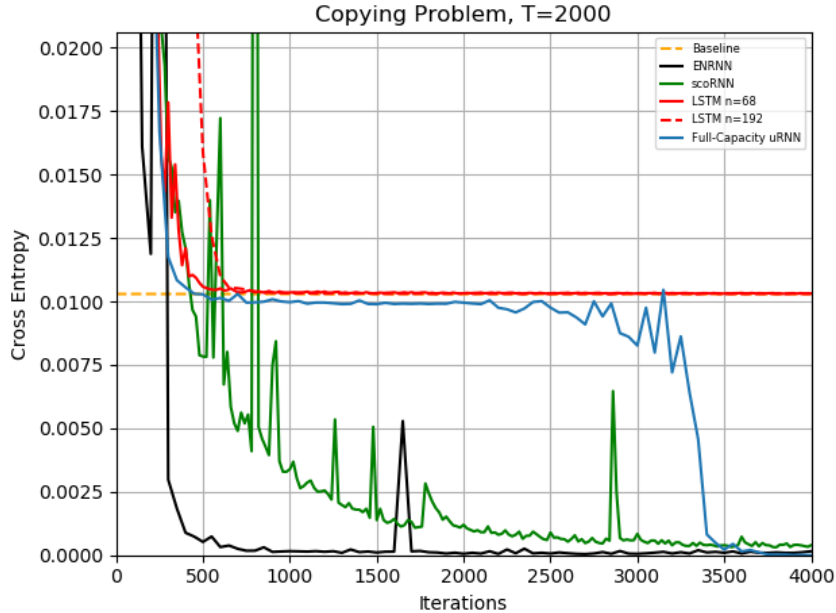


Figure 2.13: Cross-entropy for the copying problem with sequence length of $T = 2000$.

of size 172 and 20 with a coupling matrix $W^{(C)}$, see (2.82). An RMSProp optimizer was used with a learning rate of 10^{-5} for $h^{(L)}$ and learning rate of 10^{-3} for all other weights. The $W^{(L)}$ was parameterized with 52 negative ones. The LSTM used an RMSProp optimizer with learning rate 10^{-3} with an initial forget gate bias of 1.0 for $n = 68$. As per [14], the scoRNN model used an RMSProp optimizer with learning rate 10^{-4} for the recurrent weights and 10^{-3} for all other weights. The recurrent weight was parameterized with 95 negative ones. The Full-Capacity uRNN used an RMSProp optimizer with learning rate 10^{-3} .

Figure 2.13 plots cross-entropy values for 4000 iterations. As a reference, the LSTM was also run with the same hidden size of ENRNN, $n = 192$ with an initial forget gate bias of -2 , which has ≈ 7 times more trainable parameters than ENRNN and is still unable to drop below the baseline. Again, ENRNN outperforms other methods.

TIMIT

The setup of this experiment is the same as described in Section 2.5 except the MSE for all models was computed the same by taking the squared difference between the predicted and actual log magnitudes and applying a mask to zero out padded entries before computing the batch mean. In addition, the networks were trained for 300 epochs instead of 200 epochs. The hidden sizes of each model were adjusted so they each had $\approx 200k$ trainable parameters. This results in hidden sizes of $n = 468, 158,$ and 425 for ENRNN, LSTM, and scoRNN respectively. The ENRNN consisted of $h^{(L)}$ and $h^{(S)}$ of sizes 374 and 94 respectively with no coupling matrix. The number of negative ones for $W^{(L)}$ was 374. An Adam optimizer with learning rate 10^{-4} was used

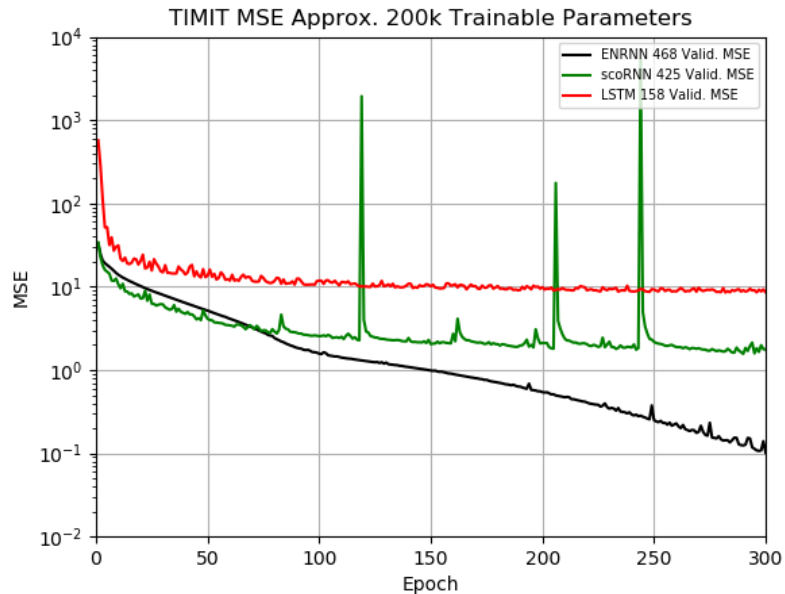


Figure 2.14: Validation set MSE for the TIMIT problem

for all weights. The number of negative ones for $W^{(L)}$ was 374. The LSTM used an RMSProp optimizer with learning rate 10^{-3} with gradient clipping of 1.0 and forget gate bias initialization of -4.0 . As per [14], the scoRNN used an RMSProp optimizer with learning rate 10^{-4} to update the recurrent matrix and an Adam optimizer with learning rate 10^{-3} for all other weights. The number of negative ones for the recurrent weight was 43.

Each model was trained for 300 epochs. Table 2.4 reports the results of the best epoch in validation MSE scores and Figure 2.14 plots convergence of these scores. As a secondary measure, we also show in Table 2.4 scores of three perceptual metrics. As a reference, the LSTM was also run with the same hidden size of ENRNN, $n = 468$, which has ≈ 6 times more trainable parameters than ENRNN and achieves worse scores except for PESQ where it is the same. Again, ENRNN significantly outperforms scoRNN and LSTM in the validation and testing MSEs and produces the overall best scores in the perceptual metrics.

Character PTB

The models were also tested on a character prediction task using the Penn Treebank Corpus [34]. The dataset consists of a word vocabulary of 10k with all other words marked as $\langle \text{unk} \rangle$, resulting in a total of 50 characters with the training, validation, and test sets consisting of approximately 5102k, 400k, and 450k respective characters. The batch size was set to 32. Due to the length of each sequence, the sequences were unrolled in length of 50 steps. Each sequence is fed into RNNs and the output is the same sequence shifted forward by one step to predict the next character. At the end of training of each sequence in a batch, the final hidden state is passed onto the next

Table 2.4: TIMIT: Best validation MSE after 300 epochs with test MSE and perceptual metrics. N - dimension of h (for ENRNN, dimensions of $h^{(L)}/h^{(S)}$) that match $\approx 200k$ trainable parameters.

Model	n	#Params	Valid. MSE	Test. MSE
ENRNN	374/94	$\approx 200k$	0.13	0.13
scoRNN	425	$\approx 200k$	1.56	1.52
LSTM	158	$\approx 200k$	8.53	8.27
LSTM	468	$\approx 1200k$	5.60	5.42
Model	N	SegSNR (dB)	STOI	PESQ
ENRNN	374/94	4.84	0.83	2.75
scoRNN	425	4.55	0.82	2.72
LSTM	158	4.00	0.79	2.51
LSTM	468	4.82	0.81	2.75

training sequence as the initial state. We use a linear embedding layer that maps each input character to R^N (N being the hidden state dimension). The loss function was cross-entropy. We report the customary performance metrics of bits-per-character (bpc) which is the cross-entropy loss with the natural logarithm replaced by the base 2 logarithm.

The hidden size of each model was adjusted to match the number of trainable parameters, $\approx 1016k$. It is 350 for LSTM and 1030 for ENRNN with an $h^{(L)}$ and $h^{(S)}$ of sizes 310 and 720. The ENRNN uses a coupling matrix, $W^{(C)}$, with a truncated orthogonal initialization, a fixed input weight matrix set to identity, and ReLU non-linearity. An Adam optimizer was used with learning rate of 10^{-4} to update $W^{(L)}$ and 10^{-3} for all other weights. The $W^{(L)}$ had 186 negative ones. The LSTM uses a forget gate bias initialization of 0.0 and gradient clipping of 8.

We report the best results after 20 epochs training in Table 2.5. Also included in the table are the results from [21, 25, 35] for the same problem. As a reference, the LSTM was also run with the same hidden size of ENRNN, $n = 1030$, which results in a better score but requires ≈ 8.5 times more trainable parameters than ENRNN. We see that ENRNN slightly outperforms LSTM when matching the number of trainable parameters and all other models. To test the capability of the ENRNN to be stacked in a multilayer fashion, we also conducted an experiment on the character prediction task using a 3 layer ENRNN and LSTM. The ENRNN and LSTM had the same hidden sizes as in the single layer model, but with learning-rate decay and dropout applied between each layer. For the 3 layer experiment, the sequence was unrolled 100 steps and a batch size of 64 was used. The ENRNN achieved a validation BPC of 1.330 and a test BPC of 1.297 and the LSTM achieved a validation BPC of 1.328 and a test BPC of 1.293.

Table 2.5: Character PTB: Best testing MSE in BPC after 20 epochs. N - dimension of h (for ENRNN, dimensions of $h^{(L)}/h^{(S)}$). Entries marked by an asterix (*), (**), and (***) are reported from [21], [35], and [25], resp.

Model	n	# Param	Valid. BPC	Test BPC
ENRNN	310/720	\approx 1016k	1.475	1.429
LSTM	350	\approx 1016k	1.506	1.461
GRU	415	-	-	1.601*
EURNN	2048	-	-	1.715*
GORU	512	-	-	1.623*
oRNN	512	\approx 183k	1.73**	1.68**
nnRNN	1024	\approx 1320k	-	1.47***
LSTM	1030	\approx 8600k	1.447	1.408

Gradient Analysis

For each pair of time steps $t \leq \tau$, we use $\|\frac{\partial h_\tau^{(S)}}{\partial x_t}\|_2$ and $\|\frac{\partial h_\tau^{(L)}}{\partial x_t}\|_2$ to measure the dependency of $h_\tau^{(S)}$ and $h_\tau^{(L)}$ at time τ respectively on the input x_t at time t . We consider a small Adding Problem of sequence length $T = 50$ using an ENRNN of hidden size $n = 40$ with $h^{(L)}$ block size of 24 and $h^{(S)}$ block size of 16. We compute the gradient norms over the first random mini-batch at the beginning of the sixth epoch and plot them as a heat map in Figure 2.15 and 2.16. Here the x-axis is the input time step (going from left to right) and the y-axis is the hidden state time step (going from top to bottom).

As can be seen, the short-term state gradient (Figure 2.15) diminishes quickly as τ increases from t , demonstrating the short-term dependency of $h_\tau^{(S)}$. On the other hand, the long-term state gradient (Figure 2.16) may stay large for all τ showing long-term dependency of $h_\tau^{(L)}$. Of particular note, there appears to be a few vertical lines that have higher gradient norms relative to other input steps. It appears that these inputs have a greater effect on the model than others.

Hidden State Sizes

In this section, we explore the effect of different short-term hidden states on model performance on the adding problem using similar settings as discussed in Section 2.5. In Figure 2.17, we keep the $h^{(L)}$ state size fixed at $n = 96$ and adjust the $h^{(S)}$ state size from 0 to 96 for testing sensitivity. In Figure 2.18, we keep the total hidden state size fixed at $n = 160$ and adjust the $h^{(L)}$ and $h^{(S)}$ sizes. In addition, we run the experiment with no $h^{(L)}$ and no $h^{(S)}$. As can be seen, having no long-term memory state, $h^{(L)} = 0$, the ENRNN is unable to approach zero MSE and having no short-term memory state, $h^{(S)} = 0$, the ENRNN is only able to pass the baseline after almost 6 epochs, if at all. In general, as the size of $h^{(S)}$ increases, the performance

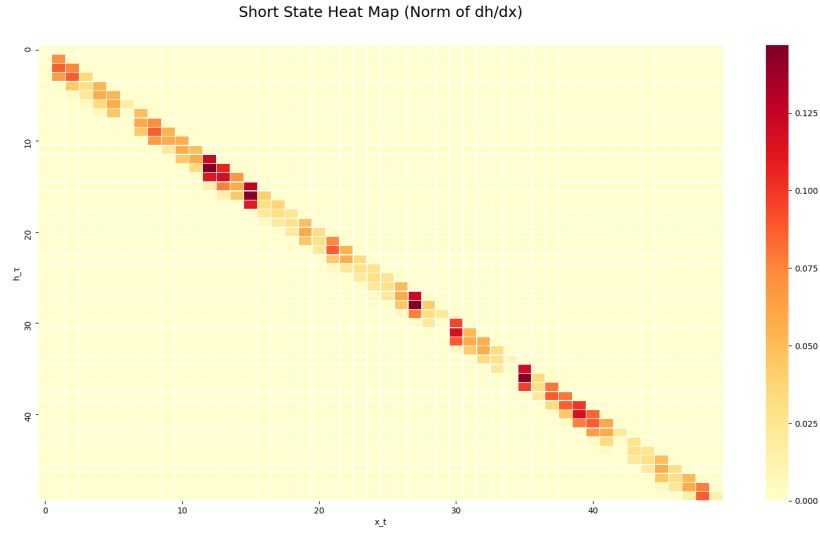


Figure 2.15: Gradient norms $\|\frac{\partial h_\tau^{(S)}}{\partial x_t}\|$. The x -axis is t from left to right and y -axis is τ from top to bottom. The column at t shows dependence of states $h_\tau^{(S)}/h_\tau^{(L)}$ on x_t .

increases with optimal performance occurring around $h^{(S)} = 64$ and $h^{(L)} = 96$ or $h^{(S)} = 96$ and $h^{(L)} = 64$. It should be noted that for the large range $h^{(S)} > 38$, near optimal performance is achieved in Figure 2.17.

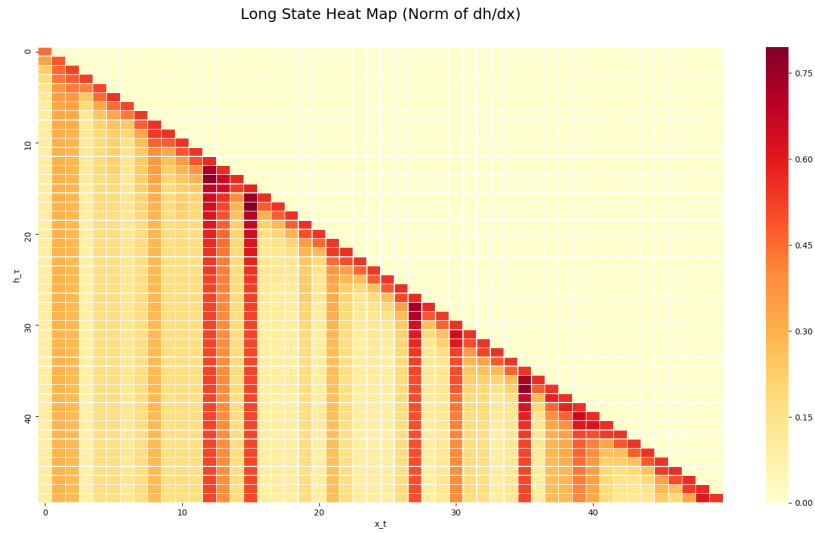


Figure 2.16: Gradient norms $\|\frac{\partial h_t^{(L)}}{\partial x_t}\|$. The x -axis is t from left to right and y -axis is τ from top to bottom. The column at t shows dependence of states $h_t^{(S)}/h_t^{(L)}$ on x_t .

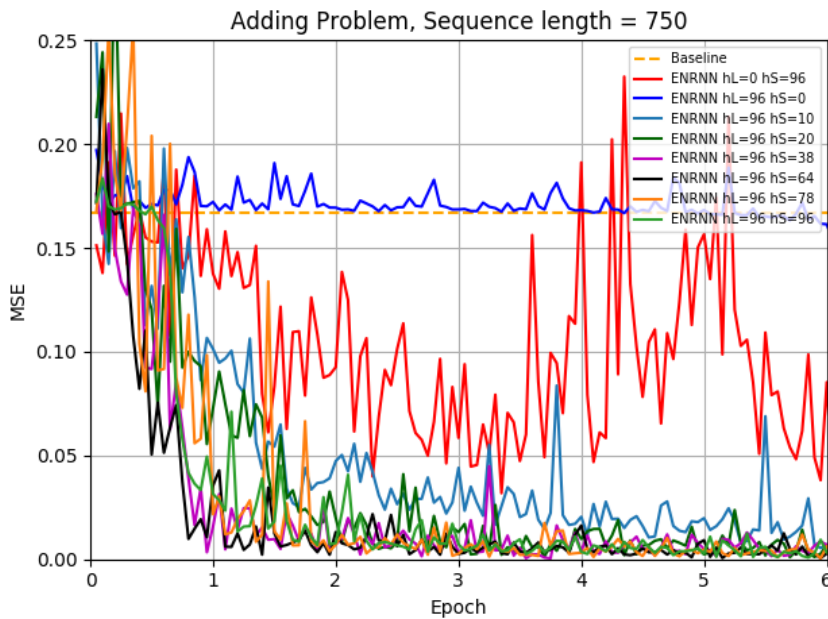


Figure 2.17: Test set MSE for the ENRNN on the adding problem with sequence length of $T = 750$ with various short-term hidden state sizes $h^{(S)}$.

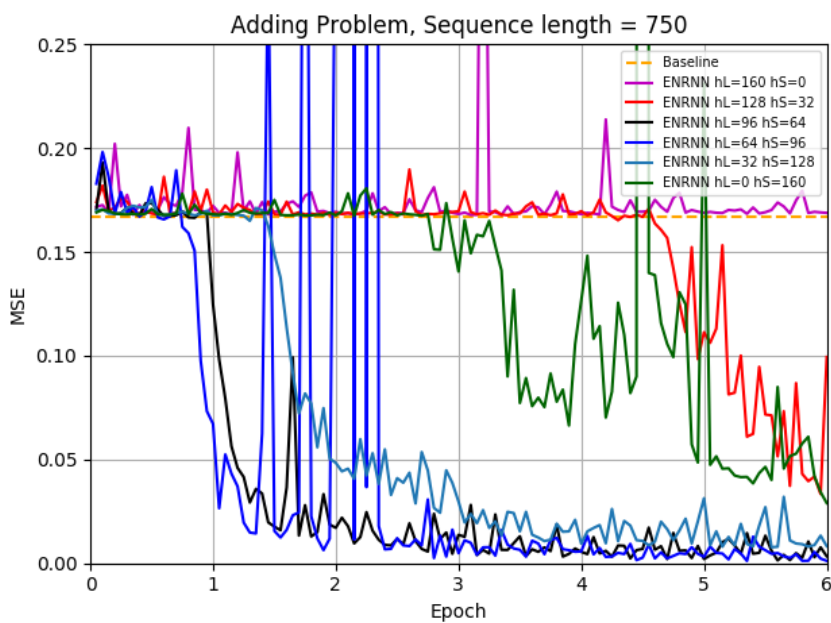


Figure 2.18: Test set MSE for the ENRNN on the adding problem with sequence length of $T = 750$ with fixed hidden state size of 160 and various short-term and long-term hidden state sizes $h^{(S)}$ and $h^{(L)}$.

Chapter 3 Batch Normalization

One of the major obstacles in training FFNs, see Section 1.3, is known as **internal covariate shift**. As speculated in the seminal work by [20], training of a deep FFN can be difficult since a small update in a parameter can result in a significant change in the input received at a much later layer in the network. In a particular update step, a layer will receive an input following a certain distribution and the layer parameters will be updated accordingly in the steepest gradient descent direction using SGD or some other gradient descent based algorithm. After the update step, the effect of these small parameter changes can accumulate and can cause a drastic shift of the distribution received by later layers. This constant shifting of distributions diminishes the effectiveness of SGD since the steepest descent direction can be continuously changing after each iteration. In an attempt to maintain a consistent distribution for each layer, **batch normalization** was introduced [20].

In batch normalization, a linear layer is inserted between two hidden layers to center and scale the input to have zero mean and unit variance. Since it might not be necessarily the case that the optimal input distribution should have a zero mean or unit variance, the additional trainable parameters γ and β are introduced, see Algorithm 4. The trainable γ parameters are used to rescale the input and the β parameters are used to shift the input where needed.

If we denote the batch normalization layer as $\mathcal{B}(\cdot)$, the typical application of batch normalization between two layers is

$$H^{(k)} = \sigma \left(W^{(k)} \mathcal{B} \left(H^{(k-1)} \right) + b^{(k)} \right) \quad (3.1)$$

where the previous layer output is $H^{(k-1)}$ which is a matrix consisting of N mini-batches with each column a single training example, the weights associated with the k th layer is $W^{(k)}$, and the bias associated with the k th layer is $b^{(k)}$. As originally introduced by [20], batch normalization can alternatively be applied before the non-linearity

$$H^{(k)} = \sigma \left(\mathcal{B} \left(W^{(k)} H^{(k-1)} \right) \right) \quad (3.2)$$

In this case, there is no need for the bias term as the centering step will negate any

effects from the bias. In practice, (3.1) is more commonly used than (3.2).

Algorithm 4: Batch Normalization (Training)

Given:

Re-scaling parameter $\gamma \in \mathbb{R}^n$

Re-shifting parameter $\beta \in \mathbb{R}^n$

Small $\epsilon > 0$

Input:

Batch of N examples, each with n features $A = [a_1, a_2, \dots, a_N] \in \mathbb{R}^{n \times N}$

Procedure:

Mini-batch mean: $\mu_A \leftarrow \frac{1}{N} \sum_{i=1}^N a_i \in \mathbb{R}^n$

Mini-batch variance: $\sigma_A^2 \leftarrow \frac{1}{N} \sum_{i=1}^N (a_i - \mu(A))^2 \in \mathbb{R}^n$ (square is elementwise)

Centering and scaling: $Z \leftarrow \text{diag} \left(\frac{1}{\sqrt{\sigma_A^2 + \epsilon}} \right) (A - \mu_A 1_N^T) \in \mathbb{R}^{n \times N}$

Re-scaling/Re-shift: $H \leftarrow \text{diag}(\gamma)Z + \beta 1_N^T$

Output:

$\mathcal{B}(A) := H$

Note that Algorithm 4 is used only during training since the centering and scaling operations are based on mini-batch statistics. During inference, if the batch mean and variance from the given mini-batch is used, the model predictions will vary based on how the batches are shuffled which can result in different predictions for the same example. It should also be noted that Algorithm 4 is not designed to work on a single example because it requires the computation of the mean and variance.

To perform inference, a fixed mean and variance should be used. As first proposed by [20], the mean and variance used during inference are simply the average of the mini-batch means and variances. Given x minibatches with individual batch mean and variances consisting of $\{\mu_1, \mu_2, \dots, \mu_x\}$ and $\{\sigma_1^2, \sigma_2^2, \dots, \sigma_x^2\}$ respectively, the mean and variance used during inference are computed as

$$\mu \leftarrow \frac{1}{x} \sum_{i=1}^x \mu_i \tag{3.3}$$

$$\sigma^2 \leftarrow \frac{1}{x-1} \sum_{i=1}^x \sigma_i^2 \tag{3.4}$$

Note that the unbiased estimator is used when computing the variance. Batch normalization during inference is identical to Algorithm 4 except Z is computed using the μ and σ^2 from (3.3) and (3.4).

In practice, the running mean and variance are used in place of (3.3) and (3.4). To use the running mean and variance, the running mean is initialized to be all zeros and the running variance is initialized as all ones. If we denote the current minibatch mean as μ_B and current minibatch variance as σ_B^2 , the running mean and variance updates are

$$\mu \leftarrow \rho \mu + (1 - \rho) \mu_B \tag{3.5}$$

$$\sigma^2 \leftarrow \rho \sigma^2 + (1 - \rho) \sigma_B^2 \tag{3.6}$$

Here ρ is the momentum term with a typical value of 0.99. Note that the running mean and variance are only used during inference.

Since the introduction of batch normalization, it has grown in popularity and is commonly used in different architectures. The primary advantage of using batch normalization is that it can decrease training times by improving initial results. Batch normalization also allows the use of larger learning rates and is more robust to different learning rates and initializations, making the selection of the learning rate and weight initialization less critical. Finally, batch normalization tends to create a regularization effect by preventing overfitting and improving generalization.

3.1 Related Work

Although found to be quite successful, batch normalization is very little understood with many papers analyzing different aspects of batch normalization. Recent work include [32] where different moments are used in computing the moving average and a convergence analysis of batch normalization is provided. The work by [42] claim that batch normalization does not improve training because it reduces internal covariate shift, but it improves training times due to a smaller bound on the weight gradients. In [4], experimental results are presented that also confirm batch normalization decreases training time by allowing larger learning rates. The paper [7] analyzes batch normalization on the ordinary least squares problem and claim that the additional parameters γ and β improve the condition number of the Hessian matrix and that batch normalization is less sensitive to learning rate selection and thus has faster convergence. In [28], optimization theory is used to prove that batch normalization can accelerate convergence within certain assumptions. The paper [31] analyzes how batch normalization tends to perform poorly on smaller batch sizes and also claim that batch normalization performs well by improving the condition number of the loss function. They also provide a different algorithm to update the batch statistics using compositional optimization. The work by [50] establishes a mean field theory of batch normalization and claim that batch normalization may actually result in unbounded gradients as depth of the network increases.

3.2 Issues with Batch Normalization

There are still several issues that need to be resolved concerning batch normalization. The main issue is that despite numerous work analyzing batch normalization, see Section 3.1, it is still not well understood and there is no consensus on why it works. From this lack of understanding, there is a general confusion on how to implement batch normalization. In most cases, batch normalization is used after the nonlinearity as a separate layer, see (3.1), instead of before the nonlinearity, see (3.2). Another source of confusion is whether to use the running batch statistics during inference, see (3.5) and (3.6), or the average of the batch statistics, see (3.3) and (3.4). We should note that using either the running batch statistics or the average of the batch statistics during inference may negatively impact training. During training

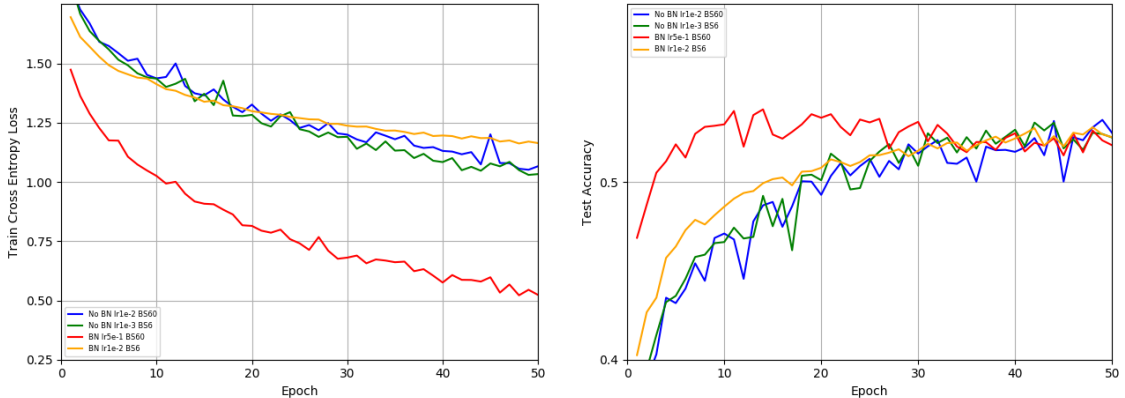


Figure 3.1: Training set cross-entropy loss (left) and test accuracy (right) using batch normalization on the CIFAR10 dataset. The network is a simple 3 hidden layer network of size 100 per layer plus one output layer of size 10. The SGD optimizer is used with batch normalization applied before each layer using running batch statistics during inference. Models trained using a batch size of 60 and 6 are denoted BS60 and BS6 respectively. The learning rate was optimized for both batch sizes.

the gradients are allowed to propagate through the batch normalization layer but the resulting gradient descent direction is dependent on the current batch statistics and not the statistics used during inference. Thus the gradients might not be in the steepest descent direction with respect to the inference model.

In Figure 3.1 we plot the convergence curves of a batch normalized FFN and a plain vanilla FFN without batch normalization. Note that we did not regularize these networks to prevent overfitting, but simply optimized over their learning rates to obtain the highest test accuracy possible. As can be seen, the performance of the batch normalized network on the training data set and the initial performance on the test data set decreases as the batch size decreases. Despite this, the batch normalized FFN with smaller batch size 6 does not overfit near the end of training and eventually performs similar to the vanilla FFN with batch size 6 despite having a much worse training loss. This suggests that batch normalization may improve the generalization of the network. For batch size 60, the batch normalized FFN has a much lower training loss curve and achieves a larger test accuracy much earlier than all the other networks, indicating how batch normalization can decrease training times. Note the batch size 60 batch normalized network still overfits. Finally, notice how the optimal batch normalized networks have larger learning rates. Currently, there is no consensus why the batch normalized networks behave this way.

3.3 Batch Normalized Preconditioning

In our current work, we are devising an alternative algorithm to batch normalization. We call the proposed method **Batch Normalized Preconditioning (BNP)**. The main idea behind BNP is that instead of applying normalization explicitly through

a batch normalization layer, normalization is applied by conditioning the parameter gradients directly during training. This preconditioning of the gradients is expected to improve the Hessian matrix of the loss function and thus improve convergence of the parameters during training. In BNP, there is no need to maintain running statistics or the need of additional trainable parameters. The effects of centering and scaling steps are incorporated into the update of the parameters.

Rate of Convergence and the Hessian

In a FFN, weights and biases are updated by gradient descent as discussed in Section 1.6. In particular, given the parameter vector θ and some differentiable loss function L with respect to θ , the parameters are iteratively updated by (1.24) where λ is the learning rate or step size and k is the k th iteration. Now suppose L has continuous second partial derivatives, θ^* is a local minimizer, and consider an initialization of θ in a small neighborhood around θ^* . We define $g(\theta) = \theta - \alpha \nabla_{\theta} L(\theta)$ where α is some learning rate and note that θ^* is a fixed point of g . Using g to iteratively define each θ_k , by Taylor's Theorem about the point θ^*

$$\theta_{k+1} = g(\theta_k) \quad (3.7)$$

$$= \theta^* - \nabla_{\theta} g(\xi)(\theta_k - \theta^*) \quad (3.8)$$

$$= \theta^* + (I - \nabla_{\theta}^2 L(\xi))(\theta_k - \theta^*) \quad (3.9)$$

for some ξ . Rearranging (3.9) and using the 2-norm we obtain

$$\|\theta_{k+1} - \theta^*\|_2 \leq \|I - \alpha \nabla_{\theta}^2 L(\xi)\|_2 \|\theta_k - \theta^*\|_2 \quad (3.10)$$

Let λ_{\min} and λ_{\max} be the minimum and maximum eigenvalues of the Hessian matrix. Since $I - \alpha \nabla_{\theta}^2 L(\xi)$ is real valued and symmetric and thus normal we can rewrite (3.10)

$$\|\theta_{k+1} - \theta^*\|_2 \leq \rho \|\theta_k - \theta^*\|_2 \quad (3.11)$$

where $\rho = \max\{|1 - \alpha \lambda_{\min}|, |1 - \alpha \lambda_{\max}|\}$. For simplicity of analysis, let us assume $\lambda_{\min} > 0$. Now ρ is minimized with respect to α when

$$\alpha = \frac{2}{\lambda_{\min} + \lambda_{\max}} \quad (3.12)$$

In this case we have either $\rho = 1 - \alpha \lambda_{\min}$ or $\rho = -1 + \alpha \lambda_{\max}$. Taking either case will result in the same value of ρ . For the first case, we obtain

$$\rho = 1 - \frac{2\lambda_{\min}}{\lambda_{\max} + \lambda_{\min}} \quad (3.13)$$

$$= \frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\max} + \lambda_{\min}} \quad (3.14)$$

$$= \frac{\frac{\lambda_{\max}}{\lambda_{\min}} - 1}{\frac{\lambda_{\max}}{\lambda_{\min}} + 1} \quad (3.15)$$

$$= \frac{\kappa - 1}{\kappa + 1} \quad (3.16)$$

where κ is the condition number of the Hessian matrix. The inequality (3.11) and the optimal value of ρ in (3.16) show that the rate of convergence is dependent on the condition number of the Hessian matrix. In the case where $\kappa \gg 1$, the optimal α will be approximately $\frac{2}{\lambda_{\max}}$ and convergence will be slow. Any learning rate larger than this α will result in divergence. Thus finding the optimal learning rate can be difficult.

FFN and the Hessian

As discussed in [7, 31] and in the previous section, convergence of a FFN may be dependent on the condition number of the Hessian of the loss function. Unfortunately, an explicit derivation of the Hessian is rarely provided due to the complicated structure of the general Hessian. In most cases, it is rarely computed because it is too computationally expensive to do so. The only work we are aware of that tries to explicitly derive the Hessian is [38].

In the theorem below, we derive the gradient and Hessian of the loss with respect to a single weight vector and single bias entry associated with a single layer state. Let $h^{(k)} = \sigma(a^{(k)}) \in \mathbb{R}^n$ be the hidden variable of layer k , i.e. the output of the k th layer of a FFN where $a^{(k)} = W^{(k)}h^{(k-1)} + b^{(k)} \in \mathbb{R}^n$, $h^{(k-1)} \in \mathbb{R}^m$ is the hidden variable of layer $k-1$, and $W^{(k)} \in \mathbb{R}^{n \times m}$ and $b^{(k)} \in \mathbb{R}^n$ the respective weight matrix and bias vector of the k th layer. We denote $h_i^{(k)} = \sigma(a_i^{(k)}) \in \mathbb{R}$ as the i th entry of $h^{(k)}$ where $a_i^{(k)} = w_i^{(k)T} h^{(k-1)} + b_i^{(k)} \in \mathbb{R}$. Here $w_i^{(k)T} \in \mathbb{R}^{1 \times m}$ and $b_i \in \mathbb{R}$ are the respective i th row and entry of $W^{(k)}$ and $b^{(k)}$. To simplify notation, let $\hat{w}^T = [b_i^{(k)}, w_i^{(k)T}] \in \mathbb{R}^{1 \times (m+1)}$ and $\hat{h}^T = [1, h^{(k-1)T}] \in \mathbb{R}^{1 \times (m+1)}$ so that $a_i^{(k)} = \hat{w}^T \hat{h}$. Using this notation, we present the gradient and the Hessian of a loss function with respect to \hat{w} in Theorem 3.3.1. Note that if \hat{H} in Theorem 3.3.1 is ill conditioned then the Hessian is ill conditioned.

Theorem 3.3.1. *Consider a differentiable FFN loss function $L = L(a_i^{(k)}) = L(\hat{w}^T \hat{h})$ written as a function of \hat{w} through $a_i^{(k)}$ for a single training example x . When training over a mini-batch with N inputs $\{x_1, x_2, \dots, x_N\}$, let $\{h_1^{(k-1)}, h_2^{(k-1)}, \dots, h_N^{(k-1)}\}$ be the associated hidden variables of layer $k-1$ and let $H = [h_1^{(k-1)}, h_2^{(k-1)}, \dots, h_N^{(k-1)}] \in \mathbb{R}^{m \times N}$. The gradient and Hessian with respect to \hat{w} of the mean total loss over the entire batch $\mathcal{L} := \frac{1}{N} \sum_{j=1}^N L(\hat{w}^T \hat{h}_j)$, where $\hat{h}_j^T = [1, h_j^{(k-1)T}] \in \mathbb{R}^{1 \times (m+1)}$, are given by*

$$\nabla_{\hat{w}} \mathcal{L} = \frac{1}{N} \sum_{j=1}^N L'(\hat{w}^T \hat{h}_j) \hat{h}_j \quad (3.17)$$

$$\nabla_{\hat{w}}^2 \mathcal{L} = \hat{H}^T S \hat{H} \quad (3.18)$$

where $\hat{H}^T = [\hat{h}_1, \hat{h}_2, \dots, \hat{h}_N] \in \mathbb{R}^{(m+1) \times N}$, $S = \frac{1}{N} \text{diag}([s_1, s_2, \dots, s_N])$, and $s_j = L''(\hat{w}^T \hat{h}_j)$.

Proof. Taking the gradient of \mathcal{L} with respect to \hat{w} ,

$$\nabla_{\hat{w}}\mathcal{L} = \frac{1}{N} \sum_{j=1}^N \mathcal{L}'(\hat{w}^T \hat{h}_j) \hat{h}_j \quad (3.19)$$

Differentiating Equation (3.19) again we obtain the desired Hessian

$$\begin{aligned} \nabla_{\hat{w}}^2\mathcal{L} &= \frac{1}{N} \sum_{j=1}^N L''(\hat{w}^T \hat{h}_j) \hat{h}_j \hat{h}_j^T \\ &= \hat{H}^T S \hat{H} \end{aligned} \quad (3.20)$$

□

Preconditioned Gradient Descent

In the proposed BNP neural network, we consider a change of variable $\theta = Mz$ which we call a **preconditioning** transformation. If we consider \mathcal{L} as a function of Mz , the gradient descent Equation (1.24) becomes

$$z_{k+1} = z_k - \alpha \nabla_z \mathcal{L}(Mz_k) = z_k - \alpha M^T \nabla_{\theta} \mathcal{L}(\theta_k) \quad (3.21)$$

Using a similar argument as used to derive (3.11), we again obtain a convergence bound of $\|z_{k+1} - z^*\| \leq \rho \|z_k - z^*\|$ where z^* is some local minimum and ρ is dependent on the condition number of $\nabla_z^2 \mathcal{L}(Mz) = M^T \nabla_{\theta}^2 \mathcal{L}(\theta) M$. Now if M is such that $M^T \nabla_{\theta}^2 \mathcal{L}(\theta) M$ has a better condition number than $\nabla_{\theta}^2 \mathcal{L}(\theta)$, then ρ is reduced and convergence is accelerated. Multiplying both sides of (3.21) by M we obtain the equivalent update scheme which is the update step used in BNP.

$$\theta_{k+1} = \theta_k - \alpha M M^T \nabla_{\theta} \mathcal{L}(\theta_k) \quad (3.22)$$

In order to improve the conditioning of the Hessian matrix, we define $M := PD$. The matrices P and D are based on the current mini-batch statistics with P acting as a centering matrix and D as a scaling matrix. Let $\mu^T = [\mu_1^{(k)}, \mu_2^{(k)}, \dots, \mu_m^{(k)}] \in \mathbb{R}^{1 \times m}$ and $s^T = [s_1^{(k)}, s_2^{(k)}, \dots, s_m^{(k)}] \in \mathbb{R}^{1 \times m}$ be the vectors consisting of the respective computed means and variances over each of the m features of the mini-batch coming from the previous layer. The scaling matrices of the k th layer are given as (3.23).

$$P = \begin{bmatrix} 1 & -\mu^T \\ 0 & I \end{bmatrix}; D = \text{diag} \left(\left[1, \frac{1}{\sqrt{s^T + \epsilon 1^T}} \right] \right) \quad (3.23)$$

where the division and square root in D are applied elementwise. Note that in practice, we add a small number to the entries of s to avoid division by zero. Using Theorem 3.3.1, the Hessian matrix becomes

$$\nabla_z^2 \mathcal{L}(Mz) = M^T \nabla_{\theta}^2 \mathcal{L}(\theta) M = D^T P^T \hat{H}^T S \hat{H} P D \quad (3.24)$$

We note by [13], multiplying \hat{H} by P improves the condition number and

$$\hat{H}P = [1_N, H^T] P = [1_N, H^T - 1_N \mu^T] \quad (3.25)$$

The additional multiplication by D further reduces the condition number by a theorem of van der Sluis [16] which states

$$\kappa_2(\hat{H}PD) \leq \sqrt{m+1} \min_{D_0 \text{ is diagonal}} \kappa_2(\hat{H}PD_0) \quad (3.26)$$

Note that if D_0 is singular, and we define the condition number of singular matrices to be infinity, then D_0 does not need to be specified as non-singular. See [13] for more details.

For a layer where BNP is to be applied, at each training step we first use the standard backpropagation algorithm to compute $\frac{\partial \mathcal{L}}{\partial W}$ and $\frac{\partial \mathcal{L}}{\partial b}$ and concatenate these gradients. We then compute the mean and variances across each feature of the mini-batch from the previous layer and update the layer weights and biases by (3.22) where $M = PD$. In particular,

$$\begin{bmatrix} b_{k+1}^T \\ W_{k+1} \end{bmatrix} \leftarrow \begin{bmatrix} b_k^T \\ W_k \end{bmatrix} - \lambda PD^2 P^T \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial b_k} \\ \frac{\partial \mathcal{L}}{\partial W_k} \end{bmatrix} \quad (3.27)$$

Note that the preconditioning matrices are only used during gradient descent and are only constructed during training and are not stored in memory. Unlike batch normalization, there are no explicit normalization layers. For inference, BNP proceeds like a normal FFN and does not require application of any scaling matrix.

3.4 Experiments

In the following experiments, we compare a BNP FFN with a batch normalized FFN and a vanilla FFN. For each experiment, we found optimal hyperparameters for each model using a grid search. For the batch normalized FFN, we apply batch normalization before each layer as in (3.1) and use running statistics during inference as in (3.5) and (3.6) with $\rho = 0.99$ unless noted otherwise. For the BNP FFN, we use batch statistics for the mean and variance terms in (3.23) and apply BNP to the gradients of each layer. We add ϵ to the variance vector to avoid division by zero. Each network initialization is Glorot Uniform [11] unless indicated otherwise and are trained using stochastic gradient descent. All networks consist of three layers with hidden sizes of 100 with an output layer with hidden size 10. Regularization to avoid overfitting is not applied unless noted.

CIFAR10

The **Canadian Institute for Advanced Research 10 (CIFAR-10)** database [27] consists of 60,000 colored images of pixel size 32 x 32 x 3. There are 10 different classes of images ranging from airplane to dog and to truck. See Figure 3.2 for

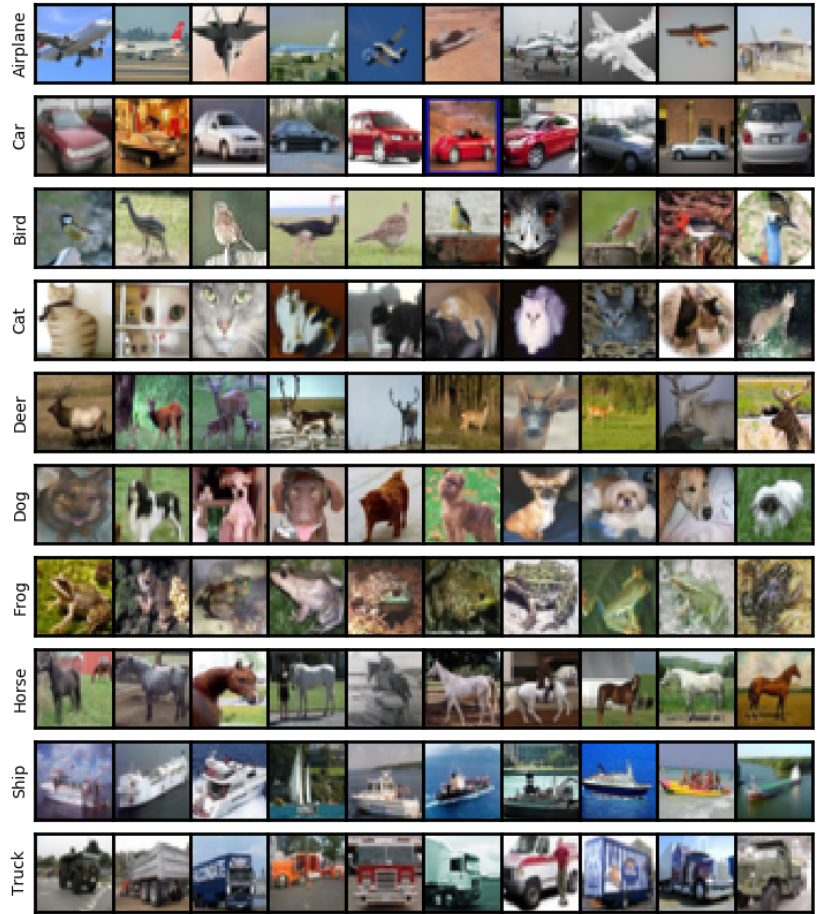


Figure 3.2: Example images from the CIFAR-10 data set.

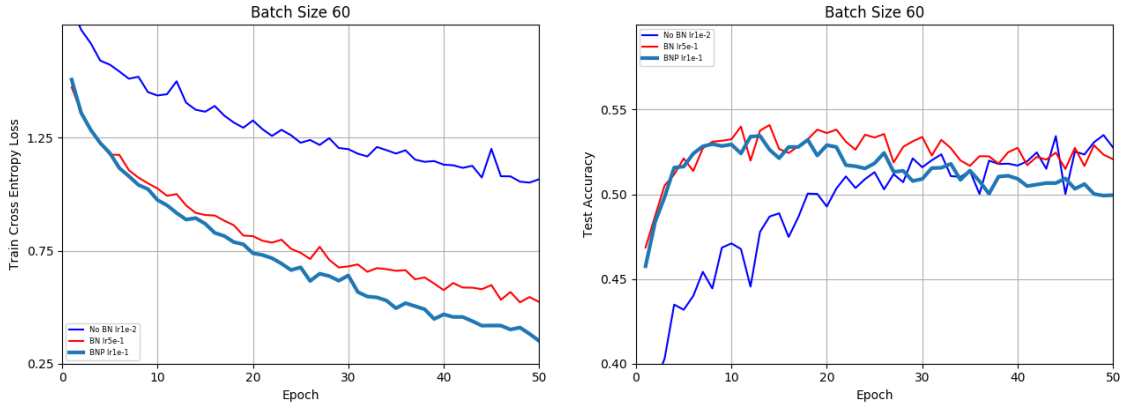


Figure 3.3: Training loss (left) and test loss (right) on the CIFAR-10 dataset.

example images. There are 6,000 images of each class and the dataset is split into a training set of 50,000 images and a test set of 10,000 images. For this experiment, each image is simply flattened into a vector of size 3,072 and fed into the network.

For this experiment, the weights in the BNP FFN were initialized using Glorot Uniform but scaled by 0.75 and the biases were initialized as 0. The ϵ in (3.23) was set to 0.5. Each network used ReLU. Results of the experiment are shown in Figure 3.3. Similar to batch normalization, results show that BNP decreases the training loss and increases the initial test accuracy as compared to a vanilla FFN. We should note the initial performance on test accuracy of the BNP FFN is comparable to the batch normalized FFN but tends to overfit more as training progresses. On the other hand, the BNP FFN is able to obtain a much lower train loss compared to all other models.

Bibliography

- [1] M. Arjovsky, A. Shah, and Y. Bengio. Unitary evolution recurrent neural networks. In *Proceedings of the 33rd International Conference on Machine Learning (ICML 2016)*, volume 48, pages 1120–1128, New York, NY, 2016. JMLR.
- [2] Y. Bengio, P. Frasconi, and P. Simard. The problem of learning long-term dependencies in recurrent networks. In *Proceedings of 1993 IEEE International Conference on Neural Networks (ICNN '93)*, pages 1183–1195, San Francisco, CA, 1993. IEEE Press.
- [3] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5:157–166, 1994.
- [4] J. Bjorck, C. Gomes, B. Selman, and K. Weignberger. Understanding batch normalization. In *Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS 2018)*, Montreal, Canada, 2018.
- [5] M. Brookes et al. Voicebox: Speech processing toolbox for matlab. *Software, available [Mar. 2011] from [www. ee. ic. ac. uk/hp/staff/dmb/voicebox/voicebox.html](http://www.ee.ic.ac.uk/hp/staff/dmb/voicebox/voicebox.html)*, 47, 1997.
- [6] J. Brown and R. Churchill. *Complex Variables and Applications*. McGraw-Hill Higher Education, 2009.
- [7] Y. Cai, Q. Li, and Z. Shen. A quantitative analysis of the effect of batch normalization on gradient descent. In *Proceedings of the 36th International Conference on Machine Learning*, Long Beach, California, 2019.
- [8] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches, 2014.
- [9] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research (JMLR)*, 12:2121–2159, 2011.
- [10] J. Garofolo, L. Lamel, W. Fisher, J. Fiscus, D. Pallett, N. Dahlgren, and V. Zue. Timit acoustic-phonetic continuous speech corpus ldc93s1. Technical report, Philadelphia: Linguistic Data Consortium, Philadelphia, PA, 1993.
- [11] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, volume 9, Sardinia, Italy, 2010. PMLR.

- [12] I. Goodfellow, Y. Bengion, and A. Courville. *Deep Learning*. The MIT Press, 2016.
- [13] K. Helfrich, S. Lange, and Q. Ye. Batch normalization preconditioning. *Work currently in progress*, 0, 2020.
- [14] K.E. Helfrich, D. Willmott, and Q. Ye. Orthogonal recurrent neural networks with scaled cayley transform. In D. Jennifer and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*, pages 1969–1978, Stockholmsmassan, Stockholm Sweden, 2018. PMLR.
- [15] M. Henaff, A. Szlam, and Y. LeCun. Recurrent orthogonal networks and long-memory tasks. In *Proceedings of the 33rd International Conference on Machine Learning (ICML 2017)*, volume 48, New York, NY, 2017. JMLR: W&CP.
- [16] N. Highman. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, 2002.
- [17] G. Hinton. Lecture 6e rmsprop: Divide the gradient by a running average of its recent magnitude. coursera, 2012.
- [18] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.
- [19] R. Hunger. An introduction to complex differentials and complex differentiability. Technical Report TUM-LNS-TR-07-06, Technische Universität München, Associate Institute for Signal Processing, July 2007.
- [20] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning (ICML32)*, volume 37, Lille, France, 2015. JMLR: W&CP.
- [21] L. Jing, C. Gülçehre, J. Peurifoy, Y. Shen, M. Tegmark, M. Soljačić, and Y. Bengio. Gated orthogonal recurrent units: On learning to forget. *Neural Computation*, 31(4):765–783, April 2019.
- [22] L. Jing, Y. Shen, T. Dubček, J. Peurifoy, S. Skirlo, M. Tegmark, and M. Soljačić. Tunable efficient unitary neural networks (eunn) and their application to rnn. *arXiv e-prints*, page arXiv:1612.05231, 2016.
- [23] L. Jing, Y. Shen, T. Dubcek, J. Peurifoy, S. Skirlo, M. Tegmark, and M. Soljadic. Tunable efficient unitary neural networks (EUNN) and their application to RNN. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*, volume abs/1612.05231, pages 1733–1741, International Convention Centre, Sydney, Australia, 2017. PMLR.

- [24] William Kahan. Is there a small skew cayley transform with zero diagonal? *Linear algebra and its applications*, 417(2-3):335–341, 2006.
- [25] G. Kerg, K. Goyette, M. Touzel, G. Gidel, E. Voronstov, Y. Bengio, and G. Lajoie. Non-normal recurrent neural network (nnrnn): Learning long time dependencies while improving expressivity with transient dynamics. In *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS2019)*, Vancouver, Canada, 2019.
- [26] D.P. Kingma and J.L. Ba. Adam: A method for stochastic optimization. In *Proceedings of ICLR 2015*, SanDiego, CA, 2015. ICLR.
- [27] A. Kirzhevsky, V. Nair, and G. Hinton. Cifar-10. <https://www.cs.toronto.edu/~kriz/cifar.html>, 2009.
- [28] J. Kohler, H. Daneshmand, A. Lucchi, M. Zhou, K. Neymeyr, and T. Hofmann. Towards a theoretical understanding of batch normalization. *arXiv:1805.10694v1*, 2018.
- [29] Q. V. Le, N. Jaitly, and G. E. Hinton. A simple way to initialize recurrent networks of rectified linear units, 2015.
- [30] Y. LeCun, C. Cortes, and C. J. C. Burges. The mnist database.
- [31] X. Lian and J. Liu. Revisit batch normalization: New understanding from an optimization view and a revinement via composition optimization. *arXiv:1810.06177v1*, 2018.
- [32] Y. Ma and D. Klabjan. Diminishing batch normalization. *arXiv:1705.08011v2*, 2019.
- [33] K. Maduranga, K. Helfrich, and Q. Ye. Complex unitary recurrent neural networks using scaled cayley transform. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI 2019)*, Honolulu, HI, 2019. AAAI Publications.
- [34] M.P. Marcus, M.A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of english: the penn treebank. *Computational Linguistics*, 19(2):313–330, 1993.
- [35] Z. Mhammedi, A. Hellicar, A. Rahman, and J. Bailey. Efficient orthogonal parameterisation of recurrent neural networks using householder reflections. In *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*, Sydney, Australia, 2017. PMLR: 70.
- [36] T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida. Spectral normalization for generative adversarial networks. In *International Conference on Learning Representations*, 2018.

- [37] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *27th International Conference on Machine Learning (ICML 2010)*, Haifa, Israel, 2010.
- [38] M. Naumov. Feedforward and recurrent neural networks backward propagation and hessian in matrix form. *arXiv:1709.06080v1*, 2017.
- [39] Evan O’Dorney. Minimizing the cayley transform of an orthogonal matrix by multiplying by signature matrices. *Linear Algebra and its Applications*, 448:97103, 05 2014.
- [40] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In S. Dasgupta and D. McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning (ICML 2013)*, pages 1310–1318, Atlanta, Georgia, USA, 2013. PMLR.
- [41] A. W. Rix, J. G. Beerends, M. P. Hollier, and A. P. Hekstra. Perceptual evaluation of speech quality (pesq)-a new method for speech quality assessment of telephone networks and codecs. In *Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP’01). 2001 IEEE International Conference on*, volume 2, pages 749–752. IEEE, 2001.
- [42] S. Santurkar, D. Tsipras, A. Ilyan, and A. Madry. How does batch normalization help optimization? In *32nd Conference on Neural Information Processing Systems (NeurIPS 2018)*, Montreal, Canada, 2019.
- [43] C. H. Taal, R. C. Hendriks, R. Heusdens, and J. Jensen. An algorithm for intelligibility prediction of time–frequency weighted noisy speech. *IEEE Transactions on Audio, Speech, and Language Processing*, 19(7):2125–2136, 2011.
- [44] Hemant D. Tagare. Notes on optimization on stiefel manifolds. Technical report, Yale University, 2011.
- [45] C. Trabelsi, O. Bilaniuk, D. Serdyuk, S. Subramanian, J. Santos, S. Mehri, N. Rostamzadeh, Y. Bengio, and C. Pal. Deep complex networks. *CoRR*, abs/1705.09792, 2017.
- [46] Zaiwen Wen and Wotao Yin. A feasible method for optimization with orthogonality constraints. In *Mathematical Programming*, volume 142(1-2), pages 397–434. Springer, 2013.
- [47] W. Wirtinger. Zur formalen theorie der funktionen von mehr komplexen veränderlichen. *Mathematische Annalen*, 97:357–375, 1926.
- [48] S. Wisdom, T. Powers, J. Hershey, J. Le Roux, and L. Atlas. Full-capacity unitary recurrent neural networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 4880–4888. Curran Associates, Inc., 2016.

- [49] M. Wolter and A. Yao. Complex gated recurrent neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 10536–10546. Curran Associates, Inc., 2018.
- [50] G. Yang, J. Pennington, V. Rao, J. Sohl-Dickstein, and S. Schoenholz. A mean field theory of batch normalization. *arXiv:1902.08129v2*, 2019.
- [51] J. Zhang, Q. Lei, and I. Dhillon. Stabilizing gradients for deep neural networks via efficient SVD parameterization. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*, volume 80 of *Proceedings of Machine Learning Research*, pages 5806–5814, Stockholmsmssan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

Kyle Helfrich

EDUCATION

- Ph.D. University of Kentucky** - Lexington, KY Expected May 2020
Mathematics
Advisor: Dr. Qiang Ye
- M.S. University of Illinois at Chicago** - Chicago, IL May 2014
Applied Mathematics
- B.S. Purdue University** - West Lafayette, IN May 2006
Civil Engineering
- B.A. Manchester University** - North Manchester, IN May 2005
Engineering Science

PUBLICATIONS

- Helfrich, K. and Ye, Q. 2020. Eigenvalue Normalized Recurrent Neural Networks for Short Term Memory. In Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2020). New York City, New York: AAAI. ArXiv e-prints arXiv:1911.07964.
- Maduranga, K.; Helfrich, K.; and Ye, Q. 2019. Complex Unitary Recurrent Neural Networks using Scaled Cayley Transform. In Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI 2019). Honolulu, Hawaii: AAAI. arXiv e-prints arXiv:1811.04142v2.
- Helfrich, K.; Willmott, D.; and Ye, Q. 2018. Orthogonal Recurrent Neural Networks with Scaled Cayley Transform. In Proceedings of the Thirty-Fifth International Conference on Machine Learning (ICML 2018). Stockholm, Sweden: PMLR. arXiv e-prints arXiv:1707.09520v3.
- Peethamparan, S.; Olek, J.; and Helfrich, K. 2006. Evaluation of the Engineering Properties of Cement Kiln Dust (CKD) Modified Kaolinite Clay. In Proceedings of the Twenty-First International Conference on Solid Waste Technology and Management. Philadelphia, Pennsylvania.