University of Kentucky
UKnowledge

Theses and Dissertations--Mathematics | Mathematics

2023

Normalization Techniques for Sequential and Graphical Data

Cole Pospisil
University of Kentucky, cmpospisil@gmail.com
Author ORCID Identifier:
https://orcid.org/0000-0002-1907-139X
Digital Object Identifier: https://doi.org/10.13023/etd.2023.090

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation

Pospisil, Cole, "Normalization Techniques for Sequential and Graphical Data" (2023). *Theses and Dissertations--Mathematics*. 95.
https://uknowledge.uky.edu/math_etds/95

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.

Cole Pospisil, Student

Dr. Qiang Ye, Major Professor

Dr. Benjamin Braun, Director of Graduate Studies

Normalization Techniques for Sequential and Graphical Data

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
Cole M. Pospisil
Lexington, Kentucky

Director: Dr. Qiang Ye, Professor of Mathematics
Lexington, Kentucky
2023

Copyright© Cole M. Pospisil 2023
https://orcid.org/0000-0002-1907-139X

ABSTRACT OF DISSERTATION

Normalization Techniques for Sequential and Graphical Data

Normalization methods have proven to be an invaluable tool in the training of deep neural networks. In particular, Layer and Batch Normalization are commonly used to mitigate the risks of exploding and vanishing gradients. This work presents two methods which are related to these normalization techniques. The first method is Batch Normalized Preconditioning (BNP) for recurrent neural networks (RNN) and graph convolutional networks (GCN). BNP has been suggested as a technique for Fully Connected and Convolutional networks for achieving similar performance benefits to Batch Normalization by controlling the condition number of the Hessian through preconditioning on the gradients. We extend this work by applying it to Recurrent Neural Networks and Graph Convolutional Networks, two architectures which are prone to high computational costs and therefore benefit from the training acceleration provided by BNP. The second method is Assorted-Time Normalization (ATN). ATN is a normalization technique designed for use in sequential problems. It combines information from the hidden layers of the model with temporal data across the sequence dimension, which remedies a weakness of Layer Normalization in these applications.

KEYWORDS: Machine Learning, Preconditioning, Batch Normalization, Recurrent Neural Networks, Graph Convolutional Neural Networks

Cole M. Pospisil

April 25, 2023

Normalization Techniques for Sequential and Graphical Data

By
Cole M. Pospisil

Dr. Qiang Ye
Director of Dissertation

Dr. Benjamin Braun
Director of Graduate Studies

April 25, 2023
Date

Dedicated to my parents and my brother.

ACKNOWLEDGMENTS

This dissertation would not have been possible without the assistance of many people throughout the process. I would first like to thank my advisor, Dr. Qiang Ye, for his support and guidance. I would also like to thank my committee members for their time and agreeing to serve on my committee.

I would like to thank my friends for their support. Firstly, I'd like to thank Susanna and Vasily for the countless office chats and support throughout my time at UK. Next, I'd like to thank my board gaming groups, Ben and Darleen, Derek, Courtney, Justin, Matias, and also Josh, for helping keep me moderately sane. I would like to thank my office mates who I have spent far too much time with, particularly Rachel, Kyle, Joe, Chloe and Jamin. Finally I'd like to thank the rest of my fellow graduate students for being there for me as we went through this process together, particularly Landon (and Kat), Will, and Travis.

I would like to thank my family. My parents and brother have been a constant source of love, support, and inspiration throughout my life. I'd like to thank my aunt and uncle, Rosemary and Tony, without whose recipes I would have had far fewer delicious meals these last six years. I'd also like to thank the Mitchells and McRees, particularly Megan and Rebecca.

Finally I'd like to thank the University of Kentucky for use of the Lipscomb Computing Cluster and associated research computing resources, and the SMART Scholar program for their financial support in the latter half of my time at UK.

TABLE OF CONTENTS

Acknowledgments . . . iii

List of Tables . . . vi

List of Figures . . . vii

Chapter 1 Machine Learning Introduction . . . 1
1.1 Types of Machine Learning . . . 1
1.2 When to use Machine Learning . . . 2
1.3 Neural Networks . . . 3
1.3.1 Forward Propagation . . . 3
1.3.2 Back Propagation . . . 6
1.3.3 Data Processing . . . 7
1.4 Recurrent Neural Networks . . . 9
1.5 Graph Convolutional Networks . . . 12
1.6 Normalization . . . 15

Chapter 2 Batch Normalized Preconditioning (BNP) . . . 18
2.1 Preconditioned Gradient Descent . . . 18
2.2 BNP . . . 20
2.3 BNP for LSTM . . . 22
2.4 BNP for Graph Convolutional Networks . . . 27
2.4.1 Implementation . . . 29
2.4.2 Experiments . . . 31
2.4.3 Ablation Studies . . . 34
2.5 Conclusion . . . 38

Chapter 3 Assorted Temporal Normalization . . . 39
3.1 Introduction . . . 39
3.2 Related Work . . . 40
3.3 Assorted Time Normalization . . . 41
3.4 Experiments . . . 45
3.4.1 Synthetic Tasks . . . 45
3.4.2 Language Models . . . 50
3.5 Ablation Studies . . . 51
3.5.1 Input Statistic Invariance Across Time . . . 51
3.5.2 Post Normalization Statistics . . . 52
3.5.3 Optimal k Value for ATN method . . . 54
3.6 Invariance Properties . . . 55
3.7 Conclusion . . . 55

Chapter 4 NC-GRU 57
4.1 Introduction 57
4.2 Related Work 58
4.2.1 Gated Recurrent Unit (GRU) 59
4.2.2 Cayley Transform Orthogonal RNN 59
4.3 Efficient Orthogonal Gated Recurrent Unit 60
4.3.1 Gradient Analysis of Hidden States in GRU 60
4.3.2 Neumann-Cayley Orthogonal Transformation 61
4.3.3 Neumann-Cayley Orthogonal GRU (NC-GRU) 63
4.4 Experiments 64
4.5 Word Level Penn TreeBank 64
4.6 Ablation Studies 64
4.7 Conclusion 65

Bibliography 66

Vita 70

LIST OF TABLES

2.1 Results for Cora 31
2.2 Results for OGB Proteins 33
2.3 Results for OGB Products 34
2.4 Testing Loss for Normalization Ablation Study 36
2.5 Testing Accuracy for Normalization Ablation Study 36
2.6 Testing Loss and Accuracy for BNP ablation study 38

3.1 Copying Results: Attained minimum values. ↓ - denotes the smaller, the better result. 47
3.2 Adding Results: Attained minimum values. ↓ - denotes the smaller, the better result. 49
3.3 Denoise Results: Attained minimum values. ↓ - denotes the smaller, the better result. 50
3.4 Character Level Penn Treebank Results: Attained minimum values. ↓ - denotes the smaller, the better result 51
3.5 Invariance properties under different normalization methods 55

4.1 Word Level PTB Results: Evaluated perplexity (PPL) for every model. * - result obtained from our experiments; † - result quoted from [Bai et al., 2018] 64

LIST OF FIGURES

1.1 A Fully Connected Neural Network with bias terms for each hidden layer 4
1.2 Common Nonlinear Activation Functions 6
1.3 A Recurrent Neural Network 10
1.4 An Example of a Graph 13

2.1 Results for Adding Problem: T = 400 26
2.2 Results for Adding Problem: T = 750 27
2.3 Results for Copying Problem: T = 1000 28
2.4 Results for Copying Problem: T = 2000 28
2.5 Accuracy Curves for Cora 32
2.6 Accuracy Curves for OGB Proteins 33
2.7 Accuracy Curves for OGB Products 35
2.8 Normalization Ablation Study 37
2.9 BNP on different layers 37

3.1 Illustration of the ATN method combined with LN using $k = 3$ time steps: Consider the preactivation state tensor in $\mathbb{R}^{5\times3\times3}$. At $t = 1$, we use a standard LN; at $t = 2$, we normalize using information from time steps $1, 2$; at $t = 3$ ATN method uses information from time step $t = 1, 2, 3$; at $t = 4$, we normalize with respect to time steps $t = 2, 3, 4$; and so on after that. 42
3.2 Copying problem with $T = 100$ 46
3.3 Copying problem with $T = 200$ 46
3.4 Adding problem with $T = 100$ 48
3.5 Adding problem with $T = 200$ 48
3.6 Denoise task with $T = 100$ 49
3.7 Denoise task with $T = 200$ 50
3.8 Pixel-by-pixel MNIST 51
3.9 Hidden-to-Hidden 53
3.10 Input-to-Hidden 53
3.11 Memory Cell 54
3.12 k value study in ATN method 54

Chapter 1 Machine Learning Introduction

In recent years, Machine Learning has found itself as the buzzword of choice in the computer programming adjacent world and as such its use in many aspects of life has become ubiquitous. Since it has become a catchall term in the public consciousness, similar to the use of quantum in popular science fiction, it is important to clarify what exactly is meant when talking about Machine Learning. To that end, this section will first discuss some general ideas about Machine Learning: what it is, how it works, and what its ideal uses are. From there, we will discuss in more detail the particular architectures upon which the later sections of this work will be based, Recurrent Neural Networks and Graph Convolutional Networks. Finally, we will give an overview of two popular normalization algorithms, Batch Normalization and Layer Normalization, which use various statistics from their inputs to rescale and recenter a model's hidden states and improve performance.

1.1 Types of Machine Learning

At its most basic level, Machine Learning is any method by which a model takes in data and adapts its parameters to perform some task based upon that data. These tasks generally fall into two main categories: regression and classification. In regression tasks, the goal of the model is to produce a value on a continuous scale, while classification tasks sort datasets into different discrete categories. Examples of regression tasks are things such as determining how positive or negative a product review is, or predicting interest rates based on economic factors. Classification tasks can include determining which subfield an academic paper belongs in, or determining which digit a handwritten number is supposed to be.

The different varieties of these tasks generally fall within three main categories, usually depending on the available dataset: supervised learning, unsupervised learning, and semi-supervised learning. Supervised learning is when the dataset is completely labeled. For each datapoint, the model is given either the category that it belongs in, or the value that the regression model is supposed to output. This setup is the most straightforward when it comes to determining how well a model performs since every example comes with a clear definition of success or failure. With classification tasks, either the model correctly identifies the associated label and succeeds, or it mislabels the example and fails. For regression tasks, there is also generally an idea of the scale of the model's success since the labels are on a continuum which allows for an idea of distance. So we are able to tell a difference between predicting something that is wildly different from the true value, versus something that is just a bit off. The tasks in the rest of this work fall in the category of supervised learning

Unsupervised learning happens when the model is given data which does not have associated labels. Examples of unsupervised learning include anomaly detection and clustering. The lack of labels means that the model has the preliminary task of deciding what the data should look like before it can move on to the general regression

or classification tasks. In anomaly detection, this means that the method builds an understanding of what the majority of data points looks like, what trends to expect, and so on, and once it has built this expectation it can then provide a score describing how similar a given example is to what is expected and whether or not it is likely to be an anomaly. For clustering tasks, the goal is to sort the examples into categories like in a classification problem, but the model must first determine what those categories should be. An example of this would be image sorting. When given a selection of pictures of farmyard animals, the model can recognize that things with udders go together and so do things with curly tails without needing to be told that they are cows or pigs.

Semi-supervised learning, as the name suggests, is a mix between supervised and unsupervised learning. In these tasks, a mixture of labeled and unlabeled data is fed into the model. This helps to blend the strengths of these approaches and is generally more feasible for novel applications. One major problem with supervised learning is that the creation of labeled data in many applications involves a large expenditure of time and effort, potentially very expensive time if the application is in a specialized field which requires particular knowledge or expertise to create the training set. A problem with unsupervised learning is that models are generally able to perform better when they have labels off of which to work. So by utilizing a blend of labeled and unlabeled data, semi-supervised learning is largely able to mitigate these issues and give better results than unsupervised learning at a smaller cost than supervised learning.

1.2 When to use Machine Learning

When it comes to the ideal use case for machine learning, there are a number of factors that a particular task should have. The first of these is that the end goal should be some form of generalization. If everything that the model is trying to predict is already known, then it is much more efficient to use any number of search algorithms or look-up tables to perform the task. The idea behind the machine learning algorithm is to recognize relationships between patterns and then to find those patterns in new data. The next thing that a machine learning task should have is a suitably large scale. Machine learning models need a lot of information to train efficiently. Without access to enough information, the model will not be able to learn properly, or even worse, could learn incorrect patterns. Finally, the desired task should be something that can be determined with the available inputs. If there is some independent variable which is omitted entirely from the dataset, then the model is only really capable of viewing its effects as random noise and cannot be expected to make truly accurate predictions.

1.3 Neural Networks

1.3.1 Forward Propagation

There are a large number of architectures for machine learning models, such as binary trees and support vector machines, but this work will be focusing on neural networks. Neural networks in their most basic form consist of stacked layers of weights and nonlinear activation functions between them; these are called feed-forward neural networks since all of the information flows in one direction from the input to the output. This takes a given input, $x_1, x_2, \ldots, x_n$, and produces outputs $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m$, via a sequence of what are called hidden states, $h_1^{(l)}, h_2^{(l)}, \ldots, h_k^{(l)}$, at each interior layer, l. For each of these states, there is an associated weight between them, $w_{i,j}^{(l)}$, that takes $h_i^{(l)}$ to $h_j^{(l+1)}$. Every weighted input into a particular state is then added together and fed into a nonlinear activation function, σ, to create the new state. So, for a given state $h_j^{(l+1)}$, its update equation is given by

$$h_j^{(l+1)} = \sigma(w_{1,j}^{(l)} h_1^{(l)} + w_{2,j}^{(l)} h_2^{(l)} + \ldots + w_{k,j}^{(l)} h_k^{(l)}).$$

These weights and states are generally given in a more concise form as matrix operations, where the vector $h^{(l+1)}$ is updated using the product of the matrix $W^{(l)} = [w_{i,j}]$ and the vector $h^{(l)}$. This equation being

$$h^{(l+1)} = \sigma(W^{(l)} h^{(l)}) = \sigma\left(\begin{bmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \ldots & w_{1,n}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \ldots & w_{2,n}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,1}^{(l)} & w_{k,2}^{(l)} & \ldots & w_{k,n}^{(l)} \end{bmatrix} \cdot \begin{bmatrix} h_1^{(l)} \\ h_2^{(l)} \\ \vdots \\ h_n^{(l)} \end{bmatrix}\right).$$

These models are called fully connected neural networks, a diagram of which can be seen in Figure 1.1. Despite the name, not every term in every layer needs to be connected to previous layers. In fact, most models have a specific term for each layer that is isolated from the previous portion of the model; this is called the bias term, $b^{(l)}$. This provides a weight which is generally taken as a separate vector that is added to the product of the weights and the nodes:

$$h^{(l+1)} = \sigma(W^{(l)} h^{(l)} + b^{(l)}) = \sigma\left(\begin{bmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \ldots & w_{1,n}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \ldots & w_{2,n}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,1}^{(l)} & w_{k,2}^{(l)} & \ldots & w_{k,n}^{(l)} \end{bmatrix} \cdot \begin{bmatrix} h_1^{(l)} \\ h_2^{(l)} \\ \vdots \\ h_n^{(l)} \end{bmatrix} + \begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_n^{(l)} \end{bmatrix}\right).$$

Bias can be included in the hidden state to form an augmented hidden state by creating a $h_0^{(l)}$ whose associated weights are $w_{0,j} = b_j$. As we go through the layers,

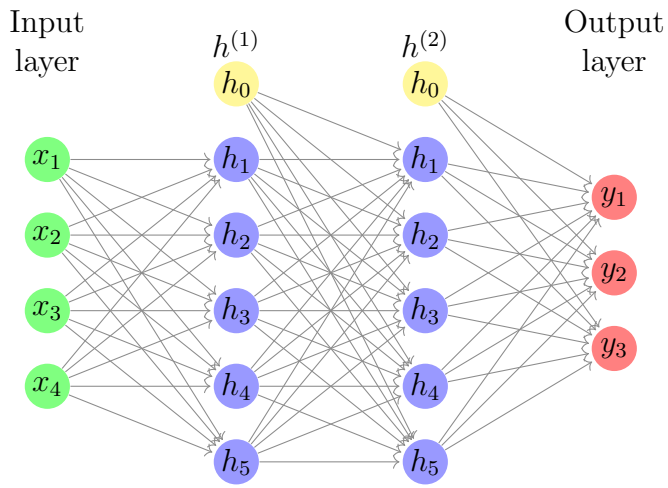


Figure 1.1: A Fully Connected Neural Network with bias terms for each hidden layer

each state builds on the previous ones so that we end up with a nested expression

$$\begin{aligned} h^{(3)} &= \sigma(W^{(2)}h^{(2)} + b^{(2)}) \\ &= \sigma(W^{(2)}\sigma(W^{(1)}h^{(1)} + b^{(1)}) + b^{(2)}) \\ &= \dots \\ &= \sigma(W^{(2)}\sigma(W^{(1)}\sigma(W^{(0)}x + b^{(0)}) + b^{(1)}) + b^{(2)}) \end{aligned}$$

Because we are using a nonlinear activation function, σ, this cannot be simplified further which allows the model to be more expressive than simple linear transformations. The choice of function is dependent on the model's architecture and intended task, but we generally use functions which are well-behaved. Since the learning process for neural networks involves the computation of gradients, the behavior that we want from our activation functions is being continuous and nicely differentiable almost everywhere. Some of the most common of these are the Sigmoid function, Hyperbolic Tangent, Rectified Linear Unit (ReLU) and Leaky ReLU. Graphs of all of these functions can be found in Figure 1.2.

The Sigmoid function refers to a particular logistic curve whose equation is

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

It is an S shaped function whose limit towards negative infinity is zero and whose limit towards positive infinity is one. Because its range is between zero and one it is a natural choice when picturing an output as a probability. This is generalized into the SoftMax function which is used to give probabilities for a set of categories $1, \dots, C$

$$\text{SoftMax}_j(x) = \frac{e^j}{\sum_{i=1}^{C} e^i}.$$

An advantage of the Sigmoid function is that its gradient is rather easily computed since

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

Since we can nicely parameterize the derivative of the Sigmoid function in terms of the function itself, and we will have already computed the function values on the forward pass, Sigmoid is a nice choice for gradient descent based neural networks. One potential issue that we have comes when the x values are particularly large, in either the positive or negative direction. Because Sigmoid approaches zero and one at the left and right tail, extreme values of x will give derivatives which are very close to zero. This is called the vanishing gradient problem. Since learning is based on the gradients, if they are too close to zero the model will not learn anything and training will stagnate.

Hyperbolic Tangent is another example of an S shaped logistic curve. Its equation is

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Similarly to the Sigmoid function, hyperbolic tangent benefits from being in a set range, from negative one to positive one. It also has a nicely computed derivative since

$$\tanh'(x) = \frac{(e^x + e^{-x})^2 - (e^x - e^{-x})^2}{(e^x + e^{-x})^2} = 1 - \tanh^2(x).$$

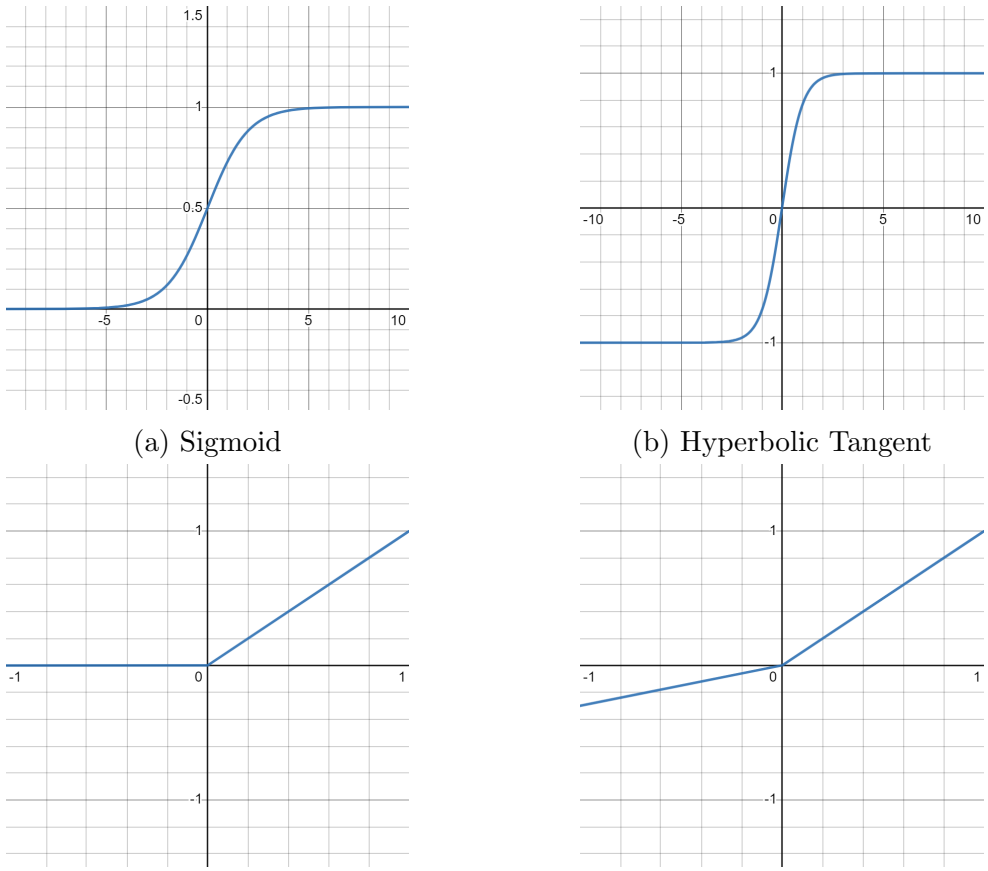
This also has issues with vanishing gradients towards infinity, but the gradients for values close to 0 are slightly larger than they would be for the sigmoid function which can contribute to faster convergence with otherwise equivalent models.

Rectified Linear Unit (ReLU) is a piecewise linear function given by the equation $\mathrm{ReLU}(x) = \max(0, x)$. The simplicity of the function and its derivative are big draws for ReLU, since the derivative is just 1 for positive values and 0 for negative ones. This is not differentiable at 0, but standard practice is to use a derivative of 0 there. A downside of this function is that it fails to provide information on negative values. Since everything is just mapped to 0, there is no visible difference between small negative values and large ones. To fix this, ReLU has been generalized to Leaky ReLU, given by the piecewise function:

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

The alpha factor allows for some information to pass through from negative values, but is generally chosen to be close to zero so that there is still an appreciable difference between positive and negative entries.

Once the data has been passed through all the layers, the model output and the ground truth labels are fed into a loss function, $\mathcal{L}$, which determines how erroneous the predictions are. The particular loss function that is chosen depends on the task and the model, but they are almost always differentiable metric functions. This is because the desired outcome of a neural network is generally substituted by the task of minimizing the loss function through a gradient descent algorithm. To use gradient descent, we need gradients and therefore our function needs to be differentiable; to have the optimization problem be meaningful, we need the positive definiteness,



(a) Sigmoid

(b) Hyperbolic Tangent

(c) ReLU

(d) Leaky ReLU

Figure 1.2: Common Nonlinear Activation Functions

reflexivity, and triangle inequality that come with metrics. For regression tasks, one of the most common loss functions is Mean Squared Error or L2 loss,

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where y_i is the true label for a given training example and $\hat{y}_i$ is the corresponding predicted label. For classification tasks, Cross Entropy Error is one of the most common loss functions,

$$CrossEntropyLoss = -(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i))$$

While there are other metrics on which we could want to judge a model's performance, like accuracy for prediction tasks or semantic values for translation tasks, those can only be improved upon if they are related to the chosen loss function since that is all that the model is capable of directly optimizing.

1.3.2 Back Propagation

Once the loss has been computed, the portion of training called forward propagation is finished and back propagation begins. In this stage, we work on optimizing the

loss function through an iterative process called gradient descent. First we calculate gradients of the loss function with respect to the weights and biases of the model.This is why our activation and loss functions were chosen to be easily differentiated as gradient calculations can become computationally expensive for models which have more layers, called deep models. The back propagation process begins at the output layer and moves backwards through the model until reaching the input. This reversed direction is because later layers are dependent on earlier layers in the forward pass, so the error in earlier layers is dependent on the error later on. This can be seen by using the chain rule to expand the gradient:

$$\frac{\partial \mathcal{L}}{\partial w_{i,j}^{(l)}} = \frac{\partial \mathcal{L}}{\partial h_j^{(l+1)}} \frac{\partial h_j^{(l+1)}}{\partial w_{i,j}^{(l)}}$$

Adding an additional step to this expansion shows how quickly the dependencies can grow. Since every hidden unit in the next layer is a function of $h_j^{(l+1)}$, we get a term for each of them. So we now have the following sum for our gradient:

$$\frac{\partial \mathcal{L}}{\partial w_{j,k}^{(l)}} = \sum_{k=1}^{n} \frac{\partial \mathcal{L}}{\partial h_k^{(l+2)}} \frac{\partial h_k^{(l+2)}}{\partial w_{j,k}^{(l+1)}} \frac{\partial w_{j,k}^{(l+1)}}{\partial h_j^{(l+1)}} \frac{\partial h_j^{(l+1)}}{\partial w_{i,j}^{(l)}}$$

Once we have the gradients, we are able to predict the changes that the loss function will undergo based on changes to the weights. As we approach a minimized loss function, we expect the gradients to approach zero, so we are able to interpret the size of the gradients as being correlated with the size of the change we need to make to that particular weight. The sign of the gradient tells us the direction that the weight update needs to be applied. Since we want to approach zero, we always want to move against the gradient, analogous to moving downhill. Formulaically, this is expressed as

$$\tilde{w} = w - \alpha \frac{\partial \mathcal{L}}{\partial w}$$

where the original weight is w, the updated weight is $\tilde{w}$, and α is the learning rate. The choice of a learning rate is a key component to training any gradient descent based machine learning model. Ideally, the learning rate is small enough that it controls the scale of the gradient so that the update step does not overshoot the minimum, while being large enough so that the update steps are not too small to efficiently train. Once each of the weights has been updated, the model is ready for another set of inputs.

1.3.3 Data Processing

While it is theoretically possible for a model to take in every possible input each training iteration, there are a multitude of reasons why this typically does not happen in practice. The first of these is that seeing every potential input would make it very difficult to assess how well-suited a model is for practical performance. The ideal use case for a machine learning model is to make predictions for a set of inputs it has

not seen before. If we use all of our data in training, then we have no reference for how well our model can process something new. To fix this, we generally separate our dataset into three groups: a training set, a validation set, and a testing set. The training set is used for the forward and back propagation steps mentioned above. The testing set is reserved until training is completed, in order to give an accurate picture of how the model might perform in real world inference tasks. The validation set is used during training as a surrogate for the testing set. Every so often, the validation set is used as input for the forward propagation step to test the model's performance on data outside of the training set. The validation loss is not used for a gradient update, but can be used as a signal to change certain settings to help convergence. Most of these settings have to do with avoiding a phenomenon called overfitting. This is when the model learns the distribution of the training set so well that it becomes less capable of generalizing to other sets. Signs of this are when there is considerable gap between the training and validation losses and when the validation loss starts to stagnate or even increase while the training loss is still decreasing. In order to avoid this problem, we can schedule changes to the model's parameters based on the behavior of the validation loss. Some examples of this are learning rate decay and early stopping. With learning rate decay, we decrease the learning rate when the validation loss tells us that we are at risk of overfitting, usually by plateauing or starting to increase. This also has the benefit of allowing faster initial convergence and finer control of the parameters as we get closer to the minimum, since we are not stuck with either a large or small learning rate. Early stopping, as the name suggests, is when the validation loss is used to trigger the end of training. This allows the model to get an idea of when it was best at inference and ignore the changes past that point. Without early stopping, we can only hope that the model became the best it could be whenever we told it to stop training.

Once we have the training, validation, and testing splits decided, we begin forward and back propagation with the training set. Whenever the model has seen the entire training set, we call this an epoch. Training continues with occasional uses of the validation set until a set epoch limit is reach, when the final model state is determined and the testing set is used to give a final performance metric. The number of epochs depends on a number of factors such as the complexity of the model, the size and complexity of the dataset, computational or time constraints, and more.

In many cases, the size of the training set can be too large for all of it to be fed into the model at once. When this happens, we need to break the training set into smaller mini-batches. These are small subsets of training examples chosen at random from the training set without replacement. Once the mini-batch is fed through forward propagation, we then compute the mini-batch gradients and use those to update the model's parameters. Since these mini-batch gradients are not necessarily the same as what we would get with whole batch training, this process is called stochastic gradient descent.

1.4 Recurrent Neural Networks

The remainder of this chapter will focus on specialized architectures and methods which will be used in the following chapters. The first of these architectures is the Recurrent Neural Network (RNN). These are models which are designed to be used with sequential data, such as words or sentences. Popular tasks for these models include things like machine translation and natural language processing. The primary problem with using Fully Connected Neural Networks for these tasks is that the inputs do not have a set length. For example, suppose that we want to predict the next word in a sentence. The first thing we need to do is embed the existing words in vector form, generally of a set length e.g. vectors of length 20. If we feed that in as a concatenated input, then the initial weight matrix needs to be $20 \times (number\ of\ words) \times (hidden\ size)$. This then locks us into only being able to predict the sixth word in a sentence, or at least only being able to look back five words. So we want a fix that prevents the length of the input from changing with the number of words. An option is to give each word its own weight matrix, but then we do not have a trained matrix for our words if any sentence in the testing or validation sets are longer than the sentences in the training set. We also want to capture the dependencies as we go through a sentence, for example, if we see the word 'neither' then we should be expecting to see 'nor' soon. To see these relationships, we want words further in the sentence to be included deeper in the model, but then we have the same issue of needing long enough sentences in our training set to have enough layers for the testing set. The Recurrent Neural Network fixes these problems by reusing weights, allowing it to be as long as needed to include inputs of any length. It also brings in a new word at each layer, so we can see those dependencies in the sequence direction. The forward step of an RNN is

$$h^{(t)} = \sigma(Wh^{(t-1)} + Ux^{(t)} + b)$$

where the trainable parameters are the hidden to hidden weight, W, the input to hidden weight, U, and the bias term, b, and σ is a nonlinear activation function. In many cases, we want to have a many-to-many mapping which takes a sequence as input and provides another sequence as output. To do this, we have an output weight, C, which produces an output, $y^{(t)}$, corresponding to each hidden state, $h^{(t)}$. We can adjust to one-to-many, many-to-one, or one-to-one mappings by omitting the corresponding connections. A graphical representation of an RNN can be seen in Figure 1.3

While reusing the weights simplifies the forward step, the back propagation becomes more complicated. The specialized process of calculating gradients for RNNs is called back propagation through time (bptt). For the output weights, there are no sequence direction dependencies since $y^{(t)}$ never reenters the model. Therefore the gradient is just the sum of the gradients at each step

$$\frac{\partial \mathcal{L}}{\partial C} = \sum_{t=1}^{\tau} \frac{\partial \mathcal{L}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial C}$$

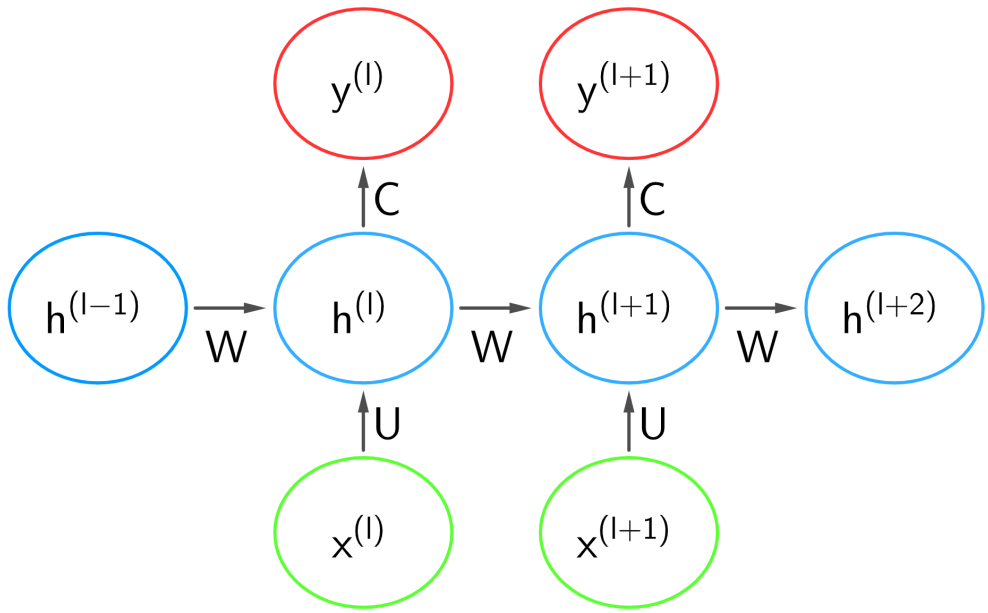


Figure 1.3: A Recurrent Neural Network

The complication for the hidden-to-hidden weights happens because there are multiple paths from the weight to the loss function and the weight is reused in some of those paths. For example, we will look at the gradient of $\mathcal{L}$ with respect to W at time step l which we will specify using $W^{(t)}$. First, we have a path to the loss function through the output at that time step, so we have one term

$$\frac{\partial \mathcal{L}}{\partial W^{(t)}} = \frac{\partial \mathcal{L}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial W^{(t)}} + f$$

We also have a path by first going to the next time step and then going to the output layer. This gives us the term

$$\frac{\partial \mathcal{L}}{\partial W^{(t)}} = \frac{\partial \mathcal{L}}{\partial y^{(t+1)}} \frac{\partial y^{(t+1)}}{\partial h^{(t+1)}} \frac{\partial h^{(t+1)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial W^{(t)}} + g$$

In fact, we get a term like that for each time step in the sequence after t. Writing out the full expansion becomes unwieldy, so we combine all of these terms into one via the chain rule. This gives us that the contribution to the gradient at time step t is

$$\frac{\partial L}{\partial W^{(t)}} = \frac{\partial L}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial W^{(t)}}$$

Since the same weight is used at every time step, the entire gradient is given by the aggregate of each of these gradients

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_{t=1}^{\tau} \frac{\partial \mathcal{L}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial W}$$

A similar calculation gives the gradients for the input weights

$$\frac{\partial \mathcal{L}}{\partial U} = \sum_{t=1}^{\tau} \frac{\partial \mathcal{L}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial U}$$

One major problem that RNNs have is that they are particularly susceptible to exploding and vanishing signals. In forward propagation, the reuse of the hidden-to-hidden weight matrix can easily cause the contribution of the hidden states to vanish or explode if the matrix norm is not suitably close to one. While the activation function does influence the behavior, we still have that by time step t, the initial hidden state will have essentially been multiplied by W t times. If the weight matrix is too small, this will zero out the initial steps and eliminate any information from the start of the sequence. If the weight matrix is too large, then it will cause divergence. One of the methods that has been used to help combat this problem with vanishing signals is the inclusion of gates into the model. This gives us two specialized versions of an RNN: the Long Short Term Memory Model (LSTM), and the Gated Recurrent Unit Model (GRU)

The LSTM was first proposed by [Hochreiter and Schmidhuber, 1997] and builds off of the RNN structure by creating a more complex method for an activation function and adding a memory cell $c^{(t)}$ to preserve information across time. It features three different gates: the forget gate f, the input gate i, and the output gate o. These are found by calculating the product of the hidden state with a weight matrix and then feeding the result into a sigmoid function to produce a value between zero and one. If a gate is close to zero, it represents being closed and will prevent information from passing through it, while a value close to one represents being open and having unimpeded flow through the gate. The idea behind this is that the model will learn when information is important and will preserve those dependencies while forgetting things that are no longer relevant. The structure of the LSTM is

$$\begin{bmatrix} \mathbf{f}^{(t)} \\ \mathbf{i}^{(t)} \\ \mathbf{o}^{(t)} \\ \mathbf{g}^{(t)} \end{bmatrix} = \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)} + \mathbf{b}$$

$$\mathbf{c}^{(t)} = \sigma(\mathbf{f}^{(t)}) \odot \mathbf{c}^{(t-1)} + \sigma(\mathbf{i}^{(t)}) \odot \tanh\,(\mathbf{g}^{(t)})$$
$$\mathbf{h}^{(t)} = \sigma(\mathbf{o}^{(t)}) \odot \tanh\,(\mathbf{c}^{(t)})$$

where $\mathbf{W} \in \mathbb{R}^{4d_h \times d_h}$, $\mathbf{U} \in \mathbb{R}^{4d_h \times d_x}$ and $\mathbf{b} \in \mathbb{R}^{4d_h}$, σ is the sigmoid function, tanh is the hyperbolic tangent, and $\odot$ is element-wise matrix mulitplication, also called the Hadamard product.

The Gated Recurrent Unit was proposed in [Cho et al., 2014] as an alternative to the LSTM and features a number of differences. Instead of having the long-term memory on its own, the GRU combines it as part of the hidden state. GRU also has fewer gates than LSTM, featuring only an update gate u and a reset gate r. The update gate takes the role of both the input and forget gates. If u is zero then the forget gate is closed and the input gate is open, and vice versa for when u is one.

This lets the GRU replicate the long-term dependency behavior of the LSTM while having fewer parameters. The structure of the GRU is

$$
\begin{aligned}
r^{(t)} &= \sigma\left(W_r x^{(t)} + U_r h^{(t-1)} + b_r\right) \\
u^{(t)} &= \sigma\left(W_u x^{(t)} + U_u h^{(t-1)} + b_u\right) \\
c^{(t)} &= \tanh\ \left(W_c x^{(t)} + U_c\left(r^{(t)} \odot h^{(t-1)}\right) + b_c\right) \\
h^{(t)} &= \left(1 - u^{(t)}\right) \odot h^{(t-1)} + u^{(t)} \odot c^{(t)}
\end{aligned}
$$

where W_r, W_u, and W_c are input weights in $\mathbb{R}^{n\times m}$, U_r, U_u, and U_c are recurrent weights in $\mathbb{R}^{n\times n}$, and b_r, b_u and b_c are the bias parameters in $\mathbb{R}^n$. Here, m represents the dimension of the input data and n represents the dimension of the hidden state. The activation functions σ and tanh are sigmoid and hyperbolic tangent function respectively, and $\odot$ is the Hadamard product. This can also be represented as a stacked matrix product like the LSTM.

1.5 Graph Convolutional Networks

Graph Convolutional Networks are Neural Networks which are designed to run on inputs that are mathematical graphs, also called networks. A mathematical graph is a collection of nodes or vertices, V, and connections or edges, E, where each edge connects two vertices in V. These have become particularly interesting to the machine learning community recently because they provide the flexibility to represent many of the datasets that people want to know things about. In general, any task that relies on knowing something about objects and relationships between them can be represented using a graph. A Graph Convolutional Network (GCN) differs from Fully Connected Networks because they include an aggregation method to collect information from neighboring nodes as well as the usual weight multiplication and nonlinear activation. The most basic GCN cell, proposed by [Kipf and Welling, 2017], is

$$
x_i^{(t)} = \phi\left(A x_i^{(t-1)} W^{(t-1)}\right)
$$

where $x_i^{(t)}$ is the value of the ith node at the tth time step, $W^{(t)}$ is a trainable weight matrix, and A is the augmented adjacency matrix. The adjacency matrix is where the information about a graph's edges is contained. For nodes i and j, if there is an edge between them then the $(i,j)^{th}$ entry of A is 1, otherwise it is 0. For more complicated graphs, these values can change. For instance, a weighted graph would have the edge's weight instead of 1 and a directed graph would have positive or negative values if the edge goes from i to j or from j to i respectively. For the purposes of GCNs, we augment the adjacency matrix in two ways. First, we add the identity matrix making all the diagonal entries 1. This represents a loop at each node which gives a path for a node's state to influence itself in the next layer. Next, we multiply on both sides by a diagonal matrix containing the reciprocal square root of each node's degree, its number of neighbors. This serves to normalize the states of the nodes otherwise well connected nodes could have states that are much larger than isolated

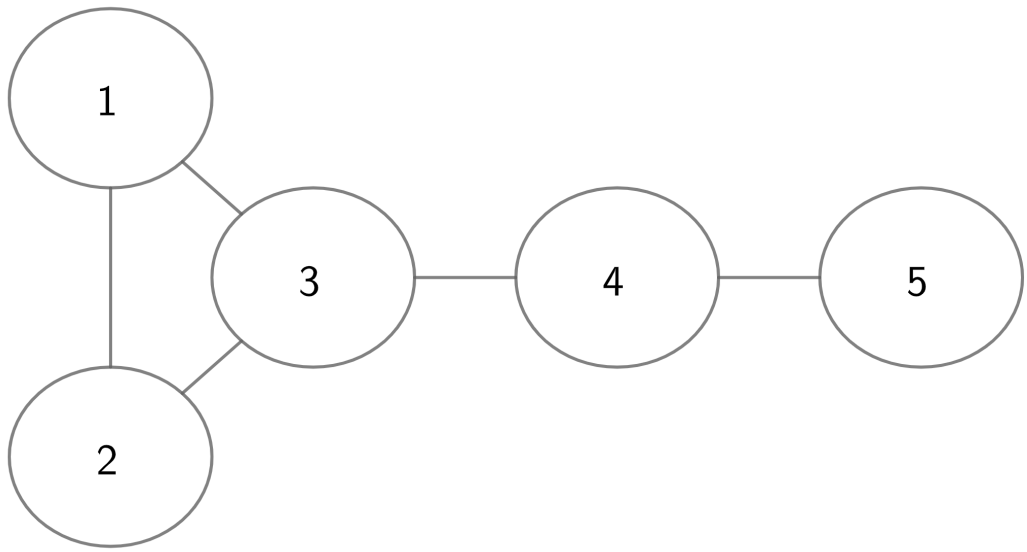


Figure 1.4: An Example of a Graph

nodes since they have so many neighbors contributing to their state. A single layer of GCN can only see information from a node's immediate neighbors, so finding trends from larger neighborhoods requires the use of multiple layers. After the first layer, the neighbors of our node will have been updated based on their neighbors, and so a second layer will provide our node with data from up to two steps away. To better see how this propagation works, we will go through an example of finding the output of a three layer GCN for node 3 of the graph in Figure 1.4

First we need our augmented adjacency matrix.

$$A = D\left(\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} + I\right) D$$

Where

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \text{ and } D = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{3}} & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{\sqrt{1}} \end{bmatrix}$$

To figure out which nodes are involved in the creation of our final state, we need to find every node within $(\tau - t)$ steps, where τ is the depth of the network and t

is the layer that we are look at. This gives us the following update scheme where $x_i^{(t)}$ is the state for node i at layer t with layer 0 being the initial input, α_{ij} is the degree factor from the D matrices, $(d_i d_j)^{-1/2}$, $w^{(t)}$ is the weight matrix for layer t, and $x_i^{(t)} = \phi(y_i^{(t)})$

Layer 1

$y_1^{(1)} = \left(\alpha_{11}x_1^{(0)} + \alpha_{12}x_2^{(0)} + \alpha_{13}x_3^{(0)}\right) W^{(1)}$

$y_2^{(1)} = \left(\alpha_{21}x_1^{(0)} + \alpha_{22}x_2^{(0)} + \alpha_{23}x_3^{(0)}\right) W^{(1)}$

$y_3^{(1)} = \left(\alpha_{31}x_1^{(0)} + \alpha_{32}x_2^{(0)} + \alpha_{33}x_3^{(0)} + \alpha_{34}x_4^{(0)}\right) W^{(1)}$

$y_4^{(1)} = \left(\alpha_{43}x_3^{(0)} + \alpha_{44}x_4^{(0)} + \alpha_{45}x_5^{(0)}\right) W^{(1)}$

$y_5^{(1)} = \left(\alpha_{54}x_4^{(0)} + \alpha_{55}x_5^{(0)}\right) W^{(1)}$

Layer 2

$y_1^{(2)} = \left(\alpha_{11}x_1^{(1)} + \alpha_{12}x_2^{(1)} + \alpha_{13}x_3^{(1)}\right) W^{(2)}$

$y_2^{(2)} = \left(\alpha_{21}x_1^{(1)} + \alpha_{22}x_2^{(1)} + \alpha_{23}x_3^{(1)}\right) W^{(2)}$

$y_3^{(2)} = \left(\alpha_{31}x_1^{(1)} + \alpha_{32}x_2^{(1)} + \alpha_{33}x_3^{(1)} + \alpha_{34}x_4^{(1)}\right) W^{(2)}$

$y_4^{(2)} = \left(\alpha_{43}x_3^{(1)} + \alpha_{44}x_4^{(1)} + \alpha_{45}x_5^{(1)}\right) W^{(2)}$

Layer 3

$y_3^{(3)} = \left(\alpha_{31}x_1^{(2)} + \alpha_{32}x_2^{(2)} + \alpha_{33}x_3^{(2)} + \alpha_{34}x_4^{(2)}\right) W^{(3)}$

When it comes to back propagation, we are in a similar position as with the recurrent neural networks. The final layer is straightforward, but as we go to earlier layers, we need to take into account the various paths that our nodes can take to reach the final node. For Node 1 in our example, we have three different ways to reach Node 3 in the two steps we take from Layer 1: we can stay at 1 for a step then move to 3, we can move to 3 then stay there for a step, and we can move to 2 and then to 3. So in a complete expansion of the gradient of our loss function with respect to $W^{(1)}$ we have three terms coming just from Node 1. Like in the case of the recurrent neural networks, we can use the chain rule to clear up our notation. So all the terms for Node 1 can be included in one term, and we can do so for every node included in that layer. With that we can express the gradient for the weight at a given layer as

$$\frac{\partial \mathcal{L}}{\partial W^{(i)}} = \frac{1}{|G|} \sum_{i \in G} \sum_{n \in \mathcal{N}_{\tau-t}(i)} \frac{\partial \mathcal{L}}{\partial y_n^{(t)}} \frac{\partial y_n^{(t)}}{\partial W^{(t)}} \tag{1.1}$$

where $\mathcal{N}_{\tau-t}(i)$ is the neighborhood around node i, and $|G|$ is the number of vertices in the graph G.

In practice, many of the graphs are too large to be processed at once, so we use mini-batching like in the previously discussed neural networks. Where this differs from the previous examples is that mini-batching may still not be enough to avoid needing too much of the graph. For even a small mini-batch and a small number of layers, if the graph is well-connected enough, a few-step neighborhood around our

example points can still reach a huge area. To avoid this, we use neighborhood sampling. This picks a specified number of random neighbors at each layer, preventing the selection from growing too large. With these in place, we can work on suitably sized subgraphs, preventing our potential memory issues. These subgraphs are generated by going backwards through the model, starting with the output node and selecting its sampled neighbors, then sampling neighbors for the selected nodes and so on until we reach the beginning of the model.

1.6 Normalization

An additional aspect of many deep learning models is the inclusion of normalization modules in their architectures. Two of the most common of these are Batch Normalization (BN) and Layer Normalization (LN). The idea behind these normalization techniques is to attempt to fix the problem of internal covariate shift. This is the phenomenon where the distribution of the layer inputs constantly changes during training. When a layer's parameters are updated during an optimization step, it is doing so based on its performance with inputs given to it with a certain mean and variance. However, on the next forward pass, the data it receives from the previous layers may now be distributed rather differently since the previous layers have also had their parameters changed. This lack of a stable base from which to learn can make the training process much more difficult, slowing down convergence or even preventing the model from converging at all.

Batch Normalization was proposed in [Ioffe and Szegedy, 2015a] as a way to combat internal covariate shift and to speed up training. To do so, BN takes the mean and variance over a particular mini-batch and uses them to recenter and rescale the inputs to have a mini-batch norm of zero and variance of one. These statistics can be too strict of a requirement to properly represent the underlying data distribution, so learnable gain and bias parameters, γ and β, are often used to allow more flexibility. Details of this can be seen in Algorithm 1.

Algorithm 1 Batch Normalization applied to a mini-batch of inputs, x

Input: Values of x over a mini-batch $\{x_1, x_2, \ldots, x_n\}$
A preset ε value to prevent division by zero
Outputs: Normalized values of y for each element of the mini-batch $\{y_1, y_2, \ldots, y_n\}$

$\mu_{\mathcal{B}} \leftarrow \frac{1}{n}\sum_{i=1}^{n} x_i$ ▷ Collection of batch mean

$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{n}\sum_{i=1}^{n} (x_i - \mu_{\mathcal{B}})^2$ ▷ Collection of batch variance

$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \varepsilon}}$ ▷ Normalization

$y_i \leftarrow \gamma \hat{x}_i + \beta$ ▷ Rescaling and Recentering by learnable parameters

While Batch Normalization was proposed as a remedy for internal covariate shift, it has been difficult to ascertain whether or not its success is actually from doing so

or whether it assists in model performance in other ways. [Santurkar et al., 2018] argues that batch normalization actually has little effect on internal covariate shift, but instead acts to smooth the optimization landscape. For many problems, the loss surface can be non-convex and difficult to optimize, but the addition of batch normalization can remedy that which provides some explanation for why it would speed up convergence. In addition to the argument about internal covariate shift, [Ioffe and Szegedy, 2015a] also point to batch normalization's role as a regularizer as explanation of its behavior. By linking examples of the mini-batch together, the output of the model for a particular input is no longer a deterministic problem. Instead it can now depend on the random shuffling of the mini-batches. This helps prevent overfitting and leads to improvements in generalization.

Although Batch Normalization has become standard in many deep learning models, there are a number of weaknesses that it suffers from. One of these is a generally decreased performance with smaller batch sizes. Since the statistics used in the normalization steps come from the batches, the selection of the mini-batches can have a huge impact on the efficacy of the method, especially when using smaller batch sizes. If outliers are present in a batch, they can introduce a considerable amount of noise to the batch statistics unless there are sufficiently many normal examples included to balance it out. While batch normalization is typically an inexpensive process, the need for large batch sizes can require prohibitively large memory costs. Batch Norm also suffers when it comes to adapting to sequential problems. In the inference step of training, the model uses estimated population statistics instead of those coming from the batch. This works well in models of specified lengths, but for variable length problems like those RNNs are designed to model, we face the problem of having no information to work with if we are given a test example which is longer than anything in our training set.

Layer Normalization, proposed in [Ba et al., 2016], attempts to avoid these problems by using statistics of a given state instead of those of a training batch. LN achieves this by taking the mean and variance over the hidden units of the state for a single training example, as can be seen in Algorithm 2.

Algorithm 2 Layer Normalization applied to an input, x, of hidden size H

Input: An input vector $x = [x_1, x_2, \ldots, x_H]^T$
A preset ε value to prevent division by zero
Outputs: A normalized vector, y

$\mu_x \leftarrow \frac{1}{H}\sum_{i=1}^{H} x_i$ ▷ Collection of mean

$\sigma_{\S}^2 \leftarrow \frac{1}{H}\sum_{i=1}^{H}(x_i - \mu_{\S})^2$ ▷ Collection of variance

$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \varepsilon}}$ ▷ Normalization

$y_i \leftarrow \gamma\hat{x}_i + \beta$ ▷ Rescaling and Recentering by learnable parameters

Since this calculation does not involve any other elements of the mini-batch, LN

is not susceptible to changes in batch size and can work even with each example fed through individually. This has the benefit of limiting the impact of outliers on training as well as requiring less precise tuning of the batch size hyperparameter. Also, Layer Normalization is better suited for RNNs since there is no need for population statistics when it comes to generalization. Since every example normalizes using its own mean and variance, we still have the necessary statistics regardless of how long our test examples are. Some implementations of Layer Normalization include learnable gain and bias parameters which allows for some information to be included across examples and through the sequence dimension, though [Xu et al., 2019] has argued that these parameters are unnecessary.

Copyright© Cole M. Pospisil, 2023.

Chapter 2 Batch Normalized Preconditioning (BNP)

In this chapter, we will develop a preconditioning method for Recurrent Neural Networks and Graph Convolutional Networks.

2.1 Preconditioned Gradient Descent

The primary method by which neural networks are trained is gradient descent. Because of this, it is important to have gradients and Hessians with desirable matrix properties such as condition number. One common way to promote this is through the use of a preconditioning algorithm. Supposing that we have a trainable parameter vector θ, learning rate α, and a twice differentiable loss function $\mathcal{L}$, the standard gradient descent update step is

$$\theta_{k+1} = \theta_k - \alpha \frac{\partial \mathcal{L}}{\partial \theta}. \tag{2.1}$$

Since we are seeking to converge to a local minimum of $\mathcal{L}$, it is reasonable to look at the convergence behavior as our approximations approach such a local minimizer, θ^*. By being a minimizer, we have that $\nabla \mathcal{L}(\theta^*) = 0$ and the Hessian matrix, $\nabla^2 \mathcal{L}(\theta^*)$ is symmetric positive semi-definite. Letting $\lambda_{\min}$ and $\lambda_{\max}$ be the minimum and maximum eigenvalues of $\nabla^2 \mathcal{L}(\theta^*)$ respectively and assuming that $\lambda_{\min} > 0$, we have that for any $\varepsilon > 0$ there is a small neighborhood around θ^* where we have

$$\|\theta_{k+1} - \theta^*\| \leq (r + \varepsilon)\|\theta_k - \theta^*\| \tag{2.2}$$

for any initial approximation θ_0 within that neighborhood and $L2$ matrix norm $\|\cdot\|$, with $r = \frac{\kappa - 1}{\kappa + 1}$ where κ is the condition number of the Hessian. This expression is a result of the Taylor series approximation of $\nabla \mathcal{L}$ about the point θ^*:

$$\begin{aligned} \theta_{k+1} &= \theta_k - \alpha \nabla \mathcal{L}(\theta_k) && (2.3) \\ &= \theta^* + (I - \alpha \nabla^2_\theta \mathcal{L}(\xi))(\theta_k - \theta^*) && (2.4) \end{aligned}$$

for some ξ. By subtracting over the first term and applying the $L2$ norm to both sides, we get

$$\|\theta_{k+1} - \theta^*\|_2 \leq \|I - \alpha \nabla^2_\theta \mathcal{L}(\xi)\|_2 \|\theta_k - \theta^*\|_2. \tag{2.5}$$

Since we assumed that θ^* is a local minimizer and $\nabla^2 \mathcal{L}(\theta^*)$ is therefore symmetric positive semi-definite, we have that $I - \alpha \nabla^2_\theta \mathcal{L}(\xi)$ is real-valued and symmetric, so we have that equation 2.5 can be rewritten as

$$\|\theta_{k+1} - \theta^*\|_2 \leq (r + \varepsilon)\|\theta_k - \theta^*\|_2 \tag{2.6}$$

for $r = \max\{|1 - \alpha\lambda_{\min}|, |1 - \alpha\lambda_{\max}|\}$ [Saad, 2003, Polyak, 1964]. To get a contraction mapping that guarantees eventual convergence, we want to have $r < 1$, which means that we need $\alpha < 2/\lambda_{\max} = 2/\|\nabla^2_\theta \mathcal{L}(\theta^*)\|$. If we try to optimize r with respect to

α we get that $\alpha = \frac{2}{\lambda_{\min}+\lambda_{\max}}$. This occurs when we have either $r = 1 - \alpha\lambda_{\min}$ or $r = -1 + \alpha\lambda_{\min}$. To see that these in fact give us the same value for r, we need to check both cases. Firstly, we have

$$r = \left|1 - \frac{2\lambda_{\min}}{\lambda_{\max} + \lambda_{\min}}\right| \tag{2.7}$$

$$= \left|\frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\max} + \lambda_{\min}}\right| \tag{2.8}$$

$$= \left|\frac{\frac{\lambda_{\max}}{\lambda_{\min}-1}}{\frac{\lambda_{\max}}{\lambda_{\min}} + 1}\right| \tag{2.9}$$

$$= \left|\frac{\kappa - 1}{\kappa + 1}\right| \tag{2.10}$$

In the second case, we use similar steps to see that $r = \left|\frac{1-\kappa}{\kappa+1}\right| = \left|\frac{\kappa-1}{\kappa+1}\right|$. These calculations show that the optimal convergence rate for our gradient descent method is

$$r = \frac{\kappa - 1}{\kappa + 1} \tag{2.11}$$

where κ is the condition number of the Hessian in the $L2$ matrix norm,

$$\kappa = \kappa(\nabla^2\mathcal{L}(\theta^*)) = \frac{\lambda_{\max}}{\lambda_{\min}}.$$

Now that we have a clear picture on the relationship between training a model and the condition number of the Hessian, we can attempt to improve the rate of convergence through a manipulation of the Hessian. Since the calculation of the Hessian matrix can be expensive and is not necessary for training, we achieve this manipulation through the application of a *preconditioning* transformation on the loss gradient. This is done through a change of variables for our trainable parameter, giving us $\theta = Pz$ for some z. This changes the gradient descent step from Equation 2.1 to

$$z_{k+1} = z_k - \alpha\nabla_z\mathcal{L}(Pz_k) = z_k - \alpha P^T\nabla_\theta\mathcal{L}(\theta_k) \tag{2.12}$$

We can similarly rewrite equation 2.2 as

$$\|z_{k+1} - z^*\| \leq (r + \varepsilon)\|z_k - z^*\| \tag{2.13}$$

where our Hessian with respect to z of $\mathcal{L}$ is given by

$$\nabla_z^2\mathcal{L}(Pz^*) = P^T\nabla_\theta^2\mathcal{L}(\theta^*)P.$$

If we can achieve a better condition number for $P^T\nabla_\theta^2\mathcal{L}(\theta^*)P$ than we have for $\nabla_\theta^2\mathcal{L}(\theta^*)$, then we have a reduced r which should guarantee faster convergence from Equations 2.2 and 2.13. In this case, we have an updated gradient descent step obtained from multiplying both sides of Equation 2.12 by P:

$$\theta_{k+1} = \theta_k - \alpha PP^T\nabla_\theta\mathcal{L}(\theta_k) \tag{2.14}$$

Through the use of the preconditioner, PP^T, we get the benefits of the improved conditioning on the Hessian through the much cheaper process of adapting the gradients used in the gradient descent step.

While these convergence properties assume strong local convexity at the local minimum, we are not guaranteed that this holds in our setting of training neural networks. In this case, we have an adapted result

Theorem 1 ([Lange et al., 2021])**.** *Consider a loss function $\mathcal{L}$ with continuous third order derivatives and with a positively scale-invariant property. If $\theta = \theta^*(t)$ is a positively scale-invariant manifold at local minimizer θ^*, then the null space of the Hessian $\nabla^2\mathcal{L}(\theta^*(t))$ contains at least the column space $Col(D_t\theta^*(t))$. Furthermore, for the gradient descent iteration, $\theta_{k+1} = \theta_k k - \alpha\nabla\mathcal{L}(\theta_k)$, let θ_k^* be the local minimizer closest to θ_k, i.e. $\theta_k^* = \theta^*(t_k)$ where $t_k = argmin_{t>0}\|\theta_k - \theta^*(t)\|$ and assume that the null space of $\nabla^2\mathcal{L}(\theta^*(t))$ is equal to $Col(D_t\theta^*(t))$. Then, for any $\varepsilon > 0m$ if our initial approximation θ_0 is such that $\|\theta_0 - \theta_0^*\|$ is sufficiently small, then we have, for all $k \geq 0$,*

$$\|\theta_{k+1} - \theta_{k+1}^*\| \leq (r + \varepsilon)\|\theta_k - \theta_k^*\|,$$

where $r = \max\{|1 - \alpha\lambda_{\min}^*|, |1 - \alpha\lambda_{\max}|\}$ and $\lambda_{\min}^*$ and $\lambda_{\max}$ are the smallest nonzero eigenvalue and largest eigenvalue of $\nabla^2\mathcal{L}(\theta_k^*)$, respectively.

2.2 BNP

Batch Normalized Preconditioning (BNP) is a technique that was developed for fully connected and convolutional neural networks in [Lange et al., 2021]. BNP is a preconditioning method where the Hessian is expressed in connection to the hidden state activations of a mini-batch, and those connections are used to create a preconditioner to improve the condition number of the Hessian. They start with a rewritten version of the neural network forward step, $h^{(l)} = g(W^{(l)}h^{(l-1)} + b^{(l)})$. They change this by considering vectors of the augmented weight and hidden state matrices,

$$\hat{w}^T = \begin{bmatrix} b_i^{(l)} & w_i^{(l)T} \end{bmatrix} \text{ and } \hat{h} = \begin{bmatrix} 1 \\ h^{(l-1)} \end{bmatrix}$$

and consider the the ith entry of the preactivation state, $a^{(l)}$, given by $a_i^{(l)} = \hat{w}^T\hat{h}$. With this set up they create a parameterization of the Hessian as follows:

Proposition 1 ([Lange et al., 2021])**.** *Consider a loss function L defined from the output of a fully connected multi-layer neural network for a single network input x. Consider the weight and bias parameters $w_i^{(l)}, b_i^{(l)}$ at layer l, and write $L = L\left(a_i^{(l)}\right) = L(\hat{w}^T\hat{h})$ as a function of the parameter $\hat{w}$ through $a_i^{(l)}$ as above. When training over a mini-batch of N inputs, let $\{h_1^{(l-1)}, h_2^{(l-1)}, \ldots, h_N^{(l-1)}\}$ be the associated hidden states, $h^{(l-1)}$ and let $\hat{h}_j = \begin{bmatrix} 1 \\ h_j^{(l-1)} \end{bmatrix} \in \mathbb{R}^{(n_{l-1}+1)\times 1}$. Let*

$$\mathcal{L} = \mathcal{L}(\hat{w}) := \frac{1}{N}\sum_{j=1}^{N} L(\hat{w}^T\hat{h}_j)$$

be the mean loss over the mini-batch. Then, its Hessian with respect to $\hat{w}$ is

$$\nabla^2_{\hat{w}}\mathcal{L}(\hat{w}) = \hat{H}^T S \hat{H}$$

where $\hat{H} = \begin{bmatrix} e & H \end{bmatrix}$ *and*

$$H = \begin{bmatrix} h_1^{(l-1)T} \\ \vdots \\ h_N^{(l-1)T} \end{bmatrix} \text{ and } S = \frac{1}{N}\begin{bmatrix} L''(\hat{w}^T\hat{h}_1) & & \\ & \ddots & \\ & & L''(\hat{w}^T\hat{h}_N) \end{bmatrix}$$

with all off-diagonal elements of S equal to 0, and e the vector of all ones.

This gives a situation where they can relate the condition number of the Hessian to the condition numbers of the $\hat{H}$ and S matrices, through

Proposition 2 ([Lange et al., 2021])**.** *Let S be an $m \times m$ symmetric positive definite matrix and $\hat{H}$ an $m \times n$ matrix. Then we have $\kappa(\hat{H}^T S \hat{H}) \leq \kappa(\hat{H})^2 \kappa(S)$.*

They then create a preconditioner to improve the condition number of $\hat{H}$ as follows:

$$\hat{w} = Pz, \text{ with } P := UD \text{ where } U := \begin{bmatrix} 1 & -\mu_H^T \\ 0 & I \end{bmatrix} \text{ and } D := \begin{bmatrix} 1 & 0 \\ 0 & \text{diag}(\sigma_H) \end{bmatrix}^{-1} \tag{2.15}$$

where

$$\mu_H := \frac{1}{N} H^T e = \frac{1}{N}\sum_{j=1}^{N} h_j^{(l-1)}, \text{ and } \sigma_H^2 := \frac{1}{N}\sum_{j=1}^{N}(h_j^{(l-1)} - \mu_H)^2 \tag{2.16}$$

Here U recenters the hidden states to have zero mean and D rescales them to have unit variance. This improves the condition number in two ways. Using

$$\hat{H}U = \begin{bmatrix} e & H - e\mu_H^T \end{bmatrix} \text{ and } (H - e\mu_H^T)^T e = 0$$

multiplication of $\hat{H}$ by U makes the first column orthogonal to the rest, and multiplying $H - e\mu_H^T$ by D scales all the columns of $H - e\mu_H^T$ to have the same norm. This allows the use of the following theorem to show improved condition number:

Theorem 2 ([Lange et al., 2021])**.** *Let $\hat{H} = [e, H]$ be the extended hidden variable matrix, U the centering transformation matrix, and D the variance normalizing matrix as in 2.15, Assume $\hat{H}$ has full column rank. We have*

$$\kappa(\hat{H}U) \leq \kappa(\hat{H}).$$

This inequality is strict if $\mu_H \neq 0$ and is not orthogonal to $x_{\max}$ where $x_{\max}$ is an eigenvector corresponding to the largest eigenvalue of the sample covariance matrix $\frac{1}{N-1}(H - e\mu_H^T)^T(H - e\mu_H^T)$. Moreover,

$$\kappa(\hat{H}UD) \leq \sqrt{n_{l-1} + 1} \min_{D_0 \text{ is diagonal}} \kappa(\hat{H}UD_0)$$

The batch normalization preconditioning step is summarized as

Algorithm 3 One Step of BNP Training on $W^{(l)}, b^{(l)}$ of the lth Dense Layer

Given: $\varepsilon_1 = 10^{-2}, \varepsilon_2 = 10^{-4}$ and $\rho = 0.99$; learning rate α; $\mu = 0$, $\sigma = 1$
Input: Mini-batch output of previous layer $H = [h_1^{(l-1)}, h_2^{(l-1)}, \ldots, h_N^{(l-1)}]^T \subset \mathbb{R}^{n_{l-1}}$ and parameters gradients: $G_w \leftarrow \frac{\partial \mathcal{L}}{\partial W^{(l)}}, G_b \leftarrow \frac{\partial \mathcal{L}}{\partial b^{(l)}}$
Compute mini-batch mean/variance: μ_H, σ_H^2;
Compute running average statistics: $\mu \leftarrow \rho\mu + (1-\rho)\mu_H, \sigma^2 \leftarrow \rho\sigma^2 + (1-\rho)\sigma_H^2$;
Set $\tilde{\sigma}^2 = \sigma^2 + \varepsilon_1 \max\{\sigma^2\} + \varepsilon_2$ and $q^2 = \max\{n_{l-1}/N, 1\}$;
Update G_w: $G_w(i,j) \leftarrow \dfrac{1}{q^2}\dfrac{G_w(i,j) - \mu(j)G_b(i)}{\tilde{\sigma}^2(j)}$;
Update G_b: $G_b(i) \leftarrow \dfrac{1}{q^2}G_b(i) - \displaystyle\sum_j G_w(i,j)\mu(j)$;
Output: Preconditioned gradients G_w, G_b.

2.3 BNP for LSTM

In this section, we develop a batch normalized preconditioning method for RNNs. This method will leverage the convergence results from [Lange et al., 2021] to this new architecture. We will also show empirical results supporting the method's claims.

Consider a recurrent neural network (RNN) in which the tth time step is defined by

$$h^{(t)} = \sigma(Wh^{(t-1)} + Ux^{(t)} + b) \tag{2.17}$$

where σ is an elementwise nonlinearity, $x^{(t)}$ is the tth element of a sequential input $\underline{x}$, W, U being weight matrices and b being bias. During training, let $\{\underline{x}_1, \underline{x}_2, \ldots, \underline{x}_N\}$ be a minibatch consisting of N sequential examples and let the corresponding hidden states be $H = \{h_1^{(0)}, h_1^{(1)}, \ldots, h_1^{(\tau)}, h_2^{(0)}, \ldots, h_N^{(\tau)}\}$.
We denote $h_i^{(t)} = \sigma(a_i^{(t)})$ as the i-th entry of $h^{(t)}$, where $a_i^{(t)} = w_i^T h^{(t-1)} + u_i^T x^{(t)} + b_i \in \mathbb{R}$. Here $w_i^T \in \mathbb{R}^{1\times m}$, $u_i^T \in \mathbb{R}^{1\times n}$, and $b_i \in \mathbb{R}$ are the ith row of W, U and b respectively, m is the dimension of $h^{(t)}$ and n is the dimension of $x^{(t)}$. Let

$$\hat{w}^T = \begin{bmatrix} b_i & w_i^T & u_i^T \end{bmatrix} \in \mathbb{R}^{1\times(m+n+1)}, \hat{h}^{(t)T} = \begin{bmatrix} 1 & h^{(t-1)T} & x^{(t)T} \end{bmatrix} \in \mathbb{R}^{1\times(m+n+1)}$$

then we have that $a_i^{(t)} = \hat{w}^T \hat{h}^{(t)}$

Proposition 3. *Consider a recurrent neural network loss function L defined for a single sequential network input $\underline{x}$. Write $L = L(a_i) = L(\hat{w}_i{}^T \hat{h})$ as a function of the corresponding a and hence of the parameter $\hat{w}$. When training over a mini-batch of N inputs, let $\{h_1^{(t-1)}, h_2^{(t-1)}, \ldots, h_N^{(t-1)}\}$ be the associated hidden states for each example, $\{x_1^{(t)}, x_2^{(t)}, \ldots, x_N^{(t)}\}$ the associated inputs, and let $\hat{h}_j^{(t)T} = \begin{bmatrix} 1 & h_j^{(t-1)T} & x_j^{(t)T} \end{bmatrix} \in \mathbb{R}^{(1\times(m+n+1)}$. Let $L = L(\hat{w}) := \frac{1}{N}\sum_{j=1}^N \sum_{l=0}^{\tau} L(\hat{w}_i^T \hat{h}^{(t)})$ be the mean loss over the mini-batch. Then, its Hessian with respect to $\hat{w}$ is*

$$\begin{aligned} \nabla_{\hat{w}}^2 L(\hat{w}) = \hat{H}^T S \hat{H}, \text{ where } \hat{H}^T = \begin{bmatrix} \hat{h}_1^{(0)} & \hat{h}_1^{(1)} & \ldots & \hat{h}_1^{(\tau)} & \hat{h}_2^{(0)} & \ldots & \hat{h}_N^{(\tau)} \end{bmatrix} \\ \text{and } S = \frac{1}{N} diag(\{S_k\}_{k=1}^N) \text{where } S_k = \left[\frac{\partial^2 L}{\partial a_i^{(col)} \partial a_i^{(row)}}\right]_{row,col=0}^{\tau} \end{aligned} \tag{2.18}$$

We propose to use the following preconditioning transformation:

$$\hat{w} = Pz, \text{ where } P := UD,\ U = \begin{bmatrix} 1 & -\mu_h^T & -\mu_x^T \\ 0 & I & 0 \\ 0 & 0 & I \end{bmatrix}, D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \text{diag}(\sigma_h) & 0 \\ 0 & 0 & \text{diag}(\sigma_x) \end{bmatrix}^{-1} \tag{2.19}$$

where $\mu_h := \frac{1}{N(\tau+1)} \sum_{i=1}^{N} \sum_{j=0}^{\tau} h_i^{(j-1)}$ and $\mu_x := \frac{1}{N(\tau+1)} \sum_{i=1}^{N} \sum_{j=0}^{\tau} x_i^{(j)}$ are the means of the batch's hidden states and inputs respectively, taken across the batch and across time, and σ_h^2 and σ_x^2 are the variances. The preconditioned Hessian matrix is $P^T \nabla_{\hat{w}}^2 L P = (\hat{H}P)^T S (\hat{H}P)$.

We will now confirm that P satisfies the criteria for the centering and variance normalization transformations to be able to use Theorem 2, and thereby show that P improves the conditioning of $\hat{H}$ and therefore of $\nabla_{\hat{w}}^2 L$.

We can write $\hat{H}$ as $\begin{bmatrix} e & H^T & X^T \end{bmatrix}$ where

$$H = \begin{bmatrix} h_1^{(-1)} & h_1^{(0)} & \dots & h_1^{(\tau-1)} & h_2^{(-1)} & \dots & h_N^{(\tau-1)} \end{bmatrix}$$
$$X = \begin{bmatrix} x_1^{(0)} & x_1^{(1)} & \dots & x_1^{(\tau)} & x_2^{(0)} & \dots & x_N^{(\tau)}, \end{bmatrix}$$

and e is the vector of all ones. Then we can see that $\mu_h = \frac{1}{N\cdot(\tau+1)} He, \mu_x = \frac{1}{N\cdot(\tau+1)} Xe$, $\hat{H}U = \begin{bmatrix} e & H^T - e\mu_h^T & X^T - e\mu_x^T \end{bmatrix}$ and $(H^T - e\mu_h^T)^T e = 0$, $(X^T - e\mu_x)^T e = 0$. So multiplying $\hat{H}$ by U makes the first column orthogonal to the rest of the matrix.

Corollary 1. *Let $\hat{H}$ be defined as in Proposition 3, and U and D be defined as in Equation 2.19*

Then we have

$$\kappa(\hat{H}U) \leq \kappa(\hat{H}) \text{ and } \kappa(\hat{H}UD) \leq \sqrt{m+1} \min_{D_0 \text{ is diagonal}} \kappa(\hat{H}UD_0) \tag{2.20}$$

Implementation

While the derivation of the batch normalization preconditioning method uses a generic RNN as its basis, our experimental results will all be using LSTM as their base model. Recall the architecture of an LSTM cell:

$$\begin{bmatrix} \mathbf{f}^{(t)} \\ \mathbf{i}^{(t)} \\ \mathbf{o}^{(t)} \\ \mathbf{g}^{(t)} \end{bmatrix} = \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)} + \mathbf{b} \tag{2.21}$$

$$\mathbf{c}^{(t)} = \sigma(\mathbf{f}^{(t)}) \odot \mathbf{c}^{(t-1)} + \sigma(\mathbf{i}^{(t)}) \odot \tanh\left(\mathbf{g}^{(t)}\right) \tag{2.22}$$

$$\mathbf{h}^{(t)} = \sigma(\mathbf{o}^{(t)}) \odot \tanh\left(\mathbf{c}^{(t)}\right) \tag{2.23}$$

Where $\mathbf{W} \in \mathbb{R}^{4d_h \times d_h}$, $\mathbf{U} \in \mathbb{R}^{4d_h \times d_x}$ and $\mathbf{b} \in \mathbb{R}^{4d_h}$

The gating mechanism for the LSTM acts as the nonlinearity, g in our derivation, so the adaptation from a basic RNN cell to an LSTM cell is straightforward. By

basing our models in LSTM, which is one of the most popular sequence task models, we are more able to accurately compare our results with the relevant work in the field.

The main comparisons against which we will present our model's performance are the vanilla LSTM cell and two normalized LSTM models: a recurrent batch normalization model [Cooijmans et al., 2017] and a recurrent layer normalization model [Ba et al., 2016].

Recall that batch normalization is a network reparameterization designed to prevent internal covariate shift by using estimates of parameter's means and standard deviations. A batch normalization transformation is as follows:

$$\mathrm{BN}(\mathbf{h};\gamma,\beta) = \beta + \gamma \odot \frac{\mathbf{h} - \mu(h)}{\sqrt{\sigma^2(h) + \varepsilon}} \tag{2.24}$$

The statistics μ and σ are generally estimated from the minibatch statistics for training, and the training population for validation and testing. [Cooijmans et al., 2017] proposed the following architecture for adapting batch normalization to LSTMs

$$\begin{bmatrix} \mathbf{f}^{(t)} \\ \mathbf{i}^{(t)} \\ \mathbf{o}^{(t)} \\ \mathbf{g}^{(t)} \end{bmatrix} = \mathrm{BN}(\mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{b_h}) + \mathrm{BN}(\mathbf{U}\mathbf{x}^{(t)} + \mathbf{b_x}) \tag{2.25}$$

$$\mathbf{c}^{(t)} = \sigma(\mathbf{f}^{(t)}) \odot \mathbf{c}^{(t-1)} + \sigma(\mathbf{i}^{(t)}) \odot \tanh\,(\mathbf{g}^{(t)}) \tag{2.26}$$

$$\mathbf{h}^{(t)} = \sigma(\mathbf{o}^{(t)}) \odot \tanh\,(\mathrm{BN}(\mathbf{c}^{(t)})) \tag{2.27}$$

Layer Normalization behaves similarly to Batch Normalization, but differs in that the statistics are determined over the hidden units in a particular layer instead of over a given minibatch. [Ba et al., 2016] proposed the following architecture for adapting layer normalization to LSTMs

$$\begin{bmatrix} \mathbf{f}^{(t)} \\ \mathbf{i}^{(t)} \\ \mathbf{o}^{(t)} \\ \mathbf{g}^{(t)} \end{bmatrix} = \mathrm{LN}(\mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{b_h}) + \mathrm{LN}(\mathbf{U}\mathbf{x}^{(t)} + \mathbf{b_x}) \tag{2.28}$$

$$\mathbf{c}^{(t)} = \sigma(\mathbf{f}^{(t)}) \odot \mathbf{c}^{(t-1)} + \sigma(\mathbf{i}^{(t)}) \odot \tanh\,(\mathbf{g}^{(t)}) \tag{2.29}$$

$$\mathbf{h}^{(t)} = \sigma(\mathbf{o}^{(t)}) \odot \tanh\,(\mathrm{LN}(\mathbf{c}^{(t)})) \tag{2.30}$$

Our proposed architecture for a Recurrent BNP model is an adaptation of the Layer Normalization model with a number of changes. In the forward direction, we use layer normalization on the input and hidden terms of the preactivation state, but omit the normalization on the memory cell from Equation 2.30. The inclusion of the normalization in the forward pass is important because it addresses a weakness in our BNP method, namely that it only controls model behavior in the backward propagation step, so it is susceptible to exploding or vanishing states in the forward pass. The inclusion of the normalization on the input and hidden terms helps to

alleviate that problem. We found no notable benefit to the inclusion of the memory cell normalization, so it was removed to simplify computations.

$$\begin{bmatrix} \mathbf{f}^{(t)} \\ \mathbf{i}^{(t)} \\ \mathbf{o}^{(t)} \\ \mathbf{g}^{(t)} \end{bmatrix} = \mathrm{LN}(\mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{b_h}) + \mathrm{LN}(\mathbf{U}\mathbf{x}^{(t)} + \mathbf{b_x}) \tag{2.31}$$

$$\mathbf{c}^{(t)} = \sigma(\mathbf{f}^{(t)}) \odot \mathbf{c}^{(t-1)} + \sigma(\mathbf{i}^{(t)}) \odot \tanh\,(\mathbf{g}^{(t)}) \tag{2.32}$$

$$\mathbf{h}^{(t)} = \sigma(\mathbf{o}^{(t)}) \odot \tanh\,(\mathbf{c}^{(t)}) \tag{2.33}$$

In the backward propagation step, our model includes the derived batch normalization preconditioning step between gradient calculations and optimization. Our algorithm is slightly simplified from Algorithm 3 in that it does not use running statistics, includes a single epsilon term to modify the variance and does not use the q term to rescale the gradients.

Algorithm 4 One step of Recurrent BNP Training on W, U, b

Given: $\varepsilon = 10^{-3}$; learning rate α

Input: Mini-batch hidden states of the recurrent layer $H = \{h_1^{(0)}, h_1^{(1)}, \ldots, h_1^{(\tau)}, h_2^{(0)}, \ldots, h_N^{(\tau)}\} \subset \mathbb{R}^m$ and mini-batch inputs for the recurrent layer $X = \{x_1^{(0)}, \ldots, x_1^{(\tau)}, x_2^{(0)}, \ldots, x_N^{(\tau)}\} \subset \mathbb{R}^n$

Compute parameter gradients: $\dfrac{\partial L}{\partial W} \in \mathbb{R}^{m\times m}, \dfrac{\partial L}{\partial U} \in \mathbb{R}^{m\times n}, \dfrac{\partial L}{\partial b} \in \mathbb{R}^m$

$\mu_H \leftarrow \dfrac{1}{\tau \cdot N} \displaystyle\sum_{i=1}^{N}\sum_{j=1}^{\tau} h_i^{(j)}$ ▷ Compute Hidden State mean

$\mu_X \leftarrow \dfrac{1}{\tau \cdot N} \displaystyle\sum_{i=1}^{N}\sum_{j=1}^{\tau} x_i^{(j)}$ ▷ Compute Input mean

$\sigma_H^2 \leftarrow \dfrac{1}{\tau \cdot N} \displaystyle\sum_{i=1}^{N}\sum_{j=1}^{\tau} (h_i^{(j)} - \mu_H)^2 + \varepsilon$ ▷ Compute Hidden State variance

$\sigma_X^2 \leftarrow \dfrac{1}{\tau \cdot N} \displaystyle\sum_{i=1}^{N}\sum_{j=1}^{\tau} (x_i^{(j)} - \mu_X)^2 + \varepsilon$ ▷ Compute Input variance

$P \leftarrow \begin{bmatrix} 1 & -\mu_H^T & -\mu_X^T \\ 0 & I & 0 \\ 0 & 0 & I \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \mathrm{diag}(\frac{1}{\sigma_H}) & 0 \\ 0 & 0 & \mathrm{diag}(\frac{1}{\sigma_X}) \end{bmatrix}$ ▷ Compute Preconditioner

$\begin{bmatrix} b \\ W \\ U \end{bmatrix} \leftarrow \begin{bmatrix} