




2018

Recurrent Neural Networks and Their Applications to RNA Secondary Structure Inference

Devin Willmott

University of Kentucky, devin.willmott@uky.edu

Author ORCID Identifier:

 <https://orcid.org/0000-0001-7512-310X>

Digital Object Identifier: <https://doi.org/10.13023/etd.2018.401>

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

Recommended Citation

Willmott, Devin, "Recurrent Neural Networks and Their Applications to RNA Secondary Structure Inference" (2018). *Theses and Dissertations--Mathematics*. 58.
https://uknowledge.uky.edu/math_etds/58

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.

Devin Willmott, Student

Dr. Qiang Ye, Major Professor

Dr. Peter Hislop, Director of Graduate Studies

Recurrent Neural Networks and Their Applications to RNA Secondary Structure
Inference

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
Devin Willmott
Lexington, Kentucky

Director: Dr. Qiang Ye, Professor of Mathematics
Lexington, Kentucky 2018

Copyright© Devin Willmott 2018

ABSTRACT OF DISSERTATION

Recurrent Neural Networks and Their Applications to RNA Secondary Structure Inference

Recurrent neural networks (RNNs) are state of the art sequential machine learning tools, but have difficulty learning sequences with long-range dependencies due to the exponential growth or decay of gradients backpropagated through the RNN. Some methods overcome this problem by modifying the standard RNN architecture to force the recurrent weight matrix W to remain orthogonal throughout training. The first half of this thesis presents a novel orthogonal RNN architecture that enforces orthogonality of W by parametrizing with a skew-symmetric matrix via the Cayley transform. We present rules for backpropagation through the Cayley transform, show how to deal with the Cayley transform's singularity, and compare its performance on benchmark tasks to other orthogonal RNN architectures. The second half explores two deep learning approaches to problems in RNA secondary structure inference and compares them to a standard structure inference tool, the nearest neighbor thermodynamic model (NNTM). The first uses RNNs to detect paired or unpaired nucleotides in the RNA structure, which are then converted into synthetic auxiliary data that direct NNTM structure predictions. The second method uses recurrent and convolutional networks to directly infer RNA base pairs. In many cases, these approaches improve over NNTM structure predictions by 20-30 percentage points.

KEYWORDS: Machine learning, deep learning, neural networks, recurrent neural networks, RNA secondary structure, secondary structure inference

Author's signature: Devin Willmott

Date: October 29, 2018

Recurrent Neural Networks and Their Applications to RNA Secondary Structure
Inference

By
Devin Willmott

Director of Dissertation: Qiang Ye

Director of Graduate Studies: Peter Hislop

Date: October 29, 2018

ACKNOWLEDGMENTS

I feel extraordinary gratitude toward many, many people who have helped me during my time in graduate school, including all of the staff and faculty in the math department, my friends and family outside of Kentucky, the staff of several bars and coffee shops in Lexington, and the University of Kentucky Medical School's Orthopedic Surgery department. More specifically, there are a few people who were particularly important to the creation of this thesis and deserve their individual thanks:

Thank you to my advisor, Qiang Ye, for taking me on as a student, and for all for your support, guidance, and patience throughout the past three years.

Thank you to David Murrugarra, for your generosity in involving me in your research and in meeting weekly with me. I appreciate all of your guidance from the years we have worked together.

Thank you to Nathan Jacobs, for teaching an outstanding first class in machine learning, and for giving me direction early on in my research.

Thank you to Kyle Helfrich, for being a great academic sibling, and for being a helpful and conscientious collaborator.

Thank you to all of my fellow graduate students in the math department, for every problem we worked on together and every class that you covered for me. More than that, thank you for all of the friendships and memories you have left with me over the years: for playing board games with me, for getting coffee with me, for going to the bar with me, for playing music with me. I feel enormously lucky that you are the people with whom I got to share this experience.

Thank you to Ian Strause, for everything.

Finally, thank you to Matt Ollis, for (almost) singlehandedly providing my entire undergraduate math education, for showing me how to be a researcher, and for

encouraging me to pursue graduate school. You introduced me to one of the most important and fulfilling pursuits of my life, I would not be submitting this thesis without you.

TABLE OF CONTENTS

Acknowledgments	iii
List of Tables	vi
List of Figures	vii
Chapter 1 Introduction	1
1.1 Machine Learning	1
1.2 Neural Networks	4
1.3 Neural Network Training	6
1.4 Neural Network Variants: CNNs and RNNs	8
1.5 Thesis Outline	12
Chapter 2 The Scaled Cayley Orthogonal RNN	14
2.1 Exploding and Vanishing Gradients	14
2.2 Solutions to Exploding and Vanishing Gradients	15
2.3 Parametrization with the Cayley Transform	19
2.4 The scoRNN Architecture	22
2.5 Copying Problem Experiments	24
2.6 RNA State Inference Experiments	27
Chapter 3 RNA Sequences and Structures	29
3.1 RNA	29
3.2 Secondary Structure Inference Methods	32
3.3 SHAPE and Data-Directed NNTM	34
Chapter 4 Improving NNTM via State Inference with Neural Networks	36
4.1 SHAPE-Directed NNTM Without SHAPE	36
4.2 Methods	37
4.3 State Inference	40
4.4 Directing NNTM with Synthetic SHAPE	45
Chapter 5 Direct Secondary Structure Inference with Neural Networks	50
5.1 Representing Secondary Structures	50
5.2 Neural Network Architecture	51
5.3 16S rRNA Structure Inference	56
5.4 General RNA Structure Inference	60
Bibliography	64
Vita	69

LIST OF TABLES

2.1	Table of hidden sizes and total number of trainable parameters of each machine used in the copying problem experiment	25
4.1	State inference PPV, sensitivity, and accuracy of our neural network vs. HMM on validation and test sets	40
4.2	State inference results on the test set from our neural network vs. an order 4 HMM. Sequences are arranged in ascending order of MFE accuracy as an indication of the difficulty of secondary structure inference for each sequence. Average indicates the average metric for each sequence, while Total gives the total metrics for all nucleotides in the test set.	41
4.3	Table of accuracy of MFE structures using NNTM with a variety of SHAPE directions. First column: undirected MFE. Second column: predicted state directed MFE. Third column: mean performance of sampled SHAPE directed NNTM in [48]. Fourth column: native state directed NNTM.	45
5.1	Enumeration of sequences in training, validation, and test sets by RNA family.	56
5.2	Results on the 16S rRNA secondary structure inference task	56
5.3	Secondary structure inference accuracy on the Rogers set with a neural network trained on 16S rRNA sequences	57
5.4	Sequence results on the Sukosd set on the 16S rRNA secondary structure inference task	58
5.5	Results on the full RNA secondary structure inference task. Rogers Set (Other) refers to the 15 non-16S rRNA sequences in the Rogers set. . . .	60
5.6	Structure inference accuracy on the Rogers set with the full training set .	62
5.7	Sequence results on the Sukosd set on the full RNA secondary structure inference task	63

LIST OF FIGURES

1.1	A visual representation of an RNN. Here, the hidden state is represented as $s^{(t)}$ and the output is represented as $o^{(t)}$. Retrieved from [21].	11
2.1	Diagram of a Long Short-Term Memory (LSTM) recurrent step. Retrieved from [8].	16
2.2	Cross entropy of each machine on the copying problem with $T = 1000$ (left) and $T = 2000$ (right).	26
2.3	Several scoRNN models on the copying problem with $T = 100$ for varying values of ρ	26
2.4	Performance of scoRNN vs. an LSTM on the simplified state inference problem	27
3.1	An RNA sequence of length $L = 23$. Nucleotides are shown above their index, and arcs are drawn between nucleotides in a base pair.	31
4.1	Diagram of a two-layer deep bidirectional RNN at three different timesteps $t - 1$, t , and $t + 1$. Solid arrows represent the propagation of information through layers at a particular timestep, while dashed arrows represent information propagating across timesteps.	39
4.2	Boxplots of the distribution of sizes of paired regions in the native state sequence, HMM predicted state sequence, and LSTM predicted state sequence (denoted N, H, and L, respectively) for each test set sequence. The red line indicates the median region size, the box contains 25th-75th percentiles, and the whiskers contain 5th-95th percentiles. Sequences are ordered from lowest to highest LSTM prediction accuracy. Several large paired regions in HMM predictions beyond the y-axis limit of 35 are not shown.	43
4.3	Plot comparing each test set sequence's LSTM (neural network) and HMM state inference accuracy vs its Kullback-Leibler divergence from training set paired region distribution. KL divergence was calculated as $KL(P Q)$, where P is the test sequence distribution and Q is the training set distribution.	44
4.4	Accuracy of native state directed MFE (top plot) and predicted state directed MFE (bottom plot) for various ranges $[a, b]$ of output from our SHAPE generation function. In each, the lower right corner corresponds to $a = b = 0.3603$, which is equivalent to no SHAPE direction.	48
5.1	Diagram of neural network architecture used for secondary structure inference in Chapter 5. SelfCartesian refers to the operation described at the beginning of Section 5.2.	52

5.2 ROC curve examining the effect of varying the threshold v on accuracy on the E. cuniculi test sequence. Labels indicate the value of v at that point. 59

Chapter 1 Introduction

1.1 Machine Learning

Machine learning is a collection of mathematical and statistical tools for gaining insight about a known dataset, or using this dataset to make predictions about data points outside of the set. Crucially, in machine learning this is done largely without domain-specific knowledge; rather than being programmed by hand, machine learning algorithms are trained through an optimization procedure.

Let $\{x_i\}_{i=1}^N$ be our known dataset, often called the **training set**. Each data point x_i has a number of known qualities, referred to as **features**. We represent each feature with a value in \mathbb{R} , and thus $x_i \in \mathbb{R}^n$, where n is the number of features of each data point. We refer to \mathbb{R}^n as **feature space**, and can represent the dataset as a matrix $X \in \mathbb{R}^{N \times n}$. In mathematical terms, we are interested in using X to determine a function $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$. This function is the **machine** or **model** in question. What this function represents varies greatly depending on the information available, and on the problem we wish to solve. There are two main classes of machine learning problems: unsupervised learning and supervised learning.

In **unsupervised learning** problems, we have no information beyond the dataset X , and g is used to give us some understanding of the data. For example, we may be interested in grouping data points in feature space into k clusters according to some notion of similarity, in which case g will map any $x \in \mathbb{R}^n$ to one of k clusters. We could also be interested in a function that reduces the dimensionality of the data but approximately preserves relative distances between points. One common motivation for this is data visualization, in which case g will often map to \mathbb{R}^2 or \mathbb{R}^3 .

In **supervised learning** problems, which are the focus of this work, we are given both the data set X and a **label** $y_i \in \mathbb{R}^m$ associated with each data point x_i . The label represents the target output of the machine, and the process of training is to use the example input-output pairs to infer the rest of the function. Thus, we will try to find a machine that recognizes that $\hat{y}_i \approx y_i$, where $\hat{y}_i := g(x_i)$ is the machine's prediction of the label y_i .

A supervised learning problem can be further categorized as a **regression** problem, where the output is a continuous value, or a **classification** problem, where the output is one of k classes where $k \in \mathbb{N}$. The simplest type of classification problem is the case where $k = 2$, called a **binary classification** problem. In binary classification the the output dimension m of g is 1; different machine learning algorithms use different labels for binary classification, but in this work we represent the labels as 0 or 1. If $k > 2$, we have a **multiclass classification** problem. To represent the output of a multiclass classification problem, we often use **one-hot encoding**. This represents the i -th class using the vector in \mathbb{R}^k with a 1 in cell i and 0 elsewhere.

There exist an enormous variety of machine learning algorithms, each with advantages and disadvantages in speed, efficiency, and underlying assumptions about the data. Though nonparametric machine learning models exist, we will only consider

parametric machine learning algorithms, and we can thus define a machine learning algorithm as a parametric family of functions $F = \{f(x, \theta) \mid \theta \in \Theta\}$. The process of finding an appropriate machine $f \in F$ is called **training**. This process varies greatly among algorithms; some algorithms require essentially no training time, while others, like neural networks, have long and complicated training processes. Frequently, training involves optimizing some or all of the parameters θ with respect to some loss function L (this can also be called the cost function or objective function). In supervised problems, L is a comparison of the predicted output \hat{y} and the real output y , so $L = L(\hat{y}, y) = L(f(x, \theta), y)$. Usually $L(\hat{y}, y) = 0$ if and only if $\hat{y} = y$, though it is not necessarily a distance metric. Once we have settled on some $f \in F$, we fix the parameters and let $g(x) := f(x, \theta)$.

Machine Learning Examples

Several classical machine learning algorithms will be familiar to mathematical audiences under different guises. One of the simplest is that of linear regression, where $f(x, \theta)$ is a linear combination of features of x . We represent coefficients of this linear combination with **weights** $w \in \mathbb{R}^n$, so that $f(x) = w^T x$. The weights are the only parameters in this machine, so $\theta = w$.

A commonly used loss function for regression problems is the square of the 2-norm of the difference between machine prediction and label, referred to as **mean squared error** (MSE). Summing the loss over each point in our data set, this can be represented as

$$L(\hat{y}, y) = \sum_{i=1}^N \|w^T x_i - y_i\|_2^2 = \|Xw - y\|_2^2$$

where y is the columns of labels $[\hat{y}_1 \dots \hat{y}_N]^T$. We would like to find the parameters $\theta^* \in \Theta$, or equivalently the weights $w^* \in \mathbb{R}^n$, that minimize this loss, i.e.,

$$w^* = \operatorname{argmin}_{w \in \mathbb{R}^n} L(\hat{y}, y) = \operatorname{argmin}_{w \in \mathbb{R}^n} \|Xw - y\|_2^2$$

Presented in this manner, our loss function is recognizable as the linear least squares problem. This optimization problem has a closed form solution; we can, for example, use the normal equations to find that

$$w^* = (X^T X)^{-1} X^T y$$

where we may substitute the inverse for the Moore-Penrose pseudoinverse if $X^T X$ is not invertible.

Having found the optimal weights w^* , we can now use $g(x) := f(x, \theta^*) = (w^*)^T x$ to make predictions. The output of $g(x)$ is the machine's prediction of its label y .

As defined, a linear regression model f will always produce the prediction $\hat{y} = 0$ when the input x is the zero vector, which is not a particularly desirable property. To avoid this, we can instead consider affine combinations of x by introducing a trainable bias term $b \in \mathbb{R}$ to the prediction. Now, $\theta = \{w, b\}$, and $f(x, \theta) = w^T x + b$. This

does not present an obstacle from an optimization perspective, as this is equivalent to appending a feature valued at 1 to each data point in X and appending a corresponding element to w .

Another illuminating example is that of logistic regression, which uses a similar linear combination of features of x . Despite its name, logistic regression is used to solve a binary classification problem. Our function f is given by $f(x, \theta) = \sigma(w^T x + b)$, where $\sigma(z) = (1 + e^{-z})^{-1}$ is the logistic function. The range of this function is $(0, 1)$, and we can thus interpret the output of the machine as a probability, specifically the probability $P(y = 1 \mid x)$ that x belongs to class 1.

For binary classification, the most popular loss function is **binary cross entropy**, given by

$$L(\hat{y}, y) = - \sum_{i=1}^N y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i).$$

Note that since $y_i \in \{0, 1\}$, only one of the terms in each summand will be nonzero.

Unlike linear regression, there is no known closed form expression for the weight w^* that minimizes binary cross entropy [21]. Instead, we must use iterative methods to solve this problem.

In most machine learning problems with a differentiable loss function L , the large dimensionality of the data means that direct computation of the full Hessian of parameters $H_L(\theta)$ is generally computationally infeasible. Thus, optimization is frequently relegated to first-order methods. The most popular training algorithms for machine learning are based upon **gradient descent**. In gradient descent, we begin with some initial set of parameters θ , which for this purpose is presented as a vector of parameters. Then, for each pair $x_i \in X$ and $y_i \in y$, we fix x_i and y_i and we find the gradient of θ with respect to L :

$$\Delta\theta = \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} L(f(x_i, \theta), y_i).$$

This $\Delta\theta$ is multiplied by a small constant η called the **learning rate** and subtracted from the current parameter vector:

$$\theta \leftarrow \theta - \eta\Delta\theta.$$

A Taylor's theorem argument can show that there exists some learning rate η such that the updated w will result in a decrease in L . The learning rate is an example of a **hyperparameter**, a variable that affects training but is not a part of the trainable parameter set θ . The learning rate must be chosen carefully; if η is too small, training time can increase dramatically, while a too large learning rate is liable to overshoot local minima in the loss surface and cause instability.

For logistic regression, we repeat this process until $\|\Delta\theta\|$ falls below a specified threshold, as this indicates that we are near a local minimum on the loss surface. At this point we let our machine $g(x)$ be $f(x, \theta)$.

Once trained, $g(x)$ outputs a predicted probability that x is in class 1. To obtain a class prediction, we threshold this value, so that $g(x) > 0.5$ is a prediction that x belongs to class 1, and $g(x) < 0.5$ a prediction that x is in class 0.

1.2 Neural Networks

The algorithms presented in the previous section make predictions based on an affine combination of features of the data. For example, from a geometric perspective, the logistic regression algorithm learns a hyperplane given by $w^T x + b = 0$, and classifies points based on which side of this hyperplane a data point falls on, as $\sigma(0) = 0.5$. These linearity assumptions make for convenient training of the parameters, but are often far too simplistic to achieve desirable accuracy on a great many machine learning tasks.

A **(feedforward) neural network** is a machine learning algorithm with significantly more expressive power than linear models. Neural networks have been studied for decades under various guises, and their popularity has fluctuated relative to other machine learning algorithms; see the introduction of [21] for a brief history. Despite their name, today's neural networks have little in common with the biological neural networks that originally inspired them.

Neural networks are most easily understood as an extension of logistic regression. A simple neural network $f(x, \theta)$ with output \hat{y} and $\theta = \{W_0, b_0, W_1, b_1\}$ is shown below.

$$\begin{aligned} h &= \sigma(W_0 x + b_0) \\ \hat{y} &= W_1 x + b_1 \end{aligned}$$

Here $x \in \mathbb{R}^n$, $h \in \mathbb{R}^{n_1}$, and $\hat{y} \in \mathbb{R}^m$. As with all neural networks, the information flows through this machine in a hierarchical fashion, with x generating h , which in turn generates \hat{y} . Each level of this hierarchy is referred to as a **layer**; the final layer is the **output layer**, while the intermediate layer is the **hidden layer**.

Each element of the hidden layer is the output of a logistic regression performed on the input x . Intuitively, we can think of each element of h as representing a high-level feature of the data, built from a combination of lower-level input features. We do not know what exactly these high-level features represent, nor do we need to; with successful optimization, training will find the parameters that generate the most useful high-level features (represented by W_0 and b_0), as well as parameters that use these high-level features to predict the label (W_1 and b_1). Note that nothing about the input or output dimensions of the problem specify the hidden dimension n_1 . In fact n_1 is a hyperparameter that we can increase or decrease to control the expressive capability of the network, at the cost of additional computational and memory requirements.

The elementwise logistic function σ , here called the **activation function**, plays an important role in the neural network, as it allows us to represent nonlinear relationships among the inputs. However, it is no longer necessary for the output of this

function to be interpretable as a probability, and thus we are free to change this from the logistic function to any elementwise nonlinearity.

Our choice of activation function will have a significant effect on network optimization. Historically, the logistic function has been a popular choice, as has the hyperbolic tangent function \tanh , which is equivalent to the logistic function rescaled to $(-1, 1)$, that is, $\tanh = 2\sigma - 1$. However, the current favorite activation function is the **rectified linear unit (ReLU)** [40], shown below.

$$\sigma_R(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

This may seem like an odd choice for an activation function, as it is both piecewise linear and nondifferentiable. In practice, it is able to represent nonlinear relationships well; we can think of it as a piecewise linearized version of $\ln(1 + e^x)$. The nondifferentiability at 0 is not an issue for the purposes of gradient descent ($\sigma'_R(0)$ is simply chosen to be either 0 or 1), and the derivative's value of either 0 or 1 has been shown to introduce favorable training dynamics into the network [40].

The final layer is left linear for flexibility; for regression problems, we can directly use \hat{y} as our output and MSE as our loss function L , and for binary classification problems we apply a sigmoid to represent the network's prediction, and let L be binary cross entropy. For multiclass classification, we first apply a **softmax** function $s : \mathbb{R}^m \rightarrow \mathbb{R}^m$ to the output. This is described below, where z_j describes the j -th element of $z \in \mathbb{R}^m$, and the exponential is applied to z elementwise.

$$s(z) = \frac{e^z}{\sum_{j=1}^m e^{z_j}} = \frac{e^z}{\|e^z\|_1}$$

The entries of $s(z)$ are bounded between $(0, 1)$ and together sum to 1, so we can interpret this vector as a probability distribution. In particular, if x is the network input and \hat{y} is the softmax output, then the j -th element of \hat{y} is the probability that x belongs to class j . We then use the **softmax cross entropy** loss function

$$L(\hat{y}, y) = \sum_{i=1}^N \sum_{j=1}^m -\log((\hat{y}_i)_j)(y_i)_j$$

where $(y_i)_j$ refers to the j -th element of the i -th label in $\{y_i\}_{i=1}^N$. Since the label y is a one-hot encoding, each label will be 1 at one particular index p_i and 0 everywhere else, so in fact this loss function amounts to $L(\hat{y}, y) = \sum_{i=1}^N -\log((\hat{y}_i)_{p_i})$.

We can further extend the idea of finding and using high-level features for supervised learning by inserting additional hidden layers into the network.

$$\begin{aligned}
h_1 &= \sigma(W_0x + b_0) \\
h_2 &= \sigma(W_1h_1 + b_1) \\
&\vdots \\
h_d &= \sigma(W_{d-1}h_{d-1} + b_{d-1}) \\
\hat{y} &= W_d h_d + b_d
\end{aligned}$$

Each hidden layer size $h_i \in \mathbb{R}^{n_i}$ for $1 \leq i \leq d$ may be chosen independently. As before, this machine is a composition of layers, each of which is itself a composition of a parametrized affine transformation and an activation function. The multiple layers of features combine the input from the previous layer to represent progressively higher level features.

A network with many hidden layers such as this is called a **deep neural network**; the network above has **depth** d . More generally, **deep learning** is the study of machine learning tools that use a hierarchy of constituent machine learning algorithms. Deep learning, and deep neural networks in particular, are responsible for the explosion in popularity of machine learning in recent years. This was initiated with Hinton’s landmark 2006 paper [24], which introduced a practical training scheme for deep neural networks and revitalized their study.

1.3 Neural Network Training

As in logistic regression, we rely on gradient descent to train a neural network. However, the increased expressive power and representational capacity of deep neural networks are accompanied by additional complications in training. In particular, the loss surface becomes nonconvex in a single layer neural network, and this nonconvexity intensifies as the depth of the network increases.

Of most immediate concern is the obtaining of the gradient $\nabla_{\theta}L$. Unlike logistic regression, we cannot consider each parameter individually, as neural network parameters are not independent of one another. However, we can exploit the hierarchical structure of neural networks to efficiently obtain parameter gradients with comparable time complexity to the forward pass of the machine. This repeated use of the chain rule is known as the **backpropagation algorithm** [46].

It is often unnecessary to compute the gradient over the entire training set at every iteration; instead, we partition the dataset into random subsets of size b called **batches**, and perform gradient descent on a single batch at each iteration. This increases computational efficiency, and the stochasticity is a benefit when training on the nonconvex loss surface of the neural network. This is called **(batch) stochastic gradient descent**, or batch SGD.

Even after obtaining the gradient of the parameters, neural network training is not straightforward. A consequence of the increased representational capacity of deep neural networks is their ability to capture patterns that exist in the training set, but

do not generalize to other points in feature space. This will result in high accuracy on the training set and low accuracy outside of it, a phenomenon called **overfitting**.

Improving performance without overfitting is one of the central struggles of machine learning. There exist numerous network modifications and training heuristics designed to counteract neural networks' tendency to overfit; these methods are called **regularization** methods. At their core, regularization techniques attempt to restrict the ability of the network to detect spurious patterns in the data, while still recognizing patterns that generalize outside the training set; the difficulty lies in the fact that, by definition, the machine cannot distinguish which of these patterns are spurious.

A common regularization technique is the inclusion of a term in the loss function, usually an **L2-regularization term**:

$$\tilde{L}(\hat{y}, y) = L(\hat{y}, y) + \frac{\lambda}{2} \|\theta\|_2^2.$$

The constant λ is the L2-regularization coefficient, a hyperparameter that controls the strength of regularization. After taking the gradient, we can see that regularization will decrease parameters by an amount proportional to that parameter's current value.

$$\theta \leftarrow \theta - \eta \Delta\theta - \lambda\theta$$

This has the effect of keeping parameters small, which is a form of restricting the representational capacity of the network; it is known to prevent overfitting in practice [21].

There exists an enormous variety of techniques, tricks, methods, and modifications to improve neural network training and performance. Among these, we briefly mention alternate learning algorithms like RMSprop [54] and Adam [30], which speed up training by incorporating momentum terms and adaptive learning rates into the computation of $\Delta\theta$, as well as the dropout training method [47], which reduces overfitting, and batch normalization [27], which allows the machine to accommodate higher learning rates and has some regularization effects. We refer to [21] for a comprehensive reference on all of these techniques and many more.

The Training Process

We have already seen several examples of hyperparameters: the number of layers d , the sizes n_1, \dots, n_d of each layer, the learning rate η , as well as various hyperparameters introduced by regularization techniques and alternate training algorithms. By and large, there is not sufficient theoretical understanding of neural networks to prescribe the best set of hyperparameters for a particular learning task [21]; instead, we are usually forced to try several different values and pick the one that gives the highest performance. This is complicated by the spectre of overfitting, which means that a decrease in training set error does not necessarily imply a more accurate machine.

With this in mind, in a typical training process, we partition our known dataset into three sets: a training set, **validation set**, and **test set**. We use the training and validation sets to find the best set of hyperparameters: for each hyperparameter configuration, we train on the training set and monitor the loss function on the

validation set. However, by modifying the hyperparameters to elicit the best performance on the validation set, we may be artificially increasing the accuracy on the validation set – we can think of tuning hyperparameters as another form of (manual) training, one in which it is also possible to overfit. Thus, to eliminate possible bias, we examine the accuracy on the test set only after fixing the hyperparameters. The loss and accuracy on this test set is then considered to be the true representation of the performance of the machine.

1.4 Neural Network Variants: CNNs and RNNs

The multilayer feedforward neural network presented in the previous section has had enormous success in a variety of contexts. However, there are situations where this architecture proves to be inefficient or inflexible. This is particularly true in domains where the features of the data have some spatial structure, that is, when the features of x can be organized in a meaningful way. Fully connected feedforward networks ignore any spatial relationship among inputs: each hidden unit is comprised of a parametrized combination of all features in the previous layer. This approach can be wasteful or redundant, depending on the context: it may be that a feature x_i is only related to a small subset of the other features in x , or that there are many relationships among inputs that are repeated. Ignoring these patterns can be inefficient, and can be harmful to the network’s ability to generalize. Alternative architectures have been presented to exploit the spatial coherence among these data types, which allow us to drastically reduce the number of parameters and speed up training.

When the input x has some spatial structure, it is often sensible to change our representation from the vector in \mathbb{R}^n used in the previous section. It may be most natural to represent x as a single high rank tensor or as a sequence of tensors, depending on the context. Regardless of the representation, in this work we will use superscripts in parentheses to denote spatial or temporal relations among data, and subscripts to denote the features at a particular time or space.

Below we present the two most commonly used and studied neural network variants: convolutional neural networks, which can take in data with spatial structure of arbitrary dimension, and recurrent neural networks, which are of use in sequential learning tasks. These variants are essentially modified layers that replace the fully-connected layer in the feedforward network presented in the previous section, and these variants are compatible with one another: we are free to vary the layers of the network as the problem requires. For example, architectures in both Chapter 4 and Chapter 5 comprise both convolutional and recurrent layers.

Convolutional Neural Networks

Perhaps the most successful breed of neural networks in the advent of deep learning are convolutional neural networks (CNNs). While small CNNs have been used successfully for many years [33], large scale use of CNNs have dramatically reshaped the field of computer vision over the past decade [2, 21].

Let f and g be functions defined on \mathbb{N} . Recall that the one-dimensional (1D) **discrete convolution** with input f and kernel g , denoted $f * g$, is itself a function on \mathbb{N} , given by

$$(f * g)(t) = \sum_{j=-\infty}^{\infty} f(t-j)g(j).$$

If f and g are defined on \mathbb{N}^2 , then we can define the two-dimensional (2D) discrete convolution

$$(f * g)(s, t) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f(s-i, t-j)g(i, j).$$

For the purposes of using convolutions with neural networks, we will instead think of discrete convolution as an operation between tensors, with the input to f and g now representing indices of these tensors. The convolved output may be represented as a tensor of the same size of the input tensor, since the support of $f * g$ is at most the support of f . We also restrict the support of g to an interval $[-s, s]$. The length $2s + 1$ of this interval is the size of the kernel.

As an example, if we have an input $x \in \mathbb{R}^{L \times n}$ and kernel $w \in \mathbb{R}^{(2s+1) \times n}$, then the 1D convolution $x * w$ is given by

$$(x * w)^{(\ell)} = \sum_{n'=1}^n \sum_{j=-j_0}^{j_0} x_{n'}^{(\ell-j)} w_i^{(j)} = \sum_{j=-j_0}^{j_0} (x^{(\ell-j)})^T w^{(j)}.$$

For notational consistency and simplicity, we will index the kernel w with $-s, \dots, s$.

Note that when x has multiple channels, we simply sum over them. However, this gives an output of only one channel. For an output with q channels, we must increase the rank of the kernel to $\mathbb{R}^{(2s+1) \times n \times q}$:

$$(x * w)_q^{(\ell)} = \sum_{n'=1}^n \sum_{j=-s}^s x_{n'}^{(\ell-j)} w_{i,q}^{(j)} = \sum_{j=-s}^s (x^{(\ell-j)})^T w_{:,q}^{(j)}.$$

Finally, for completeness, we also show a 2D convolution with a rank-3 input tensor $x \in \mathbb{R}^{K \times L \times n}$ and a rank-4 kernel $w \in \mathbb{R}^{(2s+1) \times (2s+1) \times n \times q}$. Though we now have two spatial dimensions, we use the same kernel size $2s + 1$ in each dimension.

$$(x * w)_q^{(k,\ell)} = \sum_{n'=1}^n \sum_{i=-s}^s \sum_{j=-s}^s x_{n'}^{(k-i,\ell-j)} w_{n,q}^{(i,j)}$$

The two cases above (1D convolution with a rank-2 tensor input and rank-3 kernel, and 2D convolution with a rank-3 tensor input and rank-4 kernel) are the only two kinds of convolution we will employ in this work. Thus, the dimension of the convolution and the rank of the kernel will be apparent based on the rank of the input tensor.

2D convolutions are frequently used in image processing contexts. Given an input image x , 2D convolution of x with particular kernels can denoise, sharpen, or blur an image. Other kernels can elicit information from a picture, such as the edge detection kernel defined below:

$$w = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

The convolved image $x * w$ will be large at pixels that differ significantly in value with neighboring pixels (that is, near edges in the image), and will be small if the pixel is similar in value to its neighbors. We draw analogy between the output of this kernel and the elements of a neural network’s hidden layer, both of which have combined low-level information from the input to present high-level features that may be useful for learning tasks.

To connect convolutions with the framework of neural networks, we simply replace the linear transformation in a feedforward neural network layer with a convolution:

$$h_i = \sigma(h_{i-1} * w + b).$$

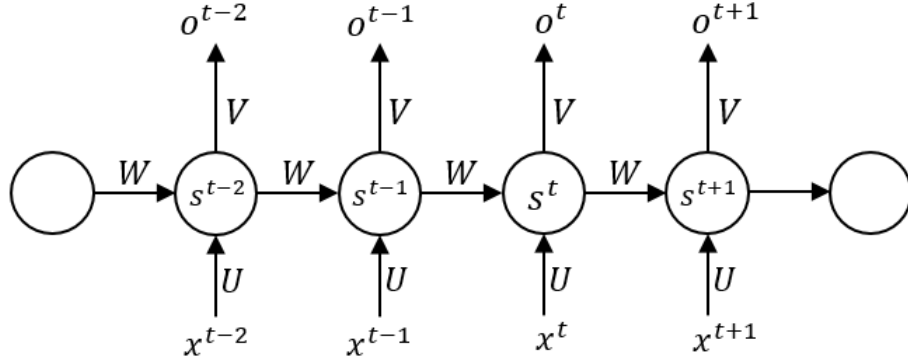
Now the kernel elements w and the bias b are the trainable variables of the layer. Here the bias is repeated across spatial dimensions, so b will have nQ independent elements, where n and Q are the number of input and output channels of the kernel w , respectively.

We can think of a convolutional layer as a learnable version of the image processing kernel described above. This setup hinges on an additional assumption about the data, namely that it is sensible to apply the same function to each $(2s + 1) \times (2s + 1)$ square in the data, regardless of the particular spatial location of this square. This is called the assumption of **translation invariance**. In return for satisfying the assumption, the benefit is a layer with many fewer parameters than the fully connected layer of a feedforward neural network. Additionally, whereas a fully connected layer can only take in data of a particular dimension, the convolutional layer can process any input with n channels, regardless of the size of the input’s spatial dimensions K and L .

The same gradient descent-like algorithms that train feedforward neural networks can be used on convolutional networks. Though our exposition above considered convolution as an operation between tensors of arbitrary rank, it can also be represented as a linear transformation with certain kinds of matrices (Toeplitz matrices for 1D convolution, and doubly block circulant matrices for 2D convolution [21]), and thus the backpropagation algorithm can be implemented without significant changes from the feedforward case.

CNNs are the current state of the art for image processing tasks, and like feedforward neural networks, high performing networks are subject to numerous modifications. We do not make use of many of these modifications, as the learning tasks we consider are relatively tame for a CNN. Our inputs and outputs are always of

Figure 1.1: A visual representation of an RNN. Here, the hidden state is represented as $s^{(t)}$ and the output is represented as $o^{(t)}$. Retrieved from [21].



the same spatial dimensions, and thus we do not need to make use of any strided convolutions, transposed convolutions, or pooling layers. We refer to [2] and [21] as references for popular CNN modifications.

Recurrent Neural Networks

Another domain where variations on feedforward neural networks can lead to increased flexibility and efficiency is that of sequence learning, where input and/or output has a sequential structure. We represent the input as a sequence $x = (x^{(1)}, \dots, x^{(L)})$ where $x^{(\ell)} \in \mathbb{R}^n$ for each ℓ . Elements of this sequence are often related either spatially or temporally. This representation is appropriate for learning many kinds of data, including natural language processing, audio processing, and bioinformatics [21].

The **recurrent neural network (RNN)** was introduced as a neural network for sequential learning. The standard formulation of an RNN is given below.

$$\begin{aligned} a^{(\ell)} &= Ux^{(\ell)} + Wh^{(\ell-1)} + b \\ h^{(\ell)} &= \sigma(a^{(\ell)}) \\ \hat{y}^{(\ell)} &= Vh^{(\ell)} + c \end{aligned}$$

At a particular time ℓ , the input x , activation $a^{(\ell)} \in \mathbb{R}^{n_h}$, hidden state $h^{(\ell)} \in \mathbb{R}^{n_h}$, and output $\hat{y} \in \mathbb{R}^m$ are elements of sequences x , a , h , and \hat{y} of length L , respectively. As before, σ is an activation function; \tanh is a common choice for RNNs [21]. The trainable variables are the input matrix $U \in \mathbb{R}^{n_h \times n}$ and input bias $b \in \mathbb{R}^{n_h}$, the recurrent matrix $W \in \mathbb{R}^{n_h \times n_h}$, and the output matrix $V \in \mathbb{R}^{m \times n_h}$ and output bias $c \in \mathbb{R}^m$. The initial hidden state h_0 is usually fixed, often the zero vector; however, some RNN architectures have found it beneficial to regard h_0 as a set of trainable parameters [4]. A visual representation of an RNN is given in Figure 1.1.

Note that trainable parameters are reused at each step, and the number of parameters is entirely independent of the input sequence length; we may run the machine

to produce output sequence elements $y^{(\ell)}$ as long as we have more input sequence elements $x^{(\ell)}$ to feed into the machine.

We can draw analogies between convolutional and recurrent neural networks: just as CNNs are learned versions of image processing kernels, we can think of RNNs as discrete-time dynamical systems with the internal machinery of a neural network. Similarly, CNNs rely the assumption of translation invariance to reduce their total number of parameters, while RNNs do the same through the Markovian assumption that the hidden state at time ℓ is dependent only on the current timestep's input $x^{(\ell)}$ and the previous timestep's hidden state $h^{(\ell-1)}$.

The target output of an RNN may be the entire output sequence y , such as cases where we are interested in classifying each element of the sequence. The loss function in this case simply sums over elements of y and \hat{y} , that is,

$$L(\hat{y}, y) = \sum_{\ell=1}^L L_i(\hat{y}^{(\ell)}, y^{(\ell)}).$$

In other cases we may only be concerned with the final output $\hat{y}^{(L)}$, such as when we wish to classify the sequence itself, rather than each element. In this case, earlier elements of the output sequence \hat{y} are ignored, and L is simply a function of $\hat{y}^{(L)}$ and $y^{(L)}$.

The same gradient descent style of learning is used in the training of RNNs. However, the reuse of parameters means that we must modify the backpropagation algorithm to obtain the correct gradients for the recurrent matrix W and input parameters U and b . Instead, the similar Backpropagation Through Time (BPTT) algorithm [45, 58] supplies these gradients.

This RNN is able to capture dependencies between an input at step ℓ and any timesteps that have come before it. However, we may have sequences with time dependencies in both forward and backward directions. One straightforward solution to this is to use a pair of RNNs, one running forward in time, and another moving backward. We then concatenate their hidden layers. This pair is called a **bidirectional RNN**; we will make use of these modified RNNs in Chapters 4 and 5.

1.5 Thesis Outline

This thesis is broken into two main parts.

In Chapter 2, we investigate a common difficulty in training RNNs called the vanishing/exploding gradient problem, a phenomenon wherein gradients backpropagated through the RNN grow or decay exponentially in norm due to repeated multiplication of the recurrent weight matrix W in the BPTT algorithm. Some of the most successful methods for overcoming exploding and vanishing gradients modify the standard RNN architecture to force W to remain orthogonal throughout training, at the expense of complicated implementation. presents a novel orthogonal RNN architecture that enforces orthogonality of W by parametrizing with a skew-symmetric matrix A via the Cayley transform. We present rules for computing backpropagation through the Cayley transform, and show how to avoid issues with the Cayley transform's singularity

at orthogonal matrices with -1 eigenvalues via multiplication by a diagonal scaling matrix. The resulting scaled Cayley orthogonal recurrent neural network (scoRNN) is a simple RNN architecture that maintains orthogonality throughout training. We compare its performance to competing orthogonal RNN architectures, and find that the scoRNN exhibits highly competitive performance on several benchmark tasks.

The second part of this work is devoted to a problem in bioinformatics known as RNA secondary structure inference, which asks, given an RNA sequence, which nucleotides in this sequence form base pairs. Chapter 3 provides background regarding RNA sequences and structures that is necessary for understanding this problem. We also look at the state of the art bioinformatics tools for RNA secondary structure inference, the nearest neighbor thermodynamic model (NNTM), to understand its capabilities and limitations.

Chapters 4 and 5 each present a neural network architecture to solve problems related to RNA secondary structure inference. In Chapter 4, we attack the related problem of state inference using CNNs and bidirectional RNNs, and exhibit significant improvements on this task over a benchmark hidden Markov model (HMM). We then show how to convert state predictions from this network into auxiliary information that we can supply to NNTM to augment its predictions, improving accuracy by 25 percentage points on a test set of 16S rRNA.

Finally, Chapter 5 introduces a novel neural network architecture based primarily around a multilayer CNN that takes an RNA sequence as input and directly produces base pair predictions. We then show how to convert this set of base pair predictions into a coherent secondary structure. In addition to outperforming NNTM structure predictions on 16S rRNA by nearly 50 percentage points, we show that this method is able to generalize to many types of RNA, and that its flexibility allows it to make predictions that NNTM is fundamentally unable to capture.

Chapter 2 The Scaled Cayley Orthogonal RNN

This chapter describes the vanishing and exploding gradient problem, a known challenge in training recurrent neural networks, and presents a novel RNN training scheme that successfully avoids this difficulty. The tendency of gradients in an RNN to grow or decay exponentially over time, called the vanishing and exploding gradient problem, is a consequence of the BPTT algorithm that causes difficulty in training RNNs on long sequences with dependencies over many timesteps. One class of algorithms that effectively deals with vanishing and exploding gradients is orthogonal RNNs, RNNs that use alternate representations and update steps to constrain an RNN's recurrent weight matrix W to remain orthogonal throughout training. We present a novel orthogonal RNN architecture using a parametrization with a modified form of the Cayley transform. The resulting scaled Cayley orthogonal RNN (scoRNN) architecture is shown to outperform competing orthogonal RNN architectures on a benchmark training task, and to beat the popular long short-term memory (LSTM) architecture on a real-world training task.

2.1 Exploding and Vanishing Gradients

The RNN architecture described in Section 1.4 gives us a foundation from which to attack sequential learning tasks. However, in many cases the optimization of these networks remains difficult. This is particularly true when we have **long-range dependencies**, where input at a particular time may have a strong effect on the desired output many steps away.

In [5], Bengio et. al. provide an investigation into the gradients in an RNN that reveals a major obstacle to successful and efficient learning of long-range dependencies. This obstacle is most easily seen through computation of the gradient $\frac{\partial L}{\partial h^{(t_0)}}$ of the hidden state at some timestep t_0 . Since the hidden state is how information propagates from one timestep to the next, very large or very small values in this gradient will lead to trouble in training.

For simplification, suppose we are training an RNN on a sequence classification problem, so that the loss L depends only on the final output $y^{(\tau)}$. Thus,

$$\frac{\partial L}{\partial h^{(t_0)}} = \frac{\partial L}{\partial h^{(\tau)}} \frac{\partial h^{(\tau)}}{\partial h^{(t_0)}} = \frac{\partial L}{\partial h^{(\tau)}} \prod_{t=t_0+1}^{\tau} \frac{\partial h^{(t)}}{\partial h^{(t-1)}}.$$

From the RNN architecture described in Section 1.4, the gradient over any single timestep $t - 1$ to t is

$$\frac{\partial h^{(t)}}{\partial h^{(t-1)}} = \frac{\partial h^{(t)}}{\partial a^{(t)}} \frac{\partial a^{(t)}}{\partial h^{(t-1)}} = \text{diag}(\sigma'(a^{(t)})) W$$

and substituting this into the previous equation shows that the gradient is

$$\frac{\partial L}{\partial h^{(t_0)}} = \frac{\partial L}{\partial h^{(\tau)}} \prod_{t=t_0+1}^{\tau} (\text{diag}(\sigma'(a^{(t)})) W).$$

Taking norms, we have

$$\begin{aligned} \left\| \frac{\partial L}{\partial h^{(t_0)}} \right\|_2 &= \left\| \frac{\partial L}{\partial h^{(\tau)}} \prod_{t=t_0+1}^{\tau} (\text{diag}(\sigma'(a^{(t)})) W) \right\|_2 \\ &\leq \left\| \frac{\partial L}{\partial h^{(\tau)}} \right\|_2 \prod_{t=t_0+1}^{\tau} \|\text{diag}(\sigma'(a^{(t)}))\|_2 \|W\|_2 \\ &\leq \left\| \frac{\partial L}{\partial h^{(\tau)}} \right\|_2 \|W\|_2^{\tau-t_0} \prod_{t=t_0+1}^{\tau} \|\text{diag}(\sigma'(a^{(t)}))\|_2 \end{aligned}$$

Since $\text{diag}(\sigma'(a^{(t)}))$ is diagonal, its 2-norm is equal to the largest entry on its diagonal. With an activation function like ReLU, $\|\text{diag}(\sigma'(a^{(t)}))\|_2$ will be 1 as long as at least one activation in $a^{(t)}$ is positive, a reasonable assumption with batch training. Thus, the inequality simplifies to

$$\left\| \frac{\partial L}{\partial h^{(t_0)}} \right\|_2 \leq \left\| \frac{\partial L}{\partial h^{(\tau)}} \right\|_2 \|W\|_2^{\tau-t_0}.$$

Clearly, if $\|W\|_2 < 1$, the norm of the hidden state will decay exponentially over timesteps. If $\|W\|_2$ is much smaller than 1, we cannot expect meaningful gradient information to extend further than a few steps backward in time. This phenomenon is referred to as the **vanishing gradient problem**.

Though it is not as easily shown through bounding norms, the opposite problem can easily occur when $\|W\|_2 > 1$. In this case, gradients quickly become troublingly large, causing instability and eventually overflow errors. This is called the **exploding gradient problem**.

2.2 Solutions to Exploding and Vanishing Gradients

Broadly speaking, architectures that have exhibited significant success against the vanishing and exploding gradient problem fall into two categories: gating mechanisms, which incorporate additional parameters into the machine that control the flow of information from input to hidden state, and orthogonal RNNs, which preserve the norm of $h^{(t)}$ over time by restricting W to be orthogonal throughout training. The two techniques and their relative advantages are described below.

Long Short-Term Memory & Gated RNNs

RNN architectures in this section are distinguished by the inclusion of additional intermediate vectors called **gates**. Unlike the vanilla RNN, in which we multiply the

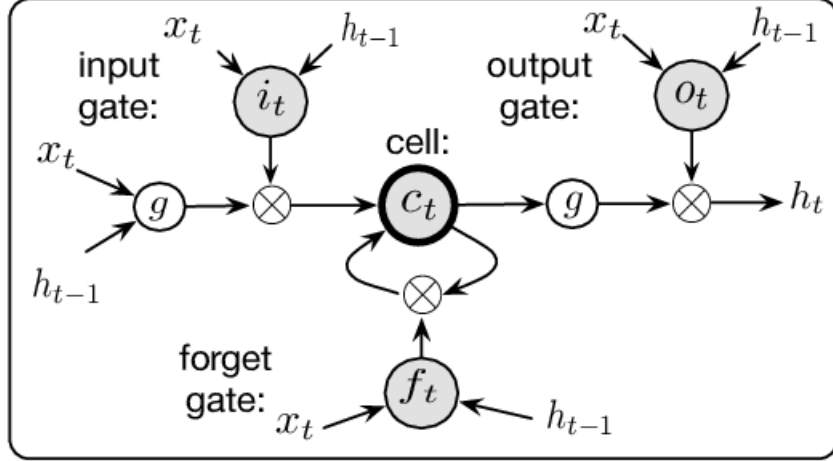


Figure 2.1: Diagram of a Long Short-Term Memory (LSTM) recurrent step. Retrieved from [8].

hidden state h by the recurrent matrix W at each step, these architectures introduce additional intermediate vectors that control the flow of information from input to hidden state. This comes with the computational and memory storage cost of additional parameters.

Gating mechanisms were introduced with the Long Short-Term Memory (LSTM) framework [25] over 20 years ago. LSTMs remain by far the most popular RNN variant, and are responsible for state of the art results on sequence learning tasks in many domains, including NLP and bioinformatics [34].

Compared to the standard RNN, the LSTM uses four times as many input and recurrent parameters, and keeps track of an additional memory cell $c^{(t)}$ of the same size as the hidden state $h^{(t)}$. A diagram of the LSTM architecture is shown in Figure 2.1. The equations used to generate the output of an LSTM are given below.

$$\begin{aligned}
 g^{(t)} &= \sigma(U_g x^{(t)} + W_g h^{(t-1)} + b_g) \\
 i^{(t)} &= \sigma(U_i x^{(t)} + W_i h^{(t-1)} + b_i) \\
 f^{(t)} &= \sigma(U_f x^{(t)} + W_f h^{(t-1)} + b_f) \\
 c^{(t)} &= c^{(t-1)} \circ f^{(t)} + g^{(t)} \circ i^{(t)} \\
 o^{(t)} &= \sigma(U_o x^{(t)} + W_o h^{(t-1)} + b_o) \\
 h^{(t)} &= \tanh(c^{(t)}) \circ o^{(t)} \\
 y^{(t)} &= V h_t + c
 \end{aligned}$$

We can think of the update to the hidden state $h^{(t)}$ as occurring in three stages:

First, from the input $x^{(t)}$ and previous hidden state $h^{(t-1)}$, we generate four intermediate vectors: the candidate cell state $g^{(t)}$, as well as the input gate $i^{(t)}$, forget gate $f^{(t)}$, and output gate $o^{(t)}$. However, we note that unlike the arbitrary activation

function in the regular RNNs, here σ always represents the sigmoid function, since for theoretical purposes we would like elements of these vectors to be in $(0, 1)$.

We combine $g^{(t)}$, $i^{(t)}$, and $f^{(t)}$ with the previous cell state $c^{(t-1)}$ to generate the new state $c^{(t)}$. We can think of $g^{(t)}$ as information from $x^{(t)}$ that we are potentially interested in adding to the current cell state. The multiplication of $g^{(t)}$ by the input gate $i^{(t)}$ controls which of that information is added to the cell state. Analogously, multiplication of $c^{(t-1)}$ by $f^{(t)}$ decides how much of the previous cell state we would like to preserve.

Finally, the hidden state is generated with the new cell state $c^{(t)}$ and the output gate $o^{(t)}$. The output gate controls how much of the new cell state we send to the hidden state $h^{(t)}$.

Computation of the gradient of all of these parameters is rather complicated; see [25] for gradient derivations and theoretical justification for this architecture.

Several heuristics specific to learning long-term dependencies with LSTMs have been introduced. One is gradient clipping [42], in which we scale gradients so that the largest values do not exceed a given threshold. This technique has been shown to successfully combat exploding gradients, but does not fix the problem of vanishing gradients. Another common practice is to initialize the forget gate bias b_f with large positive values. Intuitively, we can see that this will encourage the machine to preserve information in the hidden state by making elements of $f^{(t)}$ close to 1, and thus information from $c^{(t-1)}$ is approximately preserved when multiplied by $f^{(t)}$. A more rigorous theoretical explanation for the success of this technique is explored in [10].

Though the LSTM is by far the most well-known gated RNN architecture, several others have shown promising results. Most notable among its contenders is the gated recurrent unit (GRU), introduced in [11]. The GRU dispenses with the memory cell and has three times the input and recurrent parameters of a vanilla RNN, as compared to four times as many for an LSTM. Despite this, GRUs have exhibited comparable performance to LSTMs.

Though primarily presented for the purposes of exploring the theoretical foundations of gated RNNs, we also note the minimal RNN [10]. This architecture is a further simplification of the GRU, and uses only double the recurrent parameters of the vanilla RNN to achieve similar results as the LSTM and GRU on benchmark NLP tasks.

Orthogonal RNNs

The computation in Section 2.1 shows that exploding and vanishing gradients are a consequence of repeated multiplication by the recurrent matrix W . Exponential growth or decay of this gradient’s norm over timesteps would not occur if $\|W\|_2 = 1$, a necessary and sufficient condition for W to be orthogonal. This gives rise to the idea of initializing W with an orthogonal matrix. However, in the absence of additional constraints, even a single step of an additive update scheme like stochastic gradient descent will destroy the orthogonality of W .

This motivates the idea of using alternate training methods to preserve the orthogonality of W throughout training. This area was pioneered with the introduction of the unitary evolution RNN [4] in 2015. In the intervening three years, a host of update schemes that maintain orthogonality have been presented. We briefly consider each below.

Unitary Evolution RNN (uRNN)

The unitary evolution RNN (uRNN, or restricted capacity uRNN) [4] replaces W with a product of unitary matrices:

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

Here, D_i is a diagonal matrix, R_i is a reflection matrix, P_i is a fixed permutation matrix, and \mathcal{F} is a discrete Fast Fourier transform. Each of the three diagonal matrices is kept unitary by representing diagonal entries with $e^{i\omega_j}$, where each $\omega_j \in \mathbb{R}$ is trainable. Similarly, both reflection matrices are defined by $I - 2 \frac{vv^*}{\|v\|_2^2}$, where $v \in \mathbb{C}^{n_h}$ is trainable.

The n_h trainable parameters in each D and the $2n_h$ trainable parameters in each R make for a total of $7n_h$ trainable parameters.

Full-Capacity uRNN

In [61], Wisdom et. al. begin by noting that the $7n_h$ trainable parameters in the uRNN is insufficient for representing an arbitrary unitary matrix in $\mathbb{C}^{n_h \times n_h}$ for realistically sized n_h . To remedy this, they present the full capacity uRNN, an alternate uRNN architecture based around work in [52] regarding matrix optimization on the Stiefel manifold. In this architecture, $W \in \mathbb{C}^{n_h \times n_h}$, and the $2n^2$ parameters of W are directly trained using the following multiplicative update, where $\frac{\partial L}{\partial W} = [\frac{\partial L}{\partial W_{i,j}}]$ and $*$ denotes the conjugate transpose operation.

$$\begin{aligned} B &\leftarrow \frac{\partial L}{\partial W}^* W - W^* \frac{\partial L}{\partial W} \\ W &\leftarrow \left(I + \frac{\lambda}{2} B\right)^{-1} \left(I - \frac{\lambda}{2} B\right) W \end{aligned}$$

This update keeps W on the Stiefel manifold throughout training, and experiments presented show that it is able to outperform [4] on several benchmark training tasks.

Other Work

Since the full capacity uRNN, there has been a host of other parametrization schemes and optimization methods to force recurrent matrix orthogonality. Most notably, these include the efficient unitary RNN (EURNN) [29], which parametrizes using a product of Givens rotation matrices [15], and the orthogonal RNN (oRNN) [39], which parametrizes with a product of Householder reflection matrices [15]. We note that both of these architectures are presented with hyperparameters that control the

capacity of the machine; best results from these machines are usually found in a restricted capacity setting.

Other work in this area uses orthogonality to improve training without necessarily constraining the recurrent matrix to strict orthogonality throughout training. One such architecture is explored in [56]; here, the eigenvalues of W are allowed to deviate from norm 1 by a small amount. Other architectures combine an orthogonal/unitary weight matrix with a gating mechanism. A version with real parameters was introduced as the gated orthogonal recurrent unit (GORU) [28], while a complex arithmetic version was presented in [62].

2.3 Parametrization with the Cayley Transform

The Cayley Transform

The Cayley transform refers to a number of related transformations in several domains, including complex numbers, quaternions, and others. In this chapter, the Cayley transform will refer to the map with the domain of real matrices. We note that nearly all of the results we present below have analogues for complex matrices.

Let $\mathcal{C} : \mathbb{R}^{n_h \times n_h} \rightarrow \mathbb{R}^{n_h \times n_h}$ be the **Cayley transform**. This map is given by

$$\mathcal{C}(W) = (I + W)^{-1}(I - W).$$

The Cayley transform is undefined for matrices with -1 eigenvalues, as this would result in 0 as an eigenvalue for $I + W$, rendering it singular and thus not invertible. The map is defined for all other matrices in $\mathbb{R}^{n_h \times n_h}$.

Theorem 1. *Let $W \in \mathbb{R}^{n_h \times n_h}$. All of the following hold:*

- (a) *If λ is an eigenvalue of W , then $\frac{\lambda-1}{\lambda+1}$ is an eigenvalue of $\mathcal{C}(W)$.*
- (b) *\mathcal{C} is an involution on its domain.*
- (c) *\mathcal{C} is a bijection between orthogonal matrices without -1 eigenvalues and skew-symmetric matrices.*

Proof. Part (a): Let x be an eigenvector of W , and let λ be its associated eigenvalue. Since every x is an eigenvector of I , x is an eigenvector of $I - W$ and $(I + W)^{-1}$, so

$$(I + W)^{-1}(I - W)x = (\lambda - 1)(I + W)^{-1}x = \frac{\lambda - 1}{\lambda + 1}x.$$

Part (b): Note that by factoring $(I + W)^{-1}$ on the left,

$$I + \mathcal{C}(W) = I + (I + W)^{-1}(I - W) = (I + W)^{-1}(I + W + I - W) = 2(I + W)^{-1}.$$

and similarly, $I - \mathcal{C}(W) = 2(I + W)^{-1}W$. Using this, we get

$$\begin{aligned}
\mathcal{C}(\mathcal{C}(W)) &= (I + \mathcal{C}(W))^{-1}(I - \mathcal{C}(W)) \\
&= (2(I + W)^{-1})^{-1}(2(I + W)^{-1}W) \\
&= (I + W)(\tfrac{1}{2})(2)(I + W)^{-1}W \\
&= W.
\end{aligned}$$

Part (c): Let W be an orthogonal matrix without -1 eigenvalues, and let $e^{i\theta}$ be an eigenvalue of W . By part (a), the following is an eigenvalue of $A := \mathcal{C}(W)$:

$$\frac{e^{i\theta} - 1}{e^{i\theta} + 1} = \frac{e^{i\theta} - 1}{e^{i\theta} + 1} \left(\frac{e^{-i\theta} + 1}{e^{-i\theta} + 1} \right) = \frac{e^{i\theta} - e^{-i\theta}}{2 + e^{i\theta} + e^{-i\theta}}$$

The numerator is imaginary, and the denominator is real, so this eigenvalue of A is imaginary. Thus, the spectrum of $\mathcal{C}(W)$ is contained in the imaginary axis. This is a necessary and sufficient condition for $\mathcal{C}(W)$ to be **skew-symmetric**, that is, $A^T = -A$. Conversely, let vi be an arbitrary eigenvalue of A . Both $vi - 1$ and $vi + 1$ have modulus $\sqrt{1 + v^2}$, so the modulus of their quotient will be 1, and the eigenvalue $\frac{vi-1}{vi+1}$ of $\mathcal{C}(A)$ will lie on the unit circle. Thus, $\mathcal{C}(A)$ is orthogonal.

Thus, the Cayley transform is a bijection between the set of orthogonal matrices in $\mathbb{R}^{n_h \times n_h}$ without -1 eigenvalues and the set of skew symmetric matrices in $\mathbb{R}^{n_h \times n_h}$. \square

Part (c) of this theorem presents a potential method for parametrizing the recurrent matrix W of an RNN: we can regard W as a function of its Cayley transform A , and entries of A as trainable parameters of the machine. This parametrization is particularly attractive from a machine learning standpoint, as the set of skew symmetric matrices is closed under addition. Thus, we can use standard gradient descent algorithms, or variants like RMSprop or Adam, to train A while keeping W orthogonal.

The limitations of this parametrization method are evident in the bijection: it is impossible to represent orthogonal matrices with -1 eigenvalues in this manner. From a theoretical standpoint, this is not a particularly dire concern; the set of matrices with -1 eigenvalues is a set of measure 0 in $\mathbb{R}^{n_h \times n_h}$, and we can reach matrices with eigenvalues arbitrarily close to -1 with skew-symmetric matrices.

However, in practice, representing a matrix W with eigenvalues near -1 requires extraordinarily large values of A . As a concrete example, we consider an orthogonal W and skew-symmetric A , both in $\mathbb{R}^{2 \times 2}$, that form a bijective pair under the Cayley transform. If A has off diagonal entries a and $-a$, it is simple to check that the eigenvalues of A are $\pm ai$. Thus, by Theorem 1a, eigenvalues of W are

$$\pm \frac{1 + ai}{1 - ai} = \pm \frac{(1 - ai)^2}{1 + a^2} = \frac{1 - a^2}{1 + a^2} \pm \frac{-2a}{1 + a^2}i.$$

Let λ be either of the eigenvalues of W ; then $\text{Re}(\lambda) = \frac{1-a^2}{1+a^2}$. From this relationship, we can see that if we wish for W to have eigenvalues with real part -0.99 , we will need $a \approx \pm 14.1$. Gradient descent algorithms will be extraordinarily slow to learn

this A matrix. In a more extreme example, a λ with $\text{Re}(\lambda) = -0.9999$ will require $a \approx \pm 447.21$, far outside the realm of possibility for a neural network to learn.

More generally, since trainable neural network parameters have small entries, the relationship between eigenvalues of A and W means that parametrization with the Cayley transform is biased toward W matrices with eigenvalues near 1. Later, we will see empirical results that suggest this is not desirable for learning tasks.

The Scaled Cayley Transform

The drawbacks associated with parametrization through the Cayley transform can be remedied with a small adjustment, which comes from [41].

Theorem 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 .

In practice, -1 entries of D negate columns of the Cayley transform $\mathcal{C}(A)$. Thus, this theorem shows that any orthogonal matrix W can be represented by negating columns of $\mathcal{C}(A)$ for some A , and further guarantees that this can be done for an A matrix with relatively small entries, assuming we have chosen D correctly.

We will use the transformation in Theorem 2 to parametrize W with A , leaving D fixed. If we are to train the parameters in A , then we will need to know how to backpropagate through the scaled Cayley transform to find the matrix derivative $\frac{\partial L}{\partial A}$. The following theorem gives this matrix derivative.

Theorem 3. *Let L be a differentiable loss function for an RNN with recurrent weight matrix W . Let $W = W(A) := (I + A)^{-1}(I - A)D$ where A is skew-symmetric and D is diagonal and fixed. Then the gradient $\frac{\partial L}{\partial A} = \left[\frac{\partial L}{\partial A_{i,j}} \right]$ is*

$$\frac{\partial L}{\partial A} = V^T - V \quad V = (I + A)^{-T} \frac{\partial L}{\partial W} (D + W^T).$$

where $\frac{\partial L}{\partial W} = \left[\frac{\partial L}{\partial W_{i,j}} \right] \in \mathbb{R}^{n_h \times n_h}$.

Proof. Consider the derivative of the (i, j) entry of $\frac{\partial L}{\partial A}$:

$$\frac{\partial L}{\partial A_{i,j}} = \sum_{k,l=1}^{n_h} \frac{\partial L}{\partial W_{k,l}} \frac{\partial W_{k,l}}{\partial A_{i,j}} = \text{tr} \left(\frac{\partial L}{\partial W}^T \frac{\partial W}{\partial A_{i,j}} \right)$$

We left-multiply by $(I + A)^{-1}$ to get $(I + A)W = (I - A)D$. Taking the derivative with respect to $A_{i,j}$ and rearranging gives

$$\begin{aligned}
\frac{\partial W}{\partial A_{i,j}} + \frac{\partial A}{\partial A_{i,j}}W + A\frac{\partial W}{\partial A_{i,j}} &= -\frac{\partial A}{\partial A_{i,j}}D \\
(I + A)\frac{\partial W}{\partial A_{i,j}} &= -\frac{\partial A}{\partial A_{i,j}}D - \frac{\partial A}{\partial A_{i,j}}W \\
\frac{\partial W}{\partial A_{i,j}} &= -(I + A)^{-1}\frac{\partial A}{\partial A_{i,j}}(D + W)
\end{aligned}$$

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 $\frac{\partial A}{\partial A_{i,j}} = E_{i,j} - E_{j,i}$, giving

$$\frac{\partial W}{\partial A_{i,j}} = (I + A)^{-1}(E_{j,i} - E_{i,j})(D + W).$$

With the above, we can now directly calculate the derivative $\frac{\partial L}{\partial A_{i,j}}$. Below, we use the cyclic invariance of traces, as well as the fact that $\text{tr}(ZE_{i,j}) = z_{j,i}$.

$$\begin{aligned}
\frac{\partial L}{\partial A_{i,j}} &= \text{tr} \left(\frac{\partial L}{\partial W}^T \frac{\partial W}{\partial A_{i,j}} \right) \\
&= \text{tr} \left(\frac{\partial L}{\partial W}^T (I + A)^{-1}(E_{j,i} - E_{i,j})(D + W) \right) \\
&= \text{tr} \left((D + W) \frac{\partial L}{\partial W}^T (I + A)^{-1}(E_{j,i} - E_{i,j}) \right) \\
&= \text{tr} \left((D + W) \frac{\partial L}{\partial W}^T (I + A)^{-1}E_{j,i} \right) - \text{tr} \left((D + W) \frac{\partial L}{\partial W}^T (I + A)^{-1}E_{i,j} \right) \\
&= \left[(D + W) \frac{\partial L}{\partial W}^T (I + A)^{-1} \right]_{i,j} - \left[(D + W) \frac{\partial L}{\partial W}^T (I + A)^{-1} \right]_{j,i} \\
&= \left[(D + W) \frac{\partial L}{\partial W}^T (I + A)^{-1} \right]_{i,j} - \left[(I + A)^{-T} \frac{\partial L}{\partial W} (D + W^T) \right]_{i,j}
\end{aligned}$$

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

2.4 The scoRNN Architecture

We use results from the previous section to introduce our novel orthogonal RNN architecture, called the scaled Cayley orthogonal RNN, or scoRNN. The forward pass of this RNN is identical to the vanilla RNN given in Section 1.4, as is training for the input and output parameters. Instead of directly training the recurrent matrix W , we parametrize it through the scaled Cayley transform $W = (I + A)^{-1}(I - A)D$. Here,

W is a function only of A , as D a fixed diagonal matrix, chosen before training. Since the placement of $+1$ and -1 entries on the diagonal of D does not affect training, this choice of D is reduced to a single hyperparameter ρ that refers to the number of -1 entries in D .

The training of A is shown below. At each iteration of training, we use the standard BPTT algorithm to find $\frac{\partial L}{\partial W}$. With this matrix derivative in hand, we use Theorem 3 to find the matrix derivative $\frac{\partial L}{\partial A}$. We then update A using our optimization algorithm of choice, and finally use this new A to generate W .

$$\begin{aligned}\frac{\partial L}{\partial A} &\leftarrow (D + W) \frac{\partial L}{\partial W}^T (I + A)^{-1} - (I + A)^{-T} \frac{\partial L}{\partial W} (D + W^T) \\ A &\leftarrow A - \eta \frac{\partial L}{\partial A} \\ W &\leftarrow (I + A)^{-1} (I - A) D\end{aligned}$$

This training process has several benefits relative to other unitary and orthogonal RNNs. As previously noted, this is an additive update step, and thus is compatible optimizers like RMSprop [54] that have been successfully employed in deep learning contexts. Another benefit is the regeneration of W through the Cayley transform, which ensures that orthogonality is maintained to machine precision at each iteration of training. This is contrast to the full-capacity uRNN, whose multiplicative update step is susceptible to gradual loss of orthogonality over time.

Complexity

The time complexity of the BPTT algorithm is $O(bLn_h^2)$, where b is minibatch size and ℓ is the length of the sequence. To update A , our method requires several matrix multiplications and the requires matrix inverse $(I + A)^{-1}$, which is order $O(n_h^3)$ [15]. However, these steps are performed after the BPTT algorithm is complete, giving this method $O(bLn_h^2 + n_h^3)$ complexity. Unless the hidden size n_h is much larger than bL , this does not represent a significant increase in time complexity. In particular, on problems with long-range dependencies where orthogonal RNNs will be useful, L will often be in the hundreds or thousands, making bL much larger than n_h for even a small batch size.

Activation Function

In addition to pioneering the use of unitary RNNs, [4] introduced a novel activation function, called the modReLU, designed to be compatible with unitary RNNs. Unlike activation functions like tanh and ReLU, modReLU separates the bias b from the activation z :

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

The modReLU was originally introduced for use with the restricted-capacity uRNN, which has complex parameters. Regardless, we found increased performance in the real case with the scORNN. In the real case this simplifies to

$$\sigma_{\text{modReLU}}(z) = \begin{cases} \text{sign}(z)(z + b) & \text{if } z + b > 0 \\ 0 & \text{if } z + b < 0 \end{cases}$$

Initialization

We tested several initialization methods, and found the most effective to be one inspired by [23]. We initialize A as a block diagonal matrix with 2×2 blocks B_j :

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

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

Experiments

There exist several benchmark tasks for orthogonal RNNs that test their ability to learn long-range dependencies. Many of these are synthetic tasks that gated architectures like the LSTM fail to excel at. The remainder of this chapter is devoted to presenting the results from several such experiments.

2.5 Copying Problem Experiments

This section presents the results of performing several experiments on the copying problem, a benchmark task testing RNNs’ ability to learn long-range dependencies. This experiment follows descriptions found in other work exploring orthogonal RNNs [4, 29, 61], and tests an RNN’s ability to reproduce a sequence seen many timesteps earlier. In the problem setup, there are 10 input classes, which we denote using the digits 0-9, with 0 being used as a ‘blank’ class and 9 being used as a ‘marker’ class. In practice, these digits are represented with a one-hot encoding. The RNN receives an input sequence of length $T + 20$. This sequence consists of entirely zeros, except for the first ten elements, which are uniformly sampled from classes 1-8, and a 9 placed ten timesteps from the end. The goal for the machine is to output zeros until it sees a 9, at which point it should output the ten elements from the beginning of the input sequence. An example pair of input-output sequences with $T = 5$ is given below.

```
Input:      8 3 4 1 2 5 5 6 3 2 0 0 0 0 0 9 0 0 0 0 0 0 0 0
Target Output: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 3 4 1 2 5 5 6 3 2
```

Learning long-range dependencies are central to this learning task, as the machine must see an input and then output it $T+10$ timesteps later. As T increases, exploding or vanishing gradients become a major obstacle to successful learning.

Table 2.1: Table of hidden sizes and total number of trainable parameters of each machine used in the copying problem experiment

Machine	Hidden Size (n_h)	Trainable Parameters
LSTM	60	$\approx 22k$
scoRNN	190	$\approx 22k$
Rest. uRNN	470	$\approx 22k$
Full uRNN	128	$\approx 22k$
EURNN	512	$\approx 10.8k$

If information from the beginning of the sequence is not available, one strategy for minimizing the loss is to output 0 until the machine sees a 9, and then outputting 10 elements randomly sampled from classes 1-8. The expected cross-entropy for such a strategy is $\frac{10 \log(8)}{T+20}$. We regard this strategy as the baseline for this task; in practice, it is common to see gated RNNs such as LSTMs converge to this local minimum.

We tested the scoRNN against an LSTM, both restricted capacity and full capacity uRNNs, the EURNN, and the oRNN. These experiments use a single hidden layer and do not include any regularization methods like a loss function regularization term, dropout, or batch normalization. Consistent with reports from [39], we found that oRNN did not give satisfactory results on this problem, and we omit its performance from the figures presented later in this section. For the competing unitary and orthogonal RNNs, we used hyperparameter configurations and hidden state sizes as reported in their experiments on this problem [4, 29, 61]. Most of these machine sizes are standardized to match the number of trainable parameters in the machine to be ≈ 22000 ; we chose hidden sizes n_h for the LSTM and scoRNN in accordance with this, and performed a grid search to find best hyperparameters. In particular, we found the best performance with the scoRNN came from $\rho = n_h/2$. The hidden state size and number of trainable parameters in each model are given in Table 2.1.

Figure 2.2 compares each model’s performance for $T = 1000$ and $T = 2000$, with the cross entropy of the baseline strategy shown as a dashed line. In both cases, cross entropy for the LSTM, restricted-capacity uRNN, and EURNN remains at the baseline or does not entirely converge over the entire experiment. For the $T = 1000$ test, the full-capacity uRNN and scoRNN converge quickly to zero entropy solutions, with the full-capacity uRNN converging slightly faster. For $T = 2000$, the full-capacity uRNN remains at the baseline for several thousand iterations, but is eventually able to find a solution that bypasses the baseline. In contrast, the scoRNN error has a smooth convergence that passes the baseline within the first few hundred iterations of training.

scoRNN Hyperparameter Comparison

We also exhibit a small experiment on the copying problem designed to display the effects of modulating the scoRNN hyperparameter ρ , the number of -1 entries in the scaling matrix D . We tested several hyperparameter configurations of scoRNN on the copying problem with $T = 100$. These configurations were identical to the machine

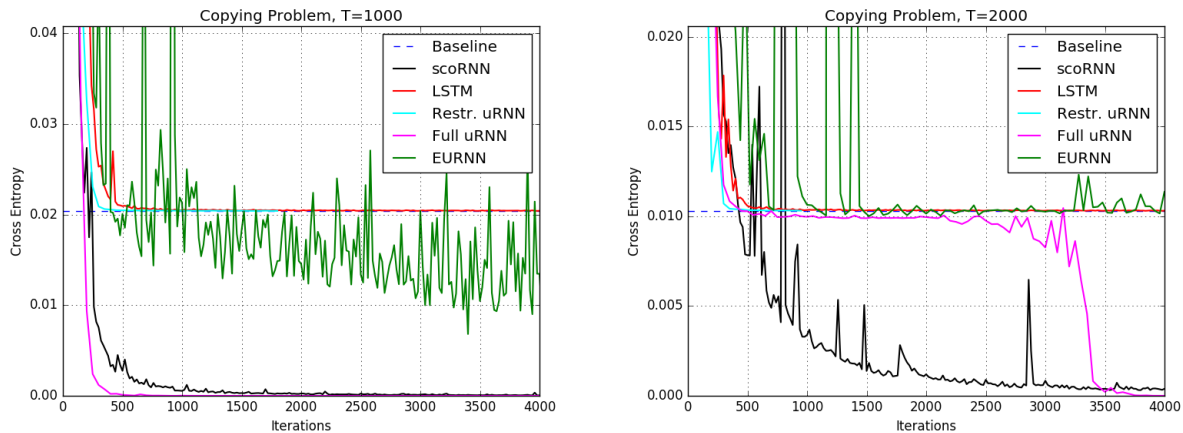


Figure 2.2: Cross entropy of each machine on the copying problem with $T = 1000$ (left) and $T = 2000$ (right).

used in the previous experiment except for the hidden size, which we slightly increase to 200, and ρ , which we vary among 0 and 200 in intervals of 50.

The results of this experiment are displayed in Figure 2.3. As noted in the previous section, a scoRNN with $\rho = n_h/2$ gives the best performance on this task. Immedi-

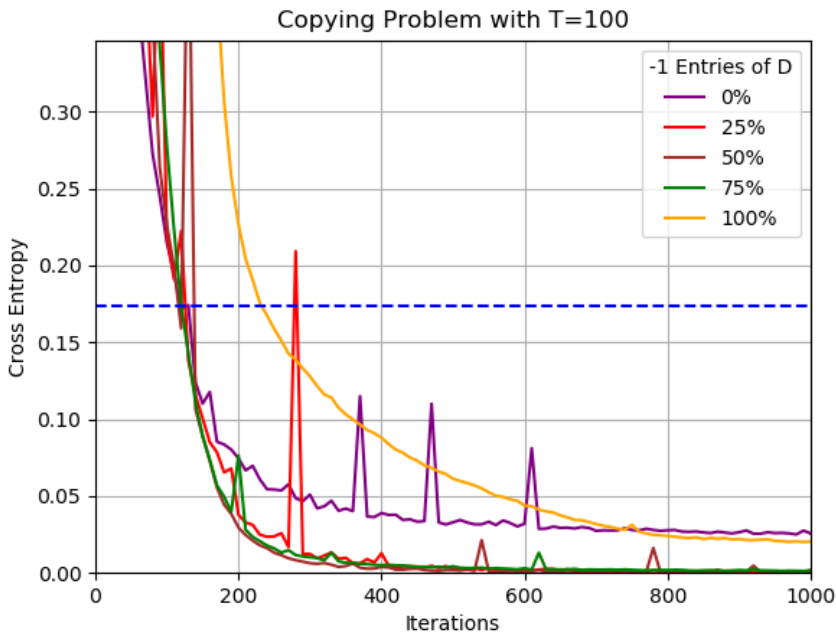


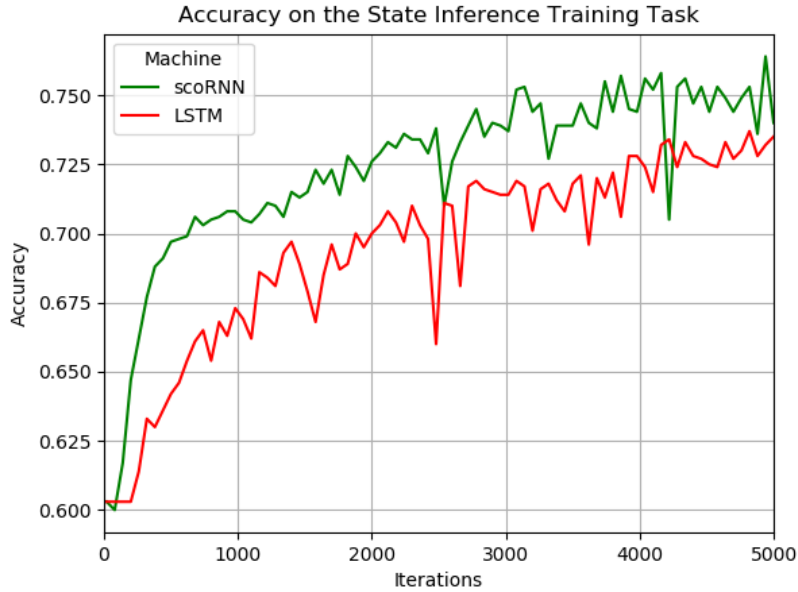
Figure 2.3: Several scoRNN models on the copying problem with $T = 100$ for varying values of ρ

ately we see that the best performance is given when $\rho = n_h/2$, followed closely by $\rho = n_h/4$ and $\rho = 3n_h/4$. More interesting is performance at the extremes $\rho = 0$ and $\rho = n_h$. When $\rho = 0$, training is initially comparable to the more successful models, but stalls after a few hundred iterations. On the other hand, initial training with $\rho = n_h$ is by far the slowest to start, taking 100 more iterations than all other models to cross the baseline, but it continues to improve and eventually surpasses $\rho = 0$.

These results provide empirical evidence that the scaling matrix D is beneficial to training: the low performing $\rho = 0$ corresponds to the case where $D = I$, meaning that the Cayley transform is unscaled. Recall that we initialize A in such a way that $\mathcal{C}(A)$ has eigenvalues distributed uniformly on the right half of the unit circle, and that D reflects ρ of these eigenvalues across the imaginary axis. This experiment suggests that successful training on a problem with long-range dependencies like this requires a recurrent matrix with eigenvalues with negative real part, and that the scoRNN struggles to reach these eigenvalues in the absence of any scaling. In the case of the copying problem, our best performance of $\rho = n/2$ initializes W with eigenvalues uniformly distributed around the entire unit circle.

2.6 RNA State Inference Experiments

Figure 2.4: Performance of scoRNN vs. an LSTM on the simplified state inference problem



We present a simplified version of the state inference task addressed in depth in Chapter 4 to test the scoRNN architecture against an LSTM on a learning task involving real world data. We use the same training and test sets as in state inference experiments in Chapter 4, but we cut off sequences after the first 300 elements of the input and output sequences. The machines we use are RNNs with one hidden

layer, and not bidirectional. We emphasize that the results in this section cannot be directly compared to those in Chapter 4, as we have simplified both the training task and the machines here.

As in the copying problem, we match the number of trainable parameters in each machine through our choice of the hidden size; here, we have $n_h = 360$ for the scoRNN, and $n_h = 128$ for the LSTM. Both machines were trained with RMSprop. We again use $\rho = n_h/2$ for the scoRNN to initialize with eigenvalues of W distributed uniformly around the unit circle.

Test set accuracy from each of these models throughout the training process is displayed in Figure 2.4. The scoRNN is clearly much faster at training, as it jumps up to 70% accuracy within 500 training iterations. Though the gap between scoRNN and LSTM performance shrinks during training, it still maintains an advantage over the LSTM after 5000 iterations.

Chapter 3 RNA Sequences and Structures

The final two chapters of this work explore deep learning methods for solving the problem of RNA secondary structure inference. This chapter gives necessary background for understanding RNA secondary structure inference, gives a formal definition to the problem, and presents the existing state of the art methods for secondary structure inference in the bioinformatics community. In particular, we detail the inner workings of the nearest neighbor thermodynamic model (NNTM), which uses hand-tuned parameters based on the physical and chemical properties of RNA to assign each RNA sequence an energy term. NNTM then efficiently produces a predicted structure based on this energy using a dynamic programming algorithm that exploits the loop structure of RNA, which we present in some detail.

We will use the concepts introduced in this chapter to inform our model choices and analyze results when we present our own secondary structure inference methods in Chapter 4, where we combine deep learning with NNTM predictions, and Chapter 5, where we use a neural network that directly produces base pair and structure predictions.

3.1 RNA

Ribonucleic acid (RNA) is one of the three essential macromolecules involved in the propagation of genetic information. See [13] for an introduction to cellular biology. In this work, it will be sufficient for us to represent an RNA sequence of length L as a sequence $r = (r_1, \dots, r_L)$. Each sequence element r_i is a **nucleotide** or **base**. In an RNA sequence, the vast majority of nucleotides are one of four types: adenine, cytosine, guanine, and uracil, denoted A, C, G, and U, respectively. We denote the rare cases where other nucleotides appear in RNA, or cases where the nucleotide is not known, with X . Thus, $r_i \in \{A, C, G, U, X\}$.

Nucleotides in an RNA sequence form hydrogen bonds with one another to create a **base pair**. Unlike double-stranded DNA, whose nucleotides pair with those from another nucleic acid sequence, RNA found in nature is usually single-stranded: base pairs form among nucleotides in the same RNA sequence. As in all nucleic acids, the most common base pairs are between cytosine and guanine (C-G pairs), and between adenine and uracil (A-U pairs). These are called Watson-Crick base pairs, or canonical base pairs. In RNA, we also frequently see bonds between guanine and uracil (G-U pairs), called wobble pairs, so named for their instability relative to Watson-Crick base pairs. Within an RNA sequence, we refer to a base pair using the indices of its bases in r ; that is, a base pair between r_i and r_j is denoted (i, j) , with $i < j$. We refer to the property of being paired or unpaired as the **state** of the nucleotide.

Numerous base pairs form in a single strand of RNA, leading to a folded **RNA structure**. Unlike more flexible macromolecules like proteins, RNA is largely stable at room temperature, and its structure is dependent only on the sequence itself, rather

than external conditions. The structure an RNA sequence forms in nature is referred to as its **native structure**.

Structure can refer to any of a number of abstractions of the folded RNA sequence:

- **Tertiary structure** is the relative position of nucleotides in three-dimensional space.
- **Secondary structure** is the projection of the tertiary structure onto two-dimensional space. A sufficient description of secondary structure is the set of base pairs among a sequence's bases.
- **Primary structure** is the projection of tertiary structure onto one-dimensional space. This is described by the sequence of nucleotides.

An RNA sequence's native secondary structure gives important information about its function[20, 38], but is often difficult to observe directly [9, 19]. This motivates the problem of **secondary structure prediction** or **secondary structure inference**: given an RNA sequence r (that is, given its primary structure), we would like to find its native secondary structure S . In the following, we occasionally refer to this simply as structure prediction. This is the central problem that the remainder of this work will address.

More formally, a secondary structure S is a set of base pairs between nucleotides in r where:

1. Each nucleotide in r is in at most one base pair;
2. Each base pair is a Watson-Crick base pair or a wobble pair, that is, if $(i, j) \in S$ then $\{r_i, r_j\} \in \{\{A, U\}, \{C, G\}, \{G, U\}\}$;
3. There are at least 3 bases between each base in a base pair, that is, $|j - i| > 3$ for all $(i, j) \in S$;
4. Base pairs follow the **nesting property**: for any two base pairs (i, j) and (i', j') in S where $i < i'$, either

$$i < i' < j' < j \quad \text{or} \quad i < j < i' < j'.$$

The nesting property is most easily understood graphically, as in the arc diagram in Figure 3.1, which shows an RNA sequence with secondary structure

$$\{(1, 22), (2, 21), (3, 20), (5, 13), (6, 12), (7, 10), (14, 18)\}$$

as seen by the edges connecting these nucleotides. A collection of base pairs has the nesting property if and only if we can draw an arc diagram without any intersecting arcs; for example, adding the base pair $(16, 19)$ to the set above would disrupt the nesting property.

We note that the nesting property is a slight simplification of the true nature of RNA structure. In rare cases, base pairs that do not respect the nesting property may

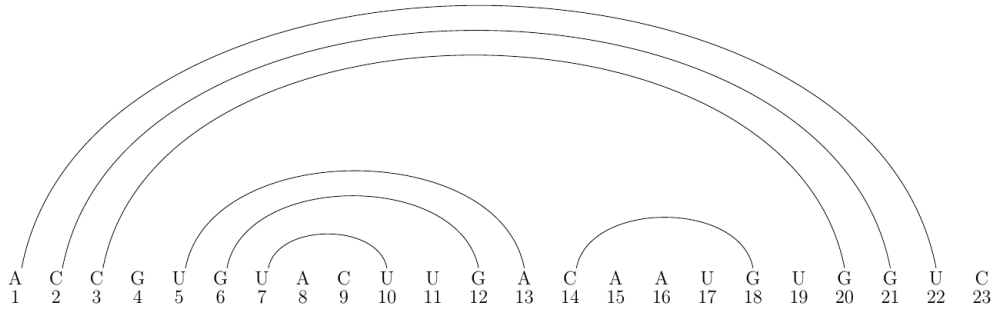


Figure 3.1: An RNA sequence of length $L = 23$. Nucleotides are shown above their index, and arcs are drawn between nucleotides in a base pair.

form in nature. These pairs, referred to as **pseudoknots**, are relatively rare among base pairs, and are frequently omitted entirely from the secondary structure [26]. Unless otherwise noted, we assume that all secondary structures have the nesting property. We will briefly return to the topic of pseudoknots and how to predict them in Chapter 5.

The nested nature of base pairs in a secondary structure allows us to partition an RNA sequence into subsets referred to as **loops**. Let $(i, j) \in S$. The **loop closed by** (i, j) , denoted $L_{i,j}$, is the set of nucleotides v such that $i < v < j$ and there exists no other base pair $(i', j') \in S$ such that $i < i' < v < j' < j$. More intuitively, a nucleotide is in loop $L_{i,j}$ if it can ‘look up’ in the arc diagram and ‘see’ base pair (i, j) . The **null loop** L_0 contains all those nucleotides not in any loop enclosed by a base pair. Thus, loops form a partition of r : every nucleotide in r is in exactly one loop, and each base pair in S encloses at least one nucleotide of r .

Base pairs between nucleotides inside $L_{i,j}$ are said to be enclosed by (i, j) , but note that the loop definition above excludes the base pair (i, j) from the loop $L_{i,j}$ that it closes. Partitioning a structure into loops allows us to classify the various substructures it creates, based on the number and arrangement of base pairs each loop encloses. For a loop $L_{i,j}$ and secondary structure S , some classes of loops are listed below.

- A **hairpin loop** is a loop with no paired nucleotides: $L_{i,j} = \{i + 1, \dots, j - 1\}$.
- A **stacked (base) pair** is a loop that only encloses two base paired nucleotides: $L_{i,j} = \{i + 1, j - 1\}$, and $(i + 1, j - 1) \in S$.
- An **internal loop** contains a pair of bonded nucleotides i' and j' , and there exist unpaired nucleotides between i and i' , and between j and j' .
- A **bulge loop** contains a pair of bonded nucleotides i' and j' , and either $i' = i + 1$ or $j' = j - 1$, that is, all unpaired nucleotides are on one side of (i, j) and (i', j') .
- A **multiloop** contains two or more bonded pairs of nucleotides.

Due to their stabilizing nature, we will often see stacked pairs occurring in tightly nested clusters, such as

$$\{(i, j), (i + 1, j - 1), \dots, (i + k, j - k)\} \subset S.$$

This collection of base pairs is a **helix**. The example above is a helix of length k . If there are no other adjacent base pairs in S (that is, if $(i - 1, j + 1) \notin S$ and $(i + k + 1, j - k - 1) \notin S$, in the above example) then it is a **maximal helix**.

RNA Types

There exists a large number of families of RNA that vary enormously in function and length. We will largely disregard the particular role each family plays in the transfer of cellular information, but it is natural to group together sequences of the same family. In ascending order of length, we will consider 5S ribosomal RNA, Group I Introns, Ribonuclease P, 16S ribosomal RNA, and 23S ribosomal RNA. The smallest of these are between 100 and 200 nucleotides, while 23S rRNA sequences can exceed 3000 nucleotides.

We are interested in developing secondary structure inference methods that are compatible with a broad range of RNA types. However, we will also pay special consideration to 16S ribosomal RNA. Relative to other RNA families, 16S rRNA structures are particularly well understood [22]. This translates to larger datasets of 16S sequences with known secondary structures, a clear advantage in a machine learning context.

3.2 Secondary Structure Inference Methods

When homologous sequences (sequences with a shared ancestry) with known secondary structure are available, comparative sequence analysis methods are the gold standard for secondary structure inference [7, 22]. Without these, the preferred method is a thermodynamic energy minimization method referred to as the Nearest Neighbor Thermodynamic Model (NNTM) [55, 48]. This model has been used and studied for decades, and the currently used set of parameters, the Turner parameters, has existed mostly unchanged since 1999 [55].

Given any secondary structure S of an RNA sequence r , NNTM assigns a numerical value to S called its **free energy**, denoted $E(r, S)$, by looking for the presence of various substructures in S . Though the specific values of free energy used in the model have well-founded physical and chemical interpretations, for our purposes it is sufficient to think of negative energy as corresponding to stabilizing substructures (more specifically, stacked pairs), while positive energy corresponds to destabilizing substructures (most other substructures). An RNA sequence r will naturally fold to maximize its stability; thus, under this model, we hope to find the secondary structure with smallest free energy, that is,

$$S^* := \operatorname{argmin}_S E(r, S)$$

where the argmin is over all possible secondary structures. This secondary structure S^* is referred to as the **minimal free energy structure** (MFE structure), and constitutes the model's prediction as to the native secondary structure of r .

Each nucleotide in an RNA sequence can potentially pair with hundreds of other nucleotides, leading to a combinatorial explosion of possible secondary structures as the length of r increases. Computation of the MFE structure is made tractable with dynamic programming methods that exploit the loop structure of RNA secondary structures. NNTM assigns an energy to each loop in S , denoted $e(r, L_{i,j})$. This energy is determined by its type and size, the nucleotides and base pairs enclosed by the loop, and the enclosing loop. Energy is additive, and each loop's free energy is independent of the other loops in the secondary structure:

$$E(r, S) = e(r, L_0) + \sum_{(i,j) \in S} e(r, L_{i,j}).$$

Below, we sketch the recursion that NNTM uses to make the prediction of the MFE structure S^* tractable. For a full exposition of this process, see [55].

Let $r_{i:j} = r_i, \dots, r_j$, and let $\tilde{E}(r_{i:j}) = \min_S E(r_{i:j}, S)$, the minimum energy over all possible secondary structures of $r_{i:j}$. To allow us to use dynamic programming methods here, we note a recursive relationship between $\tilde{E}(r_{i:j})$ and subsequences of $r_{i:j}$. Consider the states of i and j . Any secondary structure S of $r_{i:j}$ will meet exactly one of the following 5 criteria:

1. i and j are both unpaired
2. i is unpaired, and $(k, j) \in S$ for some $i < k < j$
3. j is unpaired, and $(i, k) \in S$ for some $i < k < j$
4. $(i, j) \in S$
5. $(i, k_i) \in S$ and $(k_j, k) \in S$ for some $i < k_i < k_j < k$

The minimum energy of $r_{i:j}$ can be expressed in terms of the minimum energy of a subsequence of $r_{i:j}$ according to which of the five categories the MFE structure of $r_{i:j}$ falls into.

$$\tilde{E}(r_{i:j}) = \min \begin{cases} \tilde{E}(r_{i+1:j-1}) \\ \tilde{E}(r_{i+1:j}) \\ \tilde{E}(r_{i:j-1}) \\ \tilde{E}(r_{i+1:j-1}) + e(r_{i:j}, L_{i,j}) \\ \min_{i < k < j} \tilde{E}(r_{i:k}) + \tilde{E}(r_{k:j}) \end{cases}$$

where, in the first three lines, we add the appropriate elements to the null loop L_0 , and in the fourth line $L_{i,j}$ is understood to contain all of the nucleotides in the null loop of the MFE structure of $r_{i+1:j-1}$.

We can begin by calculating $\tilde{E}(r_{i:i+3})$ for $1 \leq i \leq L - 3$, as by the secondary structure definition these are the smallest subsequences in which base pairs may occur. We then use the recursive relationship above to efficiently calculate $\tilde{E}(r_{i:j})$ for any i, j with $1 \leq i < j \leq L$. If we keep track of all of these minimum energies, we can backtrack from $\tilde{E}(r_{1:L})$ to find the MFE structure S^* .

A common hard constraint on this optimization is a maximum distance d between nucleotides. Motivation for this is twofold. Most directly, it eases computational costs; with this constraint, current implementations are able to produce an MFE structure in $O(Ld^2)$ time with $O(L + d^2)$ memory costs [18]. Further, this assumption leads to more realistic predictions in practice. Fewer than 1% of base pairs in known native secondary structures exceed a commonly used maximum distance of $d = 600$ [14], and it has been observed that structure predictions composed of several smaller, locally optimal structures are more realistic predictions than one densely interconnected globally optimal structure [18].

Various implementations of NNTM exist; in Chapter 4, our experiments will all use GTfold [49]. Other popular variants include RNAstructure [43], UNAFold [37], and ViennaRNA package [35].

The advantages and disadvantages of NNTM are well known. NNTM is highly accurate on small sequences, but has a roughly inverse relationship with sequence length [44]. More troublingly, there has been recent investigation into the fundamental ill-conditioning of NNTM [31, 32, 44]. It is known to be highly sensitive to parameter perturbations, and there are many diverse secondary structures with similar energies.

Measuring Secondary Structure Prediction Accuracy

Let S_{pred} be our prediction of a native secondary structure S_{native} . When evaluating predicted secondary structures, we compare its set of base pairs with those of the native structure. A base pair (i, j) is counted as true positive (TP) if it appears in both the predicted and native structure (if $(i, j) \in S_{\text{pred}} \cap S_{\text{native}}$), a false positive (FP) if it is in the predicted structure but not the native structure (if $(i, j) \in S_{\text{pred}} \setminus S_{\text{native}}$), and a false negative (FN) if it appears in the native structure and not in the predicted structure (if $(i, j) \in S_{\text{native}} \setminus S_{\text{pred}}$). We report on several useful [20] measures of performance for our secondary structure inference methods: positive predictive value, or PPV, the proportion of true positives in the predicted structure ($\frac{\text{TP}}{\text{TP} + \text{FP}}$); as well as sensitivity, the fraction of true positives in the native structure ($\frac{\text{TP}}{\text{TP} + \text{FN}}$). Finally, we define accuracy to be the arithmetic mean of PPV and sensitivity: $\frac{1}{2}(\frac{\text{TP}}{\text{TP} + \text{FP}} + \frac{\text{TP}}{\text{TP} + \text{FN}})$.

3.3 SHAPE and Data-Directed NNTM

NNTM is capable of receive auxiliary information in addition to the RNA sequence to improve the accuracy of its predicted structure. We call this augmented model **data-directed NNTM**.

The most successful incorporation of auxiliary information into NNTM comes from SHAPE, which stands for selective 2'-hydroxyl acylation analyzed by primer

extension [59]. SHAPE is a value associated to each nucleotide of a sequence; this value is a representation of the local flexibility of an RNA sequence, measured by high-throughput chemical probing. SHAPE can take on any nonnegative value, and experiments have shown that a nucleotide’s SHAPE is highly correlated with its state, a relationship that we can use with NNTM to improve MFE accuracy [14].

Several methods of incorporating SHAPE data into NNTM have been explored [36]. We focus on the method presented in [57], which associates a pseudo-free energy to each nucleotide based on its SHAPE. For a nucleotide r_ℓ , this energy $\Delta G_{SHAPE}(\ell)$ is given by

$$\Delta G_{SHAPE}(\ell) = 2.6 \cdot \ln(\text{SHAPE}(\ell) + 1) - 0.8.$$

This pseudo-free energy is added to any stacked pair involving nucleotide ℓ . The result is a soft constraint that discourages NNTM from forming base pairs among nucleotides with high SHAPE. Using SHAPE data to direct NNTM in this manner has been shown to have a large positive effect on MFE accuracy, as in [14], where SHAPE direction increased MFE accuracy on *Escherichia coli* 16S rRNA by more than 30 percentage points.

Exploration into the nature of SHAPE-directed NNTM has shown that improvements over regular NNTM are correlated with undirected MFE accuracy [48]; that is, low-accuracy NNTM predictions are not significantly improved through SHAPE direction. Beyond this, the efficacy of SHAPE-directed NNTM is limited by the availability of SHAPE data. SHAPE is based on real-world chemical experiments that are time-consuming to carry out; as noted in [60], SHAPE data collection for a 100-200 nucleotide sequence takes up to two days. Thus, SHAPE-directed NNTM is a promising but currently mostly inaccessible improvement in RNA secondary structure inference.

Chapter 4 Improving NNTM via State Inference with Neural Networks

This chapter presents our first venture into using deep learning to attack the problem of RNA structure inference. Instead of using neural networks to directly generate a predicted RNA structure, we present a method for structure inference based on a novel deep learning method that supplies auxiliary information to direct NNTM. To do so, we present a neural network architecture for the problem of predicting the state of each nucleotide in an RNA sequence; this is a binary classification on each element of a sequence, and is thus compatible with convolutional and recurrent network layers. We train both our method and a benchmark hidden Markov model (HMM) on sets of 16S rRNA sequences. We see that our machine outperforms the benchmark by an average of 15 percentage points on a test set, and explore how well each method is able to capture global RNA state patterns, like length and number of paired regions.

We then examine correlations between nucleotide state and real SHAPE data to derive a novel method for converting state predictions into synthetic SHAPE data. With this synthetic SHAPE in hand, we then use it to direct NNTM and provide a predicted structure to the original RNA sequence. The entire pipeline, which we call predicted state directed NNTM, is a secondary structure method that is applicable to any RNA sequence and requires no information beyond the sequence itself. On a test set of 16S rRNA, predicted state directed NNTM improves average MFE accuracy from 40.8% to 66.4%, an increase of over 25 percentage points.

4.1 SHAPE-Directed NNTM Without SHAPE

As noted in Chapter 3, SHAPE direction is capable of dramatically increasing the accuracy of NNTM predictions. A severe limitation to SHAPE-directed NNTM is the availability of SHAPE data, which is costly to determine.

The correlation between nucleotide state and SHAPE was investigated in [48]. There, the authors present distributions for the value of a nucleotide’s SHAPE data based on its state, and use these distributions to stochastically generate synthetic SHAPE data. Directing NNTM with this synthetic SHAPE is shown to dramatically improve its accuracy, as exhibited by its application to a test set of 16S rRNA sequences.

The investigation in [48] includes many interesting findings about interactions among SHAPE data, nucleotide state, and secondary structure, but the methods presented are not a viable method for secondary structure inference: the method uses information from the known secondary structure to generate the SHAPE that influences NNTM predictions. However, their work shows how knowledge about the state of each nucleotide in the sequence can be translated into improvements in secondary structure inference accuracy.

This chapter presents a method for improving the RNA secondary structure prediction partially inspired by this work. We first present a deep learning method for

determining the state of each nucleotide of an RNA sequence, which we refer to as state inference. State inference is a binary classification task on each nucleotide, so we can easily set up an RNN for the problem of state inference, where the input is the sequence of nucleotides and the output is a binary sequence of the same length. We develop, train, and test a deep recurrent neural network that performs this task. Given an RNA sequence, the machine outputs a probability that each nucleotide is paired. We can threshold this probability to obtain binary predictions for the state of each nucleotide.

To connect this to structure inference, we present our own method for converting our neural network’s probabilistic state predictions into synthetic SHAPE, and we use this SHAPE data to direct secondary structure predictions via NNTM. This direction leads to significant improvements over secondary structure accuracy on sequences where our state inference method performs well. The full pipeline is a novel method applicable to any 16S rRNA sequence, and combines machine learning with existing thermodynamic structure inference methods to improve accuracy.

Although we are primarily interested in using state inference to direct secondary structure predictions, we note that there exist other motivations for state inference. For example, such a method could be used to identify binding sites in RNA-RNA interactions [16, 51].

4.2 Methods

Directing NNTM With State Information

Our proposed method for secondary structure inference uses a viable method for state inference with the ability to influence the NNTM energy function on a per-nucleotide basis via SHAPE direction. Our method is a three-step process:

1. A machine learning method for predicting the state of each nucleotide in a sequence;
2. A function converting these state predictions into artificial SHAPE data;
3. The SHAPE-directed NNTM function that takes both the original RNA sequence and the generated SHAPE data and outputs a predicted secondary structure.

For the task of state inference, we trained a deep neural network using a set of known RNA sequences and structures that generates a sequence of state predictions, detailed later in this section. Given an RNA sequence x of length L , the output of this neural network is a sequence p of length L , where $p^{(\ell)} \in \mathbb{R}$ is the predicted probability that the nucleotide in position ℓ is paired.

With these predictions in hand, we convert each predicted probability $p^{(\ell)}$ to a SHAPE value to be associated with nucleotide ℓ . To construct a function for this purpose, we note that a SHAPE value of ≈ 0.3603 will not contribute any positive or negative energy to the NNTM energy function; this can be seen by setting

$\Delta G_{\text{SHAPE}}(\ell)$ to 0 in the pseudo-free energy equation from Section 3.3 and solving for $\text{SHAPE}(\ell)$. We would therefore like to assign predictions of 0.5 to a SHAPE value of 0.3603, as these predictions give no information as to the state of the nucleotide. With this in mind, we use the following piecewise linear function to generate SHAPE, where a and b are constants to be specified.

$$f(\ell) = \begin{cases} 2(0.3603 - b)p(\ell) + b, & \text{if } 0 < p(\ell) \leq 0.5 \\ 2(a - 0.3603)(p(\ell) - 1) + a, & \text{if } 0.5 < p(\ell) \leq 1 \end{cases}$$

This function has range $[a, b]$, with $f(\ell) = a$ if $p(\ell) = 1$, $f(\ell) = b$ if $p(\ell) = 0$, and $f(\ell) = 0.3603$ if $p(\ell) = 0.5$. To determine sensible values of a and b , we considered experimentally collected SHAPE data from two *E. coli* sequences, one 16S sequence and one 23S sequence [48]. Together, these two sequences contain a total of 4187 nucleotides, and represent a wide variety of structural motifs. We took the mean SHAPE value among both paired nucleotides and unpaired nucleotides; these values are 0.214 and 0.6624, respectively.

Our main experimental results will use $a = 0.214$ and $b = 0.6624$ in our SHAPE generation function. These choices are motivated by real SHAPE values, and thus are sensible estimations of the best values. However, they may not be the optimal values for our purposes. Later, we will explore how varying these values may affect the accuracy of our predicted secondary structures. These experiments indicate that the a and b values used in our experimental results are nearly optimal.

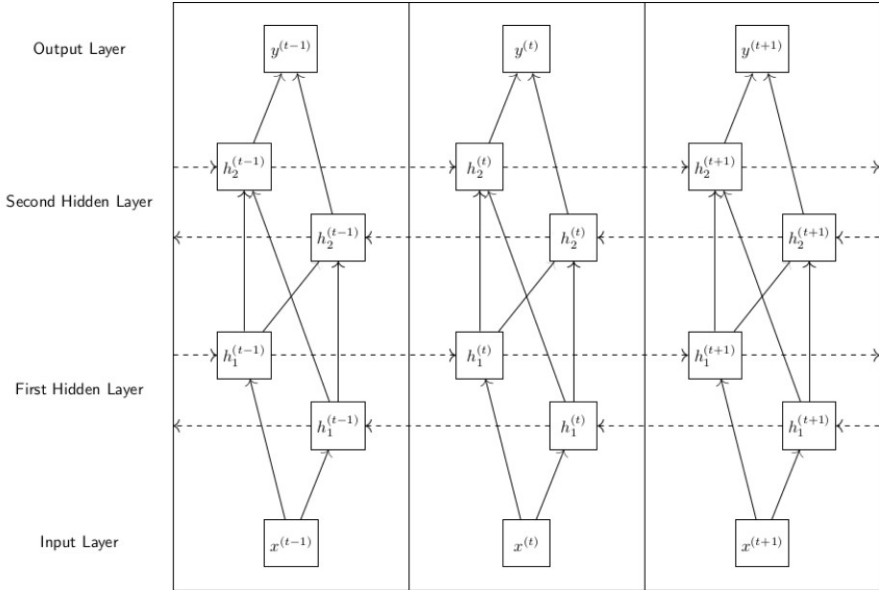
With state predictions and set values of a and b , we can generate a sequence of artificial SHAPE data. We then use SHAPE-directed NNTM as described in the previous section to obtain our secondary structure prediction.

Neural Network for State Inference

For the task of state inference, we use a four layer neural network with both convolutional and recurrent layers. The input is a sequence x of length L , with $x^{(\ell)} \in \mathbb{R}^5$ a one-hot encoding of the 5 possible nucleotides A, C, G, U, or X. The first layer is a 1D convolution with stride 1, kernel size 30, ReLU activation, and hidden size 100. This is meant to act as a learnable preprocessing filter; the kernel size is large enough to include, for example, all of the nucleotides in an average-sized hairpin loop, and this layer could learn something akin to the free energy terms used in NNTM.

This first layer takes in and processes local information, while longer-term dependencies are handled by the next two layers, both of which are LSTM layers and form the foundation of the network. Because a nucleotide’s state is liable to have dependencies on nucleotides before and after it in the sequence, we make these layers bidirectional; the second layer has hidden size 400 (200 each for the forward and backward direction LSTMs), and the third layer has hidden size 100 (50 each for forward and backward LSTMs). A diagram of a two-layer bidirectional RNN is shown in Figure 4.1. The final layer is another 1D convolution, fulfilling a similar role as the first layer; its stride is 1 and kernel size is also 30.

Figure 4.1: Diagram of a two-layer deep bidirectional RNN at three different timesteps $t - 1$, t , and $t + 1$. Solid arrows represent the propagation of information through layers at a particular timestep, while dashed arrows represent information propagating across timesteps.



In all, the machine has a total of 595,552 trainable parameters, which we trained using the RMSprop [54] algorithm with a learning rate of $\eta = 10^{-4}$. Our network was implemented in Keras [12], a Python deep learning API, with Theano [53] as a backend. We also use a regularization term and dropout with drop probability 0.5; see code available at <https://github.com/dwillmott/rna-state-inf> for other hyperparameters and training details.

Dataset, Implementation, and Metrics

Our experiments will focus on a test set of sixteen 16S ribosomal RNA sequences used in SHAPE direction experiments in [48]. Sequences in this set have a wide range of NNTM accuracies, and using this test set allows us to directly compare our method of SHAPE generation with that presented in [48].

As a training set for this task, we used secondary structure data from the Comparative RNA Web site, run by the Gutell Lab at the University of Texas [6]. This site hosts a collection of known RNA sequences and secondary structures obtained using comparative sequence analysis. Compiling all of the available 16S rRNA results in a set of 17032 sequences and a total of over 21 million nucleotides. We refer to this as the CRW dataset.

To ensure that our model does not simply memorize large portions of sequences in the test set, we compared the each CRW dataset sequence with each test set sequence and removed CRW sequences with significant similarities prior to training.

In this filtering process, if the two sequences have a common block of nucleotides of more than 10% of the length of the test sequence, or if the two sequences can be aligned such that they have common nucleotides accounting for more than 80% of nucleotides of the shorter sequence, we remove it from the training set. See available code for additional details. This process leaves us with 13118 sequences and a total of approximately 16.5 million nucleotides, with a mean and median sequence length of 1264 and 1431, respectively. We then split this set into two random halves to produce a training and validation set.

Metrics for the state inference task are shown in Section 4.3. After state inference is performed, and we have the sequence of state prediction probabilities p , we generated artificial SHAPE data using the method described in Section 4.2. Finally, we used both the original sequence and the generated SHAPE data as input for SHAPE-directed NNTM. Section 4.4 lists our experiments and results involving NNTM, which all use GTfold [49], an efficient NNTM implementation.

In this chapter, we are considering methods for both state inference, which binary classifies individual nucleotides, and secondary structure inference, which classifies base pairs. This leads to two differing definitions of accuracy. The usual definition classification accuracy (the proportion of true predictions among all predictions in the sequence) will be used when considering our neural network for state inference in Section 4.3. The secondary structure prediction accuracy definition presented in Chapter 3 as the mean of PPV and sensitivity will be used when considering the directed NNTM predictions in Section 4.4.

4.3 State Inference

State Inference Accuracy

The foundation of our method is our deep neural network for state inference: this network provides probabilities that are converted into a pseudo-free energy term in the NNTM energy function. To understand the sources of high and low performance of our structure inference method, we can directly evaluate the output of our deep neural network for state inference.

Table 4.1: State inference PPV, sensitivity, and accuracy of our neural network vs. HMM on validation and test sets

Machine	Validation Set			Test Set		
	Acc	PPV	Sen	Acc	PPV	Sen
Order 1 HMM	0.623	0.632	0.852	0.612	0.646	0.767
Order 2 HMM	0.662	0.671	0.826	0.651	0.686	0.759
Order 3 HMM	0.674	0.693	0.794	0.672	0.713	0.750
Order 4 HMM	0.685	0.714	0.771	0.684	0.729	0.742
Order 5 HMM	0.684	0.711	0.776	0.683	0.730	0.742
Neural Network	0.954	0.950	0.972	0.839	0.858	0.873

Table 4.2: State inference results on the test set from our neural network vs. an order 4 HMM. Sequences are arranged in ascending order of MFE accuracy as an indication of the difficulty of secondary structure inference for each sequence. Average indicates the average metric for each sequence, while Total gives the total metrics for all nucleotides in the test set.

Sequence Name	Acc		PPV		Sen	
	LSTM	HMM	LSTM	HMM	LSTM	HMM
E. cuniculi	0.680	0.661	0.713	0.693	0.774	0.773
Vairimorpha necatrix	0.661	0.600	0.721	0.689	0.683	0.576
C. elegans	0.558	0.584	0.570	0.613	0.624	0.552
Emericella nidulans	0.657	0.584	0.692	0.681	0.741	0.539
Nicotiana tabacum	0.913	0.705	0.917	0.734	0.938	0.787
Cryptomonas.sp	0.926	0.676	0.935	0.730	0.941	0.728
Synechococcus.sp	0.938	0.700	0.943	0.740	0.953	0.769
M. musculus	0.608	0.603	0.626	0.655	0.637	0.520
Mycoplasma gallisepticum	0.919	0.639	0.933	0.713	0.932	0.668
E. coli	0.924	0.699	0.937	0.742	0.938	0.774
Bacillus subtilis	0.973	0.698	0.979	0.731	0.976	0.788
Desulfovibrio desulfuricans	0.926	0.712	0.940	0.741	0.938	0.803
Chlamydomonas reinhardtii	0.906	0.687	0.915	0.725	0.928	0.761
Thermotoga maritima	0.931	0.752	0.944	0.760	0.943	0.864
Thermoproteus tenax	0.818	0.782	0.845	0.785	0.866	0.894
H. volcanii	0.782	0.739	0.809	0.769	0.841	0.820
Average	0.820	0.676	0.839	0.719	0.853	0.726
Total	0.839	0.684	0.858	0.729	0.873	0.742

For the sake of comparison, we trained and tested a number of higher-order hidden Markov models (HMM) using the same training, validation, and test sets used by the neural network. Training was done using maximum likelihood estimation, and state inference was performed with the Viterbi algorithm [17], a backtracking algorithm that produces the likeliest state sequence under the model probabilities. HMMs are fundamentally incapable of recognizing dependencies across many timesteps, and we therefore expect the deep neural network to outperform the HMM. However, they provide a baseline against which to measure neural network output.

To calculate the accuracy of the neural network’s output, we thresholded each prediction $p^{(i)}$ above and below 0.5, taking $p^{(i)} > 0.5$ to be a positive prediction and $p^{(i)} < 0.5$ to be a negative prediction. The accuracy, PPV, and sensitivity of both neural network and HMM predictions are shown in Table 4.1. Though the table exhibits an upward trend in accuracy as the order of the HMM increases, we found that accuracy plateaued and eventually decreased beyond order 5. As expected, the neural network clearly outperforms HMMs of all orders on the validation set. More importantly, this is the case for our test set as well, where it outperforms the best HMM in accuracy by 15%.

The order 4 HMM exhibits the highest accuracy on the validation set. We further

compared state inference accuracy on each test set sequence using both the order 4 HMM and the neural network. The accuracy, PPV, and sensitivity of these predictions are shown in Table 4.2. The accuracy of neural network state predictions were, on average, 15 percentage points higher than that of the HMM, and was higher for every sequence but one (*C. elegans*). Table 4.2 orders sequences in ascending order of undirected MFE accuracy; however, this ordering reveals no straightforward relationships among neural network state inference accuracy, HMM state inference accuracy, and MFE structure accuracy. Neural network accuracy varies much more among sequences: the difference between the sequences with lowest and highest accuracy (*C. elegans* and *B. subtilis*, respectively) is more than 40 percentage points. Sequences can be grouped according to accuracy: poor (below 70%) for five sequences, medium (near 80%) for two more, and high (above 90%) for the remaining nine.

Paired Regions & Global Structure

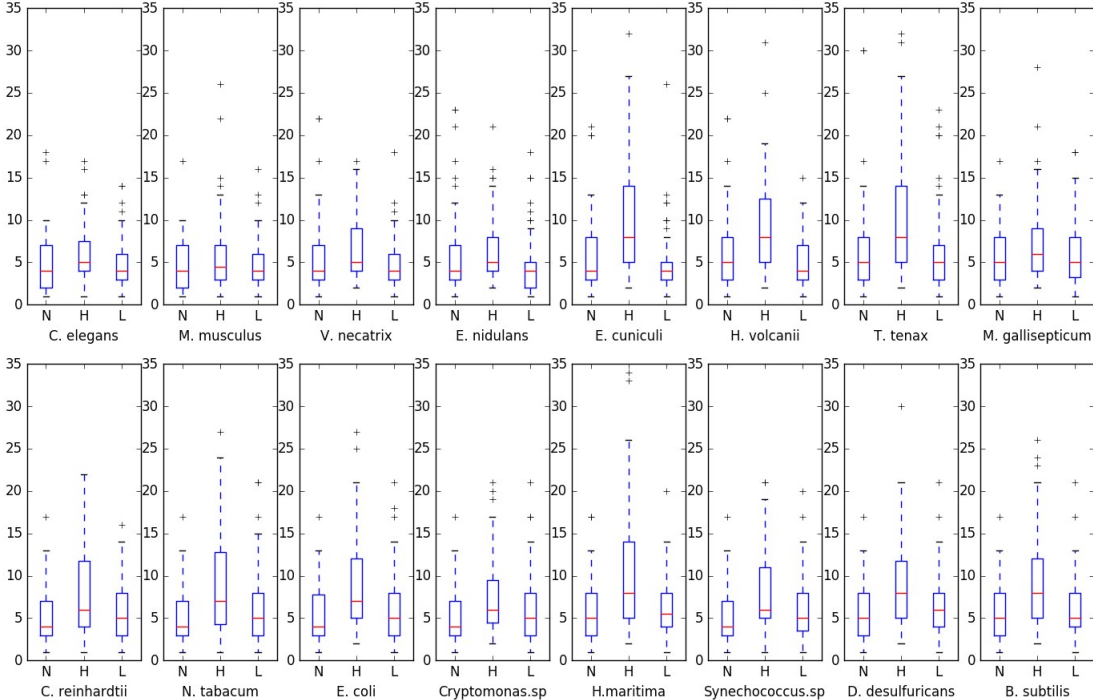
Our metrics in Table 4.2 give us an idea of the proportion of correct machine predictions on individual nucleotides’ states, but they do not indicate whether predictions produce state sequences that preserve global properties, such as patterns of paired and unpaired states. In particular, we want the number and sizes of paired and unpaired regions of the state sequence prediction to match those in the original. A paired region in the state roughly corresponds to one half of a helix in the secondary structure, so we theorize that recognizing this information is vital for producing state predictions that successfully aid structure inference.

We considered the distribution of sizes of paired regions in each test set state sequence, and compared them to the distributions of neural network and HMM state predictions; Figure 4.2 shows boxplot of these distributions for native states and compares them to HMM and neural network state predictions on each of our 16 test sequences. Despite larger variance in state inference accuracy, we found evidence that the neural network was, on average, much more capable than the HMM of capturing this global structure. The median size of paired region in neural network predicted state differed from the median in the native state by at most one for every test set sequence, while the HMM’s median paired region size was routinely several nucleotides larger. We can also consider the total number of paired regions in the state as another global feature of RNA state. The neural network performs better in this regard as well, producing predictions that, on average, had 6 more paired regions than the native state. On the other hand, HMM predictions had an average of 57 fewer regions than the native state, vastly underestimating the correct number of paired regions.

We note that this discrepancy is to be expected in the context of nonlocal interactions. Paired region size is exactly the sort of nonlocal feature that HMMs cannot predict: at a given time, the HMM does not know how long it has been outputting positive predictions, and is thus limited in its capacity to detect large paired regions.

Considering the non-locality of paired regions can help to explain the poor performance of the neural network on certain test set sequences. High neural network accuracy is nearly always accompanied by a particular type of distribution of large

Figure 4.2: Boxplots of the distribution of sizes of paired regions in the native state sequence, HMM predicted state sequence, and LSTM predicted state sequence (denoted N, H, and L, respectively) for each test set sequence. The red line indicates the median region size, the box contains 25th-75th percentiles, and the whiskers contain 5th-95th percentiles. Sequences are ordered from lowest to highest LSTM prediction accuracy. Several large paired regions in HMM predictions beyond the y-axis limit of 35 are not shown.

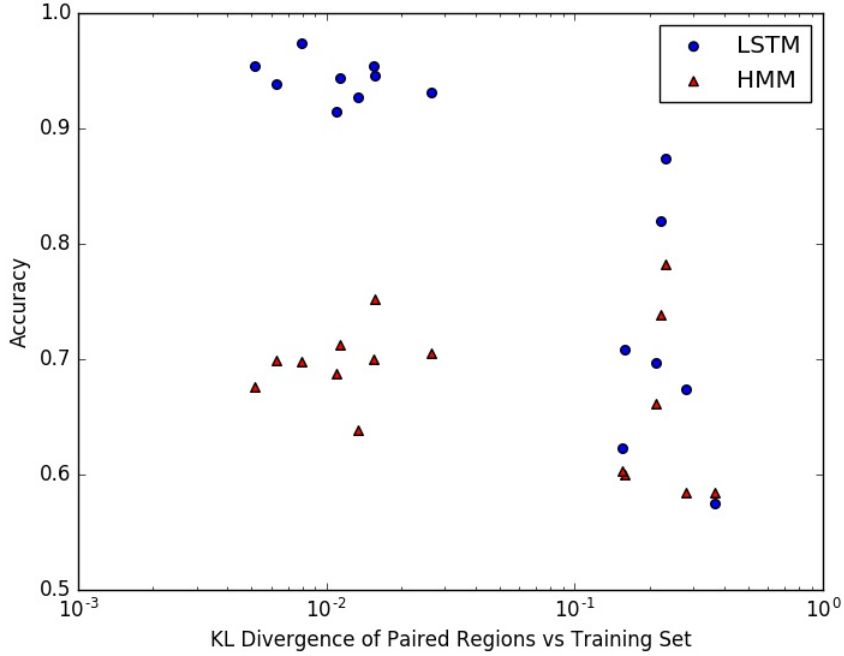


paired regions: one of length 17, one of length 13, and several more of length 12 and 11. In contrast, this pattern does not hold for those with low or medium state inference accuracy: of the remaining seven, all have either paired regions of length larger than 20 (*E. cuniculi*, *V. necatrix*, *M. nidulans*, *T. tenax*, and *H. volcanii*) or very few paired regions of length larger than 10 (*C. elegans* and *M. musculus*).

We can compare the distribution of the lengths of paired regions in each of our test sequences to the distribution in the training set. We find that the training set overwhelmingly contains sequences with paired region distributions similar to the test set sequences on which the neural network performs well. In particular, we note that the training set has relatively few large paired regions: in the entire training set, there are 5 regions of length 18, 2 regions of size 19, 4 regions of size 20, and none larger than 20. Thus, during training the machine is penalized for outputting more than 20 contiguous positive predictions. Consequently, neural network predictions do not create sufficiently large regions for many test set sequences.

To quantify this difference, we considered the Kullback-Leibler (KL) divergence of

Figure 4.3: Plot comparing each test set sequence’s LSTM (neural network) and HMM state inference accuracy vs its Kullback-Leibler divergence from training set paired region distribution. KL divergence was calculated as $KL(P||Q)$, where P is the test sequence distribution and Q is the training set distribution.



the distribution of the paired region lengths between the entire training set and the distribution for each test set sequence. The KL divergence measures the similarity of each test set sequence’s paired region distribution as compared with the distribution of the entire training set. Figure 4.3 plots state inference accuracy for each machine and test set sequence against its KL divergence.

Two clusters of sequences emerge in this plot: one with KL divergence near 0.01, and another with KL divergence near 0.5. All nine sequences with high state inference accuracy are in the former cluster, while the seven low and medium accuracy sequences are in the latter. The disparity in neural network accuracy and HMM accuracy on sequences with more similarity to the training set suggests that the increase in neural network performance comes from its ability to recognize global structure in these sequences. On the other hand, neural network accuracy is only a modest improvement from HMM accuracy in the low to medium accuracy cluster, where global structure diverges significantly from that of the training set.

Table 4.3: Table of accuracy of MFE structures using NNTM with a variety of SHAPE directions. First column: undirected MFE. Second column: predicted state directed MFE. Third column: mean performance of sampled SHAPE directed NNTM in [48]. Fourth column: native state directed NNTM.

Sequence Name	Undirected MFE	Directed MFE		
		Predicted	Sükösd	Native
<i>E. cuniculi</i>	0.171	0.183	0.273	0.336
<i>V. necatrix</i>	0.181	0.314	0.503	0.705
<i>C. elegans</i>	0.203	0.248	0.308	0.519
<i>E. nidulans</i>	0.272	0.325	0.601	0.832
<i>N. tabacum</i>	0.323	0.692	0.593	0.859
<i>Cryptomonas.sp</i>	0.339	0.838	0.739	0.898
<i>Synechococcus.sp</i>	0.361	0.848	0.697	0.885
<i>M. musculus</i>	0.375	0.397	0.509	0.782
<i>M. gallisepticum</i>	0.385	0.849	0.721	0.889
<i>E. coli</i>	0.411	0.852	0.744	0.880
<i>Bacillus subtilis</i>	0.512	0.848	0.753	0.881
<i>D. desulfuricans</i>	0.533	0.875	0.724	0.898
<i>C. reinhardtii</i>	0.537	0.845	0.702	0.868
<i>T. maritima</i>	0.562	0.881	0.733	0.896
<i>T. tenax</i>	0.619	0.766	0.754	0.861
<i>H. volcanii</i>	0.752	0.864	0.809	0.907
Mean	0.408	0.664	0.635	0.806
Median	0.380	0.841	0.712	0.874

4.4 Directing NNTM with Synthetic SHAPE

Native State Directed NNTM

Before analyzing the results of the entire pipeline of our method, we first examined our SHAPE generation function in detail. To do so, we used the native state of each sequence in our test set to generate SHAPE. This was done by setting $p^{(i)}$ to 1 if the nucleotide in position i is paired, and $p^{(i)}$ to 0 if it is unpaired. We then use the equation in Section 4.1 to generate artificial SHAPE. This will result in a generated SHAPE value of 0.6624 for all paired nucleotides and 0.214 for unpaired nucleotides, which we then use to direct NNTM. We refer to the resulting predicted structures as native state directed MFE.

This experiment is similar to those run in [48], and uses the same set of data to choose appropriate SHAPE values. The difference is in the method of SHAPE generation: whereas that paper constructs SHAPE distributions from the data and stochastically samples from these distributions, we use the mean of paired and unpaired nucleotides' SHAPE values.

The results of this experiment reinforce many of the findings in [48]. A compari-

son of accuracy of all three methods (undirected MFE, stochastically directed MFE from [48], and native state directed MFE) is available in Table 4.3. Overall, native state directed MFE structures are highly accurate, with twelve of the sixteen test sequences enjoying accuracy above 80%. Both direction methods are an improvement on the accuracy of the undirected MFE structure for every test set sequence, and native state directed accuracy represents a further improvement from the stochastic model in [48]. In the case of native state direction, accuracy improvements over undirected MFE range between 15 percentage points (*H. volcanii*) and 57 percentage points (*Cryptonomas.sp.*). Consistent with observations in [48], greatest increases are concentrated in sequences with middling undirected MFE accuracy; for sequences with undirected accuracy between 25% and 45%, native state directed MFE accuracy is an improvement by more than 40 percentage points.

This experiment is equivalent to assuming that our deep learning state inference method has perfect performance, and as such we can interpret the accuracy of native state directed MFE structures to be an upper bound on the performance of our method. On average, the high accuracy exhibited in this experiment gives strong evidence that there are large potential gains in MFE accuracy to be made with our method. However, several sequences with low undirected MFE accuracy sequences like *E. cuniculi* and *C. elegans* are known to be particularly resistant to SHAPE direction [48], and this is reflected in relatively poor native state directed MFE accuracy. We thus cannot expect our method to exhibit large improvements over undirected MFE structures in these cases.

Predicted State Directed NNTM

We now use the predictions from our deep neural network to generate SHAPE that will in turn direct NNTM; we refer to these predictions as predicted state directed MFE structures. We emphasize that, unlike the native state direction explored in the previous section, this method does not assume prior knowledge of the state of the sequence, and thus represents a practical method of secondary structure inference.

The results of applying our method to the sequences in the test set are available in Table 4.3, which indicates that the extraordinary gains from native state directed NNTM are not always preserved in practice. Predicted state directed structures fall into two clear categories: five are quite inaccurate, with accuracy below 40%, while among remaining eleven structures are all near or above 70%, and nine of these are above 80%. Even with the high variance of accuracies among these structures, predicted state directed MFE structures are 25 percentage points more accurate than undirected MFE structures on average, and every sequence in the test set experiences some increase in accuracy. However, these improvements vary greatly, with several sequence staying within 5% of undirected MFE accuracy, while for four other sequences we improve by more than 40%, with the highest improvement (*Cryptonomas.sp.*) at 50%.

To some extent, poor accuracy is explained by our experiment with native state directed NNTM. Indeed, the five sequences with poor accuracy from our method are the five worst-performing with native state direction, and only one of these exceeds

80% with native states. At worst, native state directed NNTM gives only 34% accuracy for *E. cuniculi* and 52.6% for *C. elegans*, and this ceiling is much lower than accuracy achieved in many of our other predicted structures. However, in these cases and some others (*E. nidulans*, *M. musculus*), predicted state directed MFE accuracy does not come close to native state directed MFE accuracy. This is in contrast to our highest performing sequences (*D. desulfuricans*, *T. maritima*), where predicted state directed MFE is within several percentage points of native state directed MFE accuracy.

State inference accuracy unsurprisingly exhibits a strong effect on predicted state directed MFE accuracy. State inference accuracy above 90% means that our predicted states are quite close to native states; consequently, predicted state direction and native state direction produce similar predicted structures in these cases, as evidenced by their difference of only a few percentage points in Table 4.3. Meanwhile, the five sequences with poor state inference accuracy are exactly those where predicted state directed MFE accuracy is below 40%.

The effect of state inference accuracy is particularly evident when considering the improvement over undirected MFE accuracy: for four of the five sequences with poor state inference accuracy (all but *V. necatrix*), predicted state directed MFE accuracy is within 6 percentage points of undirected MFE accuracy. For *V. necatrix* and both sequences with medium state inference accuracy, predicted state direction improves structure accuracy by 10-15 percentage points. The remaining nine sequences all have high state inference accuracy, and their directed structures are 30 percentage points more accurate than undirected MFE.

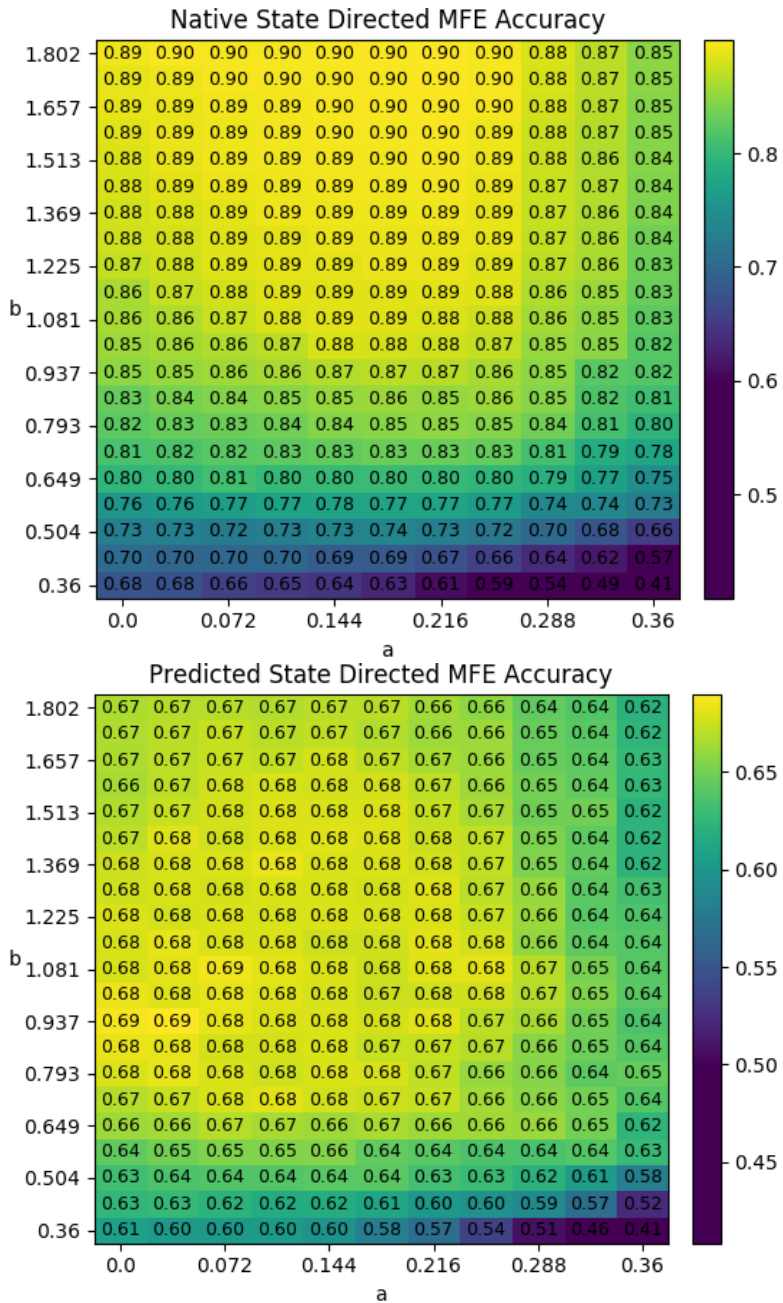
We note an interesting relationship between native state directed MFE accuracy and our neural network’s state inference accuracy. The five sequences with state inference accuracy below 70% are the five worst performing sequence when predicting structure with native state directed NNTM. This suggests that there may be fundamental difficulties in understanding pairing structures of these sequences.

Modifying Synthetic SHAPE Values

Our method uses SHAPE-directed NNTM as a means of assigning pseudo-free energies to individual nucleotides. All of our results in Section 3 assign nucleotides a SHAPE value in the range [0.214, 0.6624]. These endpoints are based on the mean SHAPE value of paired and unpaired nucleotides from 16S and 23S *E. coli* sequences. However, our method converts to SHAPE primarily as a means of assigning pseudo-free energies to individual nucleotides with NNTM, and not as a genuine attempt to generate plausible SHAPE data. Thus, the endpoints used may not be optimal for our purposes of converting from state inference predictions.

To evaluate potential output ranges for our SHAPE generation function, we reproduced experiments with native state directed NNTM (Section 3.1) and predicted state directed NNTM (Section 3.2) while varying the endpoints a and b of our SHAPE generation function. As noted previously, a SHAPE value of 0.3603 contributes no energy to the model; thus, it is only sensible to choose paired SHAPE values below 0.3603, and unpaired SHAPE values above 0.3603. NNTM software such as GTFold

Figure 4.4: Accuracy of native state directed MFE (top plot) and predicted state directed MFE (bottom plot) for various ranges $[a, b]$ of output from our SHAPE generation function. In each, the lower right corner corresponds to $a = b = 0.3603$, which is equivalent to no SHAPE direction.



ignores negative SHAPE values, so paired nucleotides' generated SHAPE must lie between 0 and 0.3603. The results of this experiment are shown in Figure 4.4.

In native state directed NNTM, increasing negative state SHAPE above 0.3603 and decreasing positive state SHAPE below 0.3603 consistently increased perfor-

mance. This is consistent with our expectations, as in this case we are increasing the energy of all base pairs involving nucleotides that remain unpaired in the native structure. Experiments with very large unpaired SHAPE values, such as $b = 20$, were similar to the largest values shown in Figure 4.4, indicating that there is a ceiling of approximately 90% test set accuracy for any method centered around SHAPE-directed NNTM such as ours.

The plot for predicted state directed NNTM shows a different picture, with increasing unpaired SHAPE values eventually leading to decreasing structure inference accuracy. That this pattern appears in the predicted state experiments but not native state experiments suggests that incorrectly assigning large SHAPE values to even a small number of natively paired nucleotides can be significantly harmful to NNTM performance. There is a large region of highest accuracy, with a between 0 and 0.22 and b between 0.7 and 1.5 giving accuracies near 68%. The values of $a = 0.214$ and $b = 0.6624$ used in our results are near the boundary of this region. But we note that even optimal values of a and b give an accuracy of 69%, only 2.5 percentage points above the experimentally motivated choices of a and b used in our results.

Chapter 5 Direct Secondary Structure Inference with Neural Networks

In Chapter 4, we presented a neural network that performed state inference on a sequence, and put forward a way to convert these state predictions into auxiliary data to improve NNTM secondary structure predictions. In contrast, in this chapter we present a novel architecture of neural networks that uses a two-dimensional representation to output a prediction for each possible pair to appear in the secondary structure. To use these predictions to form a secondary structure prediction, we present an algorithm to iteratively construct a likeliest secondary structure given these probabilities. This algorithm has several foundational benefits over NNTM: we may construct a secondary structure that includes or excludes pseudoknots as we wish, and we are given a notion of confidence in each base pair prediction.

Our main experimental result is based on training and test sets of 16S rRNA; using our method, we achieve an extraordinarily high average structure inference accuracy of 84.9% on our test set, an increase of nearly fifty percentage points over undirected NNTM on the same set. We also find satisfactory performance when expanding our training and test sets to include other RNA types beyond 16S rRNA; in a test set of 49 sequences that spans 5S and 16S rRNA, Ribonuclease P, and Group I introns, our method produced a more accurate structure than NNTM for all but 6 of these sequences.

5.1 Representing Secondary Structures

The first challenge in developing a deep learning method for secondary structure inference is determining how to represent secondary structures in a manner compatible with neural networks. For this purpose, we will require a representation that can be translated into an array of real numbers, or a sequence of such arrays, to use as the labels to accompany input to the network. Because we will be receiving predictions from the neural network via this representation, we will also be interested in investigating its stability, that is, how small errors in the array affect the secondary structure it represents.

In addition to the arc diagram in Figure 3.1, there exist numerous ways to represent secondary structures, many of which can be used to elicit combinatorial information about possible secondary structures [26]. We discuss several possibilities below. In the following, let r be a sequence of length L and let S be a secondary structure of r .

We can use partial parenthesizations to represent S . This manifests as a sequence of length L on three symbols: \cdot , $($, and $)$. Here \cdot is an unpaired nucleotide, $($ is a nucleotide paired with another nucleotide later in the sequence, and $)$ is a nucleotide paired with another nucleotide earlier in the sequence. The nesting property ensures that a base pair is represented by a pair of parentheses. A one-hot encoding of this sequence can act as a length L sequence of labels, with each element in \mathbb{R}^3 .

However, we note that this representation is relatively unstable, in the sense that very few incorrect predictions can radically change the secondary structure prediction as a whole. This is exhibited in the three possible outputs and their associated secondary structures shown below, with deviations from the first sequence and structure shown in red. A small number of errors can create a significantly different secondary structure, or even worse, can fail to be a partial parenthesization at all, making it unclear how to interpret the output.

$$\begin{array}{ll}
 (\cdot (\cdot) \cdot ((\cdot \cdot))) & S = \{(1, 13), (3, 5), (7, 12), (8, 11)\} \\
 (\cdot (\cdot) \cdot) (\cdot \cdot) () & S = \{(1, 7), (3, 5), (8, 11), (12, 13)\} \\
 (\cdot (\cdot) \cdot (() \cdot)) & S = \{(1, 12), (3, 5), (6, 11), (7, 8), (? , 13)\}
 \end{array}$$

Another possible representation is the two-dimension representation using an array $A \in \mathbb{R}^{L \times L}$. We let $A^{(i,j)} = 1$ if $(i, j) \in S$ and $i < j$, and 0 otherwise. We disregard the diagonal and lower triangle of A .

Like the partial parenthesizations representation above, a drawback of this 2D array representation is that we cannot necessarily interpret an arbitrary upper-triangular binary array as a secondary structure, as our prediction may include a number of pairs that cannot both exist in the secondary structure (e.g., a matrix may contain two 1s in the same row; this represents a single nucleotide being paired with two different nucleotides, which is not permissible in a secondary structure). However, unlike the the partial parenthesizations representation, individual entries of the array do not interact to form base pair predictions, and thus a small error in one entry will not invalidate a correct prediction in another.

We will use this representation as an output to the neural network; rather than a binary matrix, entries of the network output \hat{Y} will be bounded between 0 and 1, with $\hat{Y}^{(i,j)} \in [0, 1]$ representing the predicted probability that (i, j) is paired in the secondary structure. We present an algorithm in Section 5.2 that uses these probabilities to convert the set of probabilistic base pair predictions into a coherent secondary structure prediction.

5.2 Neural Network Architecture

Using the 2D array from the previous section to represent the output, we now have a problem with a 1D sequential input and a 2D sequential output. To move from a 1D to 2D representation, we perform a Cartesian product of the sequence with itself, as described below. If $x = (x^{(1)}, \dots, x^{(L)})$ with $x^{(\ell)} \in \mathbb{R}^n$, then the output of the Cartesian product with itself is an array $y \in \mathbb{R}^{L \times L \times 2n}$, where

$$y_i^{(k,\ell)} = \begin{cases} x_i^{(k)} & \text{if } i \leq n \\ x_{i-n}^{(\ell)} & \text{if } i > n \end{cases}$$

Thus, $y^{(k,\ell)}$ consists of sequences elements $x^{(k)}$ and $x^{(\ell)}$ stacked together.

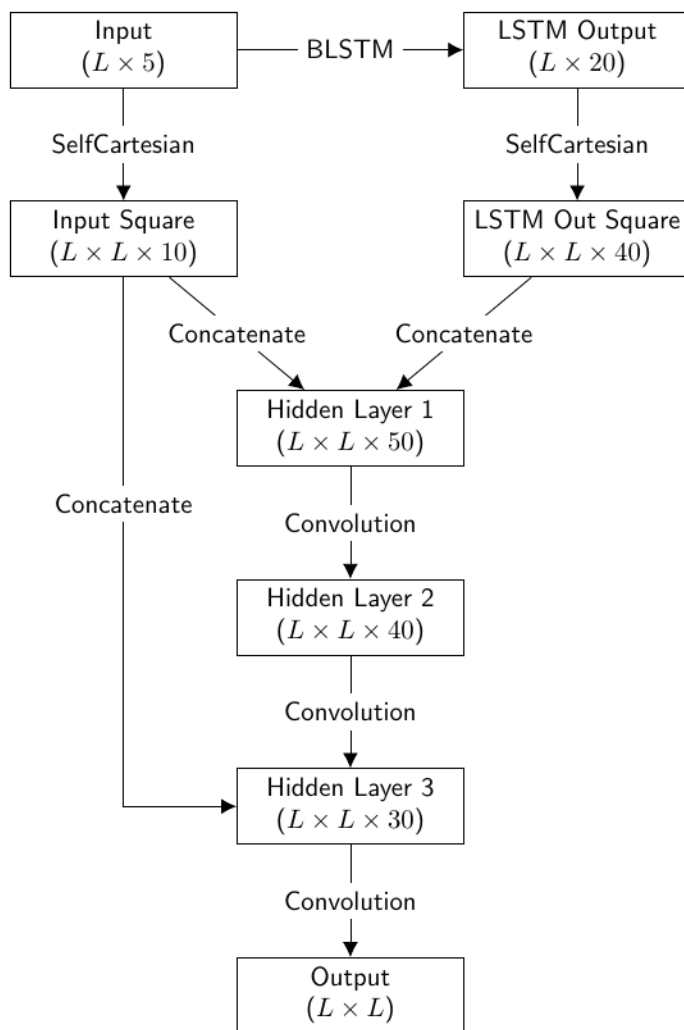


Figure 5.1: Diagram of neural network architecture used for secondary structure inference in Chapter 5. SelfCartesian refers to the operation described at the beginning of Section 5.2.

The output of this Cartesian product is a 2D representation of the input where a square around the pixel $y^{(k,\ell)}$ contains local information about the nucleotides near $x^{(k)}$ and $x^{(\ell)}$. This makes convolutional neural networks an attractive option: we can use several convolutional layers to eventually output the probability that k and ℓ are paired together. However, translation invariance is not necessarily a safe assumption here: the probability that k and ℓ are paired together in the secondary structure depends on not only local information, but also potentially on all nucleotides between k and ℓ , and on the distance between k and ℓ , neither of which can be detected by a CNN. To remedy this, we prepend a bidirectional LSTM to the Cartesian product. With this addition, global information can propagate through this network, and information from arbitrarily far away in the sequence can influence the prediction about pair (k, ℓ) .

The architecture used for this problem is shown in Figure 5.1. We begin with the RNA sequence itself, one-hot encoded to be of size $L \times 5$, with each class representing A, C, G, U, and X, respectively. We first pass the sequence through a bidirectional LSTM to generate a hidden layer of size $L \times 20$. We then perform the self-Cartesian product on both the input sequence and the LSTM output, making arrays of size $L \times L \times 10$ and $L \times L \times 40$, respectively. These are stacked to make a single hidden layer of size $L \times L \times 50$.

We then run this image through several convolutional layers. Taking inspiration from high-performing image processing architectures [50], we use several different kernel sizes at each layer to capture features of varying size without a large increase in the number of parameters.

Because pairs are highly dependent on their constituent nucleotides (that is, only Watson-Crick pairs and wobble pairs may form), we concatenate the original sequence square with the penultimate hidden layer to allow the parameters in the output layer to directly access information about each nucleotide’s type. We found empirically that this increased accuracy and reduced the number of non-canonical base pair predictions.

Each of the hidden convolutional layers are given a ReLU activation function. The final layer’s activation function is an elementwise sigmoid, making the output an array $\hat{Y} \in \mathbb{R}^{L \times L}$ with $\hat{Y}^{(i,j)}$ representing the machine’s prediction that base pair (i, j) is in the secondary structure S of x when $i < j$. For the purposes of both training and testing, the diagonal and lower triangle of \hat{Y} are ignored.

Training and Hyperparameters

To train the parameters in this network, we used the Adam optimizer [30] with a batch size of 10 and an initial learning rate of $\eta = 10^{-3}$ that was reduced by half every 2500 iterations. As in Chapter 4, we also use a number of modifications to improve training speed and accuracy; we again use a regularization term, and found that applying batch normalization [27] after each convolutional layer had a profound effect on the speed and overall accuracy of training.

The loss function used is binary cross entropy applied to each of the $\frac{1}{2}L(L - 1)$ upper triangular entries of the output $\hat{Y} \in \mathbb{R}^{L \times L}$. The restriction that each nucleotide

may pair with at most one other nucleotide means that at most L of these entries will have positive labels. Thus, the machine sees many more negative training examples than positive examples, and this imbalance worsens as sequence length increases. In practice, this results in the machine being too conservative in its predictions. To combat this, we found that weighting the positive prediction term by a small constant, either 3 or 5 depending on the task, improved our final accuracy.

Depending on the dataset, we may have many sequences of varying sizes in the batch. To standardize the sequence length in a particular batch, we find the smallest sequence length L_0 , and choose a random length L_0 subsequence of each batch sequence. We further specify that the batch sequence length must be smaller than 500. This is primarily to avoid the exorbitant memory storage costs of long sequences; the hidden layer size $L \times L \times 50$ becomes unfeasibly large for even small batch sizes when training the network on a GPU. This restriction is consistent with hard constraint on the distance between nucleotides in base pairs predicted by NNTM as discussed in Chapter 3. We also note that this limitation exists only during training; at test time, we can run each sequence individually to conserve memory.

An implementation of this architecture using Keras [12] with the TensorFlow backend [1] is available at <http://github.com/dwillmott/ss-inf>.

Interpreting Network Output

The set of base pair predictions represented by \hat{Y} does not necessarily respect the rules of secondary structures - that is, there are no restrictions that prevent the machine from predicting that a nucleotide is paired with multiple other nucleotides, or that force the set of predicted base pairs to respect the nesting property of secondary structures. To convert \hat{Y} into a coherent prediction S of the native secondary structure, we follow an iterative process of selecting the base pair (k, ℓ) with the highest predicted probability in \hat{Y} , placing it in S , and setting the probability of all potential base pairs involving nucleotides k and ℓ to 0, to prevent the secondary structure from including overlapping base pairs. At each step, we can interpret each $\hat{Y}^{(i,j)}$ as the predicted conditional probability that (i, j) is paired given the partially constructed secondary structure S .

This iterative process is continued until there are no more positive predictions in the upper triangle of \hat{Y} , that is, all entries of \hat{Y} are below 0.5. This process is formally presented below as Algorithm 1.

As presented, Algorithm 1 does not enforce the nesting property - it will admit pseudoknots into the secondary structure. In contrast to all secondary structures in Chapters 3 and 4, we include pseudoknots in both the predicted and target secondary structure in experiments in this chapter. Pseudoknots break the loop structure that NNTM's dynamic programming algorithm relies on; we thus regard the ability to predict these pairs as an advantage of our method. If desired, we could also use a variant of this algorithm to construct a secondary structure with the nesting property by additionally removing base pairs with one nucleotide between k and ℓ and another outside of them at each step.

Algorithm 1 Algorithm to take a set \hat{Y} of base pair predictions and return a secondary structure prediction S

$P = \{(i, j) \mid 1 \leq i < j \leq L \text{ and } \hat{Y}^{(i,j)} > 0.5\}$

$S = \emptyset$

while $P \neq \emptyset$ **do**

$(k, \ell) \leftarrow \operatorname{argmax}_{(i,j) \in P} \hat{Y}^{(i,j)}$

$S \leftarrow S \cup \{(k, \ell)\}$

$P \leftarrow P \setminus \{(i, j) \mid i = k \text{ or } j = \ell\}$

end while

In addition to creating a valid secondary structure, our method gives a predicted probability for every possible pair. Thus, we can make this method’s prediction more or less conservative by adjusting the threshold 0.5 above which to include predictions. This flexibility is explored in depth in Section 5.3.

Datasets

We perform two main experiments using this architecture. In the first, we restrict attention to training and testing on 16S rRNA sequences. In the second, we widen our scope to consider sequences from a variety of RNA families, including other resolutions of ribosomal RNA as well as two families of ribozymes, Ribonuclease P and Group I Introns.

We display results from two test sets. The first is the sixteen 16S rRNA sequences in the Sukosd set [48] used in Chapter 4. We also use a subset of the sequences considered in [44], which we call the Rogers set. This is a group of 35 sequences, assembled to show diversity in sequence length and MFE accuracy. The set is partitioned into subsets of size 5, based on RNA type: 5S rRNA, Group I introns, Ribonuclease P, and four subsets of 16S rRNA divided by sequence length (small, medium, large, and extra large). On the 16S rRNA training task, we test the Sukosd set and the twenty 16S rRNA sequences in the Rogers set; since the Rogers and Sukosd sets contain both contain the *E. coli* and *E. cuniculi* sequences, this is 34 unique test sequences. (We note that these two sets contain slightly different versions of these sequences, which is why their accuracy varies slightly between test sets in the results reported below.) For the full training task, we consider all sequences in both test sets, for a total of 49 unique test sequences.

For both tasks, we trained on data freely available from RNA STRAND [3], a database that compiles RNA sequences with known secondary structures from a variety of sources. STRAND includes 723 sequences of 16S rRNA. For our larger training task, we incorporate the available 5S rRNA, 23S rRNA, Group I introns, and Ribonuclease P sequences, for a total of 1711 sequences.

We filter this training set against both test sets in a manner identical to the process used in Chapter 4. Finally, we reserve 5% of each family of RNA to use as a validation set. The final sizes of training, validation, and test sets are broken down by RNA family in Table 5.1.

Table 5.1: Enumeration of sequences in training, validation, and test sets by RNA family.

RNA Type	Training Set	Validation Set	Sukosd Set	Rogers set
16S rRNA	550	28	16	20
5S rRNA	58	3		5
Group I Intron	124	6		5
Ribonuclease P	380	20		5
23S rRNA	177	9		
Total	1289	66	16	35

For the 16S rRNA training task, we restrict attention to the sequences in the first row. In the full RNA training task, we include all types of RNA; total sizes of each set used in this task are shown in the final row.

Experimental Results

We present results from both the 16S rRNA training task as well as the full RNA training task in their respective subsections below. Throughout, we will refer to the accuracy of the raw predictions, which is the set of base pairs predicted by the neural network, as well as the accuracy of the structure prediction, the structure output from the algorithm presented in the previous section. These are presented primarily to display how the PPV and sensitivity of these two predictions vary.

5.3 16S rRNA Structure Inference

Table 5.2: Results on the 16S rRNA secondary structure inference task

Set	Raw Predictions		Structure Prediction		
	PPV	Sen	PPV	Sen	Acc
Training Set	0.796	0.949	0.933	0.941	0.937
Validation Set	0.766	0.856	0.899	0.845	0.872
16S Rogers Set	0.748	0.876	0.891	0.866	0.879
Sukosd Set	0.705	0.808	0.845	0.794	0.820
Test Set Total	0.730	0.843	0.867	0.831	0.849

Results from the 16S rRNA training task are available in Table 5.2. This table compares average PPV and sensitivity of both the raw neural network predictions and the coherent secondary structure prediction returned from Algorithm 1. This exhibits a trend that is reproduced throughout experiments in this chapter wherein the secondary structure prediction has markedly higher PPV than the raw base pair predictions. This is consistent with our expectations: if \hat{Y} gives a positive predictions for several conflicting base pairs, the algorithm will choose the highest probability

Table 5.3: Secondary structure inference accuracy on the Rogers set with a neural network trained on 16S rRNA sequences

Group	Sequence Name	Raw Predictions		Structure Prediction			MFE
		PPV	Sen	PPV	Sen	Acc	Acc
16S Small	V. ursinus	0.905	0.981	0.959	0.981	0.97	0.135
	S. aestuans	0.928	0.981	0.97	0.97	0.97	0.34
	L. catta	0.889	0.977	0.941	0.977	0.959	0.251
	N. robinsoni	0.956	0.981	0.989	0.981	0.985	0.447
	A. cahirinus	0.864	0.804	0.95	0.804	0.877	0.2
	Average	0.908	0.945	0.962	0.943	0.952	0.275
16S Medium	V. acridophagus	0.786	0.947	0.935	0.931	0.933	0.371
	V. corneae	0.631	0.571	0.764	0.557	0.661	0.33
	E. schubergi	0.64	0.638	0.835	0.622	0.728	0.23
	V. imperfecta	0.84	0.951	0.944	0.951	0.947	0.288
	E. cuniculi	0.644	0.672	0.807	0.669	0.738	0.17
	Average	0.708	0.756	0.857	0.746	0.801	0.278
16S Long	S. griseus	0.896	0.972	0.978	0.97	0.974	0.322
	M. leprae	0.824	0.977	0.967	0.977	0.972	0.179
	E. coli	0.834	0.933	0.969	0.929	0.949	0.41
	C. testosteroni	0.769	0.944	0.936	0.944	0.94	0.524
	M. hyopneumoniae	0.772	0.966	0.945	0.958	0.952	0.639
	Average	0.819	0.958	0.959	0.957	0.957	0.415
16S Extra	P. vivax	0.453	0.835	0.743	0.801	0.772	0.385
	R. carriebowensis	0.833	0.961	0.954	0.957	0.955	0.338
	O. cuniculus	0.671	0.957	0.893	0.941	0.917	0.177
	P. falciparum	0.347	0.807	0.656	0.764	0.71	0.423
	Z. mays	0.47	0.664	0.681	0.64	0.661	0.258
	Average	0.555	0.845	0.785	0.821	0.803	0.316
Total	Average	0.748	0.876	0.891	0.866	0.879	0.321

Table 5.4: Sequence results on the Sukosd set on the 16S rRNA secondary structure inference task

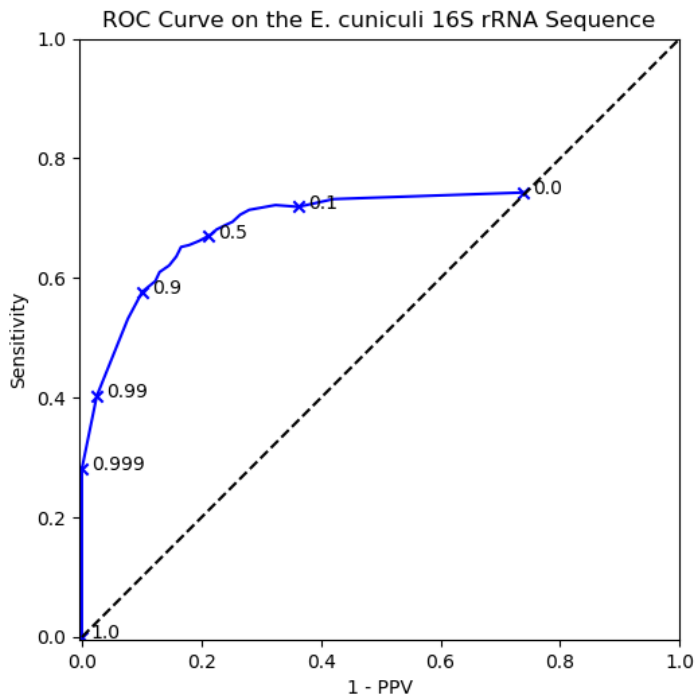
Sequence Name	Raw Predictions		Structure Prediction			MFE
	PPV	Sen	PPV	Sen	Acc	Acc
E. cuniculi	0.630	0.673	0.789	0.670	0.730	0.171
V. necatrix	0.677	0.636	0.800	0.617	0.709	0.181
C. elegans	0.211	0.132	0.188	0.088	0.138	0.203
E. nidulans	0.486	0.530	0.657	0.504	0.580	0.272
N. tabacum	0.778	0.912	0.931	0.912	0.921	0.323
Cryptonomas.sp	0.820	0.969	0.934	0.957	0.946	0.339
Synechococcus.sp	0.807	0.960	0.942	0.951	0.946	0.361
M. musculus	0.878	0.889	0.945	0.881	0.913	0.375
M. gallisepticum	0.690	0.875	0.887	0.862	0.874	0.385
E. coli	0.820	0.934	0.956	0.929	0.943	0.411
B. subtilis	0.802	0.972	0.952	0.970	0.961	0.512
D. desulfuricans	0.731	0.897	0.924	0.884	0.904	0.533
C. reinhardtii	0.804	0.902	0.918	0.897	0.908	0.537
T. maritima	0.737	0.901	0.924	0.901	0.913	0.562
T. tenax	0.691	0.899	0.888	0.871	0.880	0.618
H. volcanii	0.720	0.836	0.875	0.809	0.842	0.752
Average	0.705	0.807	0.844	0.794	0.819	0.408

prediction from among these; if the correct pair had the highest predicted probability, this will eliminate some false positives, and increase PPV.

Detailed results on each test set sequence, and a comparison between the accuracy of our method and MFE accuracy, is available in Table 5.3 for the Rogers set and Table 5.4 for the Sukosd set. From these we can see that accuracy is far higher using our method than MFE structures on all sequences on the Rogers set, and beats MFE accuracy on all but one sequence (C. elegans) in the Sukosd set. There is also a strong relationship between sequence length and accuracy, as illustrated by the 15 percentage point difference in average accuracy on small and long sequences versus medium and extra long sequences.

Comparison of Table 5.4 and Table 4.2 confirm that this method performs approximately as well or better than predicted state directed NNTM on the Sukosd set. Despite large differences between the two methods with respect to training set and network architecture, there are some striking similarities between their respective performance. In Section 4.4, we saw that the Sukosd set could be partitioned into three clusters based on the accuracy of predicted state directed NNTM. These clusters are almost exactly reproduced in accuracy from this method: the four sequences with accuracy under 80% in Table 5.4 are all in the low performance cluster; those with accuracy between 80% and 90% are the remaining low performance sequence (M. musculus) and the two sequences with medium performance (T. tenax

Figure 5.2: ROC curve examining the effect of varying the threshold v on accuracy on the *E. cuculi* test sequence. Labels indicate the value of v at that point.



and *H. volcanii*); and the sequences on which this method gives accuracy over 90% are exactly the nine sequences in the high performance cluster.

Even further, extraordinarily poor performance on *C. elegans* is reproduced throughout this method and predicted state directed NNTM. It is notable that unlike any other test set sequence, PPV and sensitivity both decreased after applying Algorithm 1 to the *C. elegans* raw base pair predictions, meaning the network's true positive predictions were not as confident as its false positive predictions. All of this suggests that the structural properties of *C. elegans* are perhaps poorly represented among sequences in the training set. This possibility is reinforced by its unusually small length: at 697 nucleotides, *C. elegans* is the fourth smallest 16S rRNA sequence, and only nine 16S rRNA sequences in the RNA STRAND database are smaller than 900 nucleotides.

Varying Prediction Threshold

Our method works by producing a predicted probability of each base pair, and using Algorithm 1 to iteratively add permissible base pairs to a predicted secondary structure until there are no remaining permissible base pairs with predicted probability above 0.5. If we choose, we may replace the value 0.5 with any threshold $v \in [0, 1]$. Larger or smaller values of v will result in a more or less selective predicted structure.

To examine our predictions in more detail, we tested the E. cuniculi sequence using a variety of thresholds v between 0 and 1. The resulting ROC curve is displayed in Figure 5.2, with several distinguished values of v labeled in the plot. This ROC curve looks slightly unusual, as we would normally expect a threshold of $v = 0$ to give a sensitivity of 0. However, Algorithm 1 chooses the most confidently predicted base pair and disregards all other possible base pairs involving the nucleotides in that base pair. Thus, employing Algorithm 1 with $v = 0$ on a sequence of length L will predict L pairs, whereas we would need to predict all possible $\frac{1}{2}L(L - 1)$ pairs to guarantee a perfect true positive rate.

Accuracy as we have defined it for secondary structures is constant along line with slope 1 in an ROC curve plot. Figure 5.2 shows that the PPV and sensitivity are relatively stable around $v = 0.5$; we receive roughly constant accuracy when v is between 0.3 and 0.9, with equal tradeoffs in sensitivity and PPV. Varying the threshold above and below these values results in quick decay of PPV and sensitivity, respectively.

We have also labeled two points generated by very high thresholds, $v = 0.99$ and $v = 0.999$. It is notable that even on a sequence with relatively low accuracy such as E. cuniculi, these high thresholds are able to provide strong predictions about base pairs in the secondary structure. Among the 159 base pairs with predicted probability above .99, 155 are in the native structure. If we further raise v to 0.999, we get perfect PPV: all 108 of these pairs are in the native structure, and these pairs constitute 28% of all native structure pairs. Thus, modifying this threshold to be very high can give us a partial predicted structure on which we are highly confident. This could be used to, for example, supply a hard constraint to NNTM, forcing it to find a MFE structure that respects these base pairings; constraints such as these are easily incorporated into NNTM in many implementations [49].

5.4 General RNA Structure Inference

Table 5.5: Results on the full RNA secondary structure inference task. Rogers Set (Other) refers to the 15 non-16S rRNA sequences in the Rogers set.

Set	Raw Predictions		Structure Prediction		
	PPV	Sen	PPV	Sen	Acc
Rogers Set (16S)	0.396	0.691	0.617	0.647	0.632
Rogers Set (Other)	0.530	0.462	0.587	0.436	0.512
Sukosd Set	0.370	0.681	0.605	0.637	0.621
Test Set Total	0.430	0.616	0.602	0.577	0.590

This section displays results from training the same architecture on the full STRAND training set. We expect that RNA families share some but not all patterns in the behavior of their nucleotides; thus, the additional RNA families in the training and test sets make this more difficult training task, as we are essentially increasing the expectations of the machine without expanding its capacity. From this, we do not

necessarily expect that the high accuracy of predictions in the previous experiment will be preserved in this new training context. In fact it is seen that they are not in Table 5.5, which shows a 20-25 percentage point drop in accuracy on 16S test sequences as compared to the previous experiment. However, from Table 5.6 we see that we still perform better on average on 16S sequences than on other RNA families, which in descending order have respective mean test accuracies of 59% (5S rRNA), 52% (Ribonuclease P), and 43% (Group I Introns).

Despite the drop in accuracy relative to the previous experiment, Tables 5.6 and 5.7 show that on both test sets this architecture outperforms NNTM on average by more than 20 percentage points, and among individual sequences, NNTM beats this architecture in predicted structure accuracy on only 6 out of the 49 test sequences: two 5S rRNA (*S. pombe*, *P. waltl*), one RnaseP (*Z. bailii*), one Intron (*B. yamatoana*), and two 16S rRNA (*C. elegans*, *H. volcanii*). We also note that for two of these sequences (*S. pombe* and *H. volcanii*), NNTM accuracy is already above 75%.

The lowered accuracy, as well as the high variance in accuracy among RNA types, is likely due to the disproportionate representation of sequences in the training set. For example, we find highest accuracy on 16S rRNA, which comprise just under half of the total number of sequences, and lowest accuracy on Group I introns, which account for less than 10% of training set sequences. As with any machine learning method, we are limited by the availability of data, and suspect that with enough training examples, this architecture could recover the extraordinary success of the 16S rRNA task.

Table 5.6: Structure inference accuracy on the Rogers set with the full training set

Set	Sequence Name	Raw Predictions		Structure Prediction			MFE
		PPV	Sen	PPV	Sen	Acc	Acc
5S rRNA	<i>S. pombe</i>	0.828	0.667	0.960	0.667	0.813	0.850
	<i>P. waltl</i>	0.657	0.657	0.679	0.543	0.611	0.760
	<i>O. sativa</i>	0.812	0.371	0.812	0.371	0.592	0.550
	<i>M. glyptostroboides</i>	0.565	0.371	0.706	0.343	0.524	0.290
	<i>M. fossilis</i>	0.522	0.343	0.529	0.257	0.393	0.150
	Average	0.677	0.482	0.737	0.436	0.587	0.520
Group I Introns	<i>H. rubra</i>	0.425	0.362	0.505	0.355	0.430	0.300
	<i>S. anglica</i>	0.235	0.275	0.262	0.246	0.254	0.060
	<i>B. yamatoana</i>	0.527	0.336	0.593	0.336	0.464	0.510
	<i>T. thermophila</i>	0.664	0.723	0.767	0.723	0.745	0.740
	<i>P. thunbergii</i>	0.198	0.381	0.186	0.302	0.244	0.130
	Average	0.410	0.415	0.463	0.392	0.427	0.348
Ribonuclease P	<i>H. chlorum</i>	0.485	0.360	0.525	0.348	0.437	0.320
	<i>T. syrichta</i>	0.257	0.220	0.276	0.195	0.235	0.130
	<i>P. fluorescens</i>	0.748	0.822	0.806	0.822	0.814	0.490
	<i>Z. bailii</i>	0.286	0.205	0.348	0.205	0.276	0.680
	<i>A. ferrooxidans</i>	0.734	0.833	0.851	0.833	0.842	0.590
	Average	0.502	0.488	0.561	0.481	0.521	0.442
16S rRNA Small	<i>V. ursinus</i>	0.557	0.801	0.777	0.762	0.770	0.135
	<i>S. aestuans</i>	0.548	0.773	0.738	0.735	0.736	0.340
	<i>L. catta</i>	0.583	0.817	0.784	0.802	0.793	0.251
	<i>N. robinsoni</i>	0.637	0.823	0.837	0.815	0.826	0.447
	<i>A. cahirinus</i>	0.576	0.700	0.770	0.681	0.725	0.200
	Average	0.580	0.783	0.781	0.759	0.770	0.275
16S rRNA Medium	<i>V. acridophagus</i>	0.358	0.641	0.574	0.583	0.578	0.371
	<i>V. corneae</i>	0.287	0.443	0.426	0.385	0.406	0.330
	<i>E. schubergi</i>	0.325	0.501	0.517	0.458	0.488	0.230
	<i>V. imperfecta</i>	0.419	0.675	0.616	0.610	0.613	0.288
	<i>E. cuniculi</i>	0.287	0.478	0.474	0.443	0.458	0.170
	Average	0.335	0.548	0.521	0.496	0.509	0.278
16S rRNA Long	<i>S. griseus</i>	0.456	0.874	0.792	0.846	0.819	0.322
	<i>M. leprae</i>	0.435	0.847	0.781	0.826	0.803	0.179
	<i>E. coli</i>	0.433	0.849	0.728	0.812	0.770	0.410
	<i>C. testosteroni</i>	0.394	0.789	0.669	0.751	0.710	0.524
	<i>M. hyopneumoniae</i>	0.384	0.728	0.625	0.669	0.647	0.639
	Average	0.420	0.817	0.719	0.781	0.750	0.415
16S rRNA Extra	<i>P. vivax</i>	0.196	0.588	0.381	0.522	0.452	0.385
	<i>R. carriebowensis</i>	0.336	0.724	0.569	0.670	0.620	0.338
	<i>O. cuniculus</i>	0.295	0.624	0.498	0.561	0.530	0.177
	<i>P. falciparum</i>	0.223	0.585	0.425	0.525	0.475	0.423
	<i>Z. mays</i>	0.196	0.558	0.365	0.484	0.424	0.258
	Average	0.249	0.616	0.448	0.552	0.500	0.316
Total	Average	0.453	0.593	0.604	0.557	0.580	0.370

Table 5.7: Sequence results on the Sukosd set on the full RNA secondary structure inference task

Sequence Name	Raw Predictions		Structure Prediction			MFE
	PPV	Sen	PPV	Sen	Acc	Acc
E. cuniculi	0.282	0.483	0.466	0.447	0.456	0.171
V. necatrix	0.320	0.496	0.478	0.452	0.465	0.181
C. elegans	0.111	0.088	0.124	0.077	0.100	0.203
E. nidulans	0.222	0.387	0.335	0.291	0.313	0.272
N. tabacum	0.406	0.769	0.690	0.723	0.707	0.323
Cryptonomas.sp	0.454	0.850	0.738	0.803	0.770	0.339
Synechococcus.sp	0.454	0.827	0.741	0.802	0.771	0.361
M. gallisepticum	0.581	0.722	0.734	0.667	0.700	0.375
M. musculus	0.388	0.697	0.643	0.645	0.644	0.385
E. coli	0.431	0.857	0.726	0.825	0.775	0.411
B. subtilis	0.433	0.870	0.779	0.844	0.812	0.512
D. desulfuricans	0.418	0.796	0.671	0.749	0.710	0.533
C. reinhardtii	0.395	0.763	0.681	0.747	0.714	0.537
T. maritima	0.355	0.808	0.681	0.757	0.719	0.562
T. tenax	0.341	0.742	0.606	0.692	0.649	0.618
H. volcanii	0.328	0.738	0.591	0.673	0.632	0.752
Average	0.370	0.681	0.605	0.637	0.621	0.408

Bibliography

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning.
- [2] Felix Altenberger and Claus Lenz. A non-technical survey on deep convolutional neural network architectures. *CoRR*, abs/1803.02129, 2018.
- [3] Mirela Andronescu, Vera Bereg, Holger H Hoos, and Anne Condon. Rna strand: the rna secondary structure and statistical analysis database. *BMC bioinformatics*, 9(1):340, 2008.
- [4] 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, pp. 1120–1128, New York, NY, 2016. JMLR.
- [5] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.
- [6] Jamie J Cannone, Sankar Subramanian, Murray N Schnare, James R Collett, Lisa M D’Souza, Yushi Du, Brian Feng, Nan Lin, Lakshmi V Madabusi, Kirsten M Müller, et al. The comparative rna web (crw) site: an online database of comparative sequence and structure information for ribosomal, intron, and other rnas. *BMC bioinformatics*, 3(1):2, 2002.
- [7] Jamie J. Cannone, Sankar Subramanian, Murray N. Schnare, James R. Collett, Lisa M. D’Souza, Yushi Du, Brian Feng, Nan Lin, Lakshmi V. Madabusi, Kirsten M. Müller, Nupur Pande, Zhidi Shang, Nan Yu, and Robin R. Gutell. The comparative rna web (crw) site: an online database of comparative sequence and structure information for ribosomal, intron, and other rnas. *BMC Bioinformatics*, 3(1):2, 2002.
- [8] Jitong Chen and DeLiang Wang. Long short-term memory for speaker generalization in supervised speech separation. *The Journal of the Acoustical Society of America*, 141(6):4705–4714, 2017.
- [9] Jonathan L Chen, Stanislav Bellaousov, and Douglas H Turner. Rna secondary structure determination by nmr. *Methods Mol Biol*, 1490:177–86, 2016.
- [10] Minmin Chen, Jeffrey Pennington, and Samuel Schoenholz. Dynamical isometry and a mean field theory of RNNs: Gating enables signal propagation in recurrent neural networks. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 872–881, Stockholmsmssan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

- [11] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.
- [12] François Chollet et al. Keras, 2015.
- [13] William W Cohen. *A computer scientist's guide to cell biology*. Springer Science & Business Media, 2007.
- [14] Katherine E Deigan, Tian W Li, David H Mathews, and Kevin M Weeks. Accurate shape-directed rna structure determination. *Proc Natl Acad Sci U S A*, 106(1):97–102, Jan 2009.
- [15] James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.
- [16] Laura DiChiacchio, Michael F Sloma, and David H Mathews. Accessfold: predicting rna–rna interactions with consideration for competing self-structure. *Bioinformatics*, 32(7):1033–1039, 2015.
- [17] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.
- [18] Jörg Fallmann, Sebastian Will, Jan Engelhardt, Björn Grüning, Rolf Backofen, and Peter F Stadler. Recent advances in rna folding. *Journal of biotechnology*, 261:97–104, 2017.
- [19] Boris Fürtig, Christian Richter, Jens Wöhnert, and Harald Schwalbe. Nmr spectroscopy of rna. *ChemBioChem*, 4(10):936–962, 2003.
- [20] Paul P Gardner and Robert Giegerich. A comprehensive comparison of comparative rna structure prediction approaches. *BMC bioinformatics*, 5(1):140, 2004.
- [21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
- [22] Robin R Gutell, Jung C Lee, and Jamie J Cannone. The accuracy of ribosomal rna comparative structure models. *Curr Opin Struct Biol*, 12(3):301–10, Jun 2002.
- [23] 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.
- [24] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

- [25] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.
- [26] Ivo L Hofacker, Peter F Stadler, and Peter F Stadler. Rna secondary structures. *Reviews in Cell Biology and Molecular Medicine*, 2006.
- [27] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.
- [28] L. Jing, C. Gülçehre, J. Peurifoy, Y. Shen, M. Tegmark, M. Soljačić, and Y. Bengio. Gated orthogonal recurrent units: On learning to forget. 2017.
- [29] 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, 2016.
- [30] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [31] D M Layton and R Bundschuh. A statistical analysis of rna folding algorithms through thermodynamic parameter perturbation. *Nucleic Acids Res*, 33(2):519–24, 2005.
- [32] S Y Le, J H Chen, and J V Maizel, Jr. Prediction of alternative rna secondary structures based on fluctuating thermodynamic parameters. *Nucleic Acids Res*, 21(9):2173–8, May 1993.
- [33] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [34] Zachary C Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.
- [35] Ronny Lorenz, Stephan H Bernhart, Christian Höner Zu Siederdisen, Hakim Tafer, Christoph Flamm, Peter F Stadler, and Ivo L Hofacker. Viennarna package 2.0. *Algorithms Mol Biol*, 6:26, Nov 2011.
- [36] Ronny Lorenz, Dominik Luntzer, Ivo L Hofacker, Peter F Stadler, and Michael T Wolfinger. Shape directed rna folding. *Bioinformatics*, 32(1):145–147, 2015.
- [37] Nicholas R Markham and Michael Zuker. Unafold: software for nucleic acid folding and hybridization. *Methods Mol Biol*, 453:3–31, 2008.
- [38] David H Mathews and Douglas H Turner. Prediction of rna secondary structure by free energy minimization. *Curr Opin Struct Biol*, 16(3):270–8, Jun 2006.

- [39] 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.
- [40] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.
- [41] E. O’Dorney. Minimizing the cayley transform of an orthogonal matrix by multiplying by signature matrices. In *Linear Algebra and its Applications*, volume 448, pp. 97–103. 05 2014.
- [42] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *ICML (3)*, 28:1310–1318, 2013.
- [43] Jessica S Reuter and David H Mathews. Rnastructure: software for rna secondary structure prediction and analysis. *BMC Bioinformatics*, 11:129, 2010.
- [44] Emily Rogers, David Murrugarra, and Christine Heitsch. Conditioning and robustness of rna boltzmann sampling under thermodynamic parameter perturbations. *Biophysical Journal*, 113(2):321–329, 2017.
- [45] David E Rumelhart and James L McClelland. Parallel distributed processing: explorations in the microstructure of cognition. volume 1. foundations. 1986.
- [46] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. Published online 2014; based on TR arXiv:1404.7828 [cs.NE].
- [47] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.
- [48] Zsuzsanna Sükösd, M Shel Swenson, Jørgen Kjems, and Christine E Heitsch. Evaluating the accuracy of shape-directed rna secondary structure predictions. *Nucleic Acids Res*, 41(5):2807–16, Mar 2013.
- [49] M Shel Swenson, Joshua Anderson, Andrew Ash, Prashant Gaurav, Zsuzsanna Sukosd, David A Bader, Stephen C Harvey, and Christine E Heitsch. Gtfold: Enabling parallel rna secondary structure prediction on multi-core desktops. *BMC Res Notes*, 5(1):341, Jul 2012.
- [50] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

- [51] Hakim Tafer, Fabian Amman, Florian Eggenhofer, Peter F Stadler, and Ivo L Hofacker. Fast accessibility-based prediction of rna-rna interactions. *Bioinformatics*, 27(14):1934–40, Jul 2011.
- [52] H. D. Tagare. Notes on optimization on stiefel manifolds. Technical report, Yale University, 2011.
- [53] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.
- [54] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2), 2012.
- [55] Douglas H Turner and David H Mathews. Nndb: the nearest neighbor parameter database for predicting stability of nucleic acid secondary structure. *Nucleic Acids Res*, 38(Database issue):D280–2, Jan 2010.
- [56] E. Vorontsov, C. Trabelsi, S. Kadoury, and C. Pal. On orthogonality and learning recurrent networks with long term dependencies, 2017.
- [57] Stefan Washietl, Ivo L Hofacker, Peter F Stadler, and Manolis Kellis. Rna folding with soft constraints: reconciliation of probing data and thermodynamic secondary structure prediction. *Nucleic Acids Res*, 40(10):4261–72, May 2012.
- [58] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.
- [59] Kevin A Wilkinson, Robert J Gorelick, Suzy M Vasa, Nicolas Guex, Alan Rein, David H Mathews, Morgan C Giddings, and Kevin M Weeks. High-throughput shape analysis reveals structures in hiv-1 genomic rna strongly conserved across distinct biological states. *PLoS Biol*, 6(4):e96, Apr 2008.
- [60] Kevin A Wilkinson, Edward J Merino, and Kevin M Weeks. Selective 2-hydroxyl acylation analyzed by primer extension (shape): quantitative rna structure analysis at single nucleotide resolution. *Nature protocols*, 1(3):1610, 2006.
- [61] 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*, pp. 4880–4888. Curran Associates, Inc., 2016.
- [62] Moritz Wolter and Angela Yao. Gated complex recurrent neural networks. *arXiv preprint arXiv:1806.08267*, 2018.

Vita

Devin Willmott

Education

University of Kentucky

Ph.D. in Mathematics

M.A. in Mathematics

Advisor: Qiang Ye

Lexington, KY

expected December 2018

May 2016

Marlboro College

B.A. in Mathematics (Highest Honors)

Advisor: Matthew Ollis

Marlboro, VT

May 2011

Professional Positions

Bosch Center for Artificial Intelligence

Research Scientist

Pittsburgh, PA

beginning October 2018

University of Kentucky

Teaching Assistant

Lexington, KY

August 2013 - May 2018

Publications

Kyle Helfrich, Devin Willmott (joint first author), Qiang Ye. Orthogonal Recurrent Neural Networks with Scaled Cayley Transform. *International Conference on Machine Learning*, 2018.

M.A. Ollis, Devin Willmott. Constructions for Terraces and R-Sequencings, Including a Proof That Bailey's Conjecture Holds for Abelian Groups. *Journal of Combinatorial Designs*, 23, 2015.

M.A. Ollis, Devin Willmott. There Are Infinitely Many Williams Squares with Circular Structure of Order a Multiple of 4. *Graphs and Combinatorics*, 30, 2014.

M.A. Ollis, Devin Willmott. An Extension Theorem for Terraces. *Electronic Journal of Combinatorics*, 20, 2013.

M.A. Ollis, Devin Willmott. On Twizzler, Zigzag, and Graceful Terraces. *Australasian Journal of Combinatorics*, 51, 2011.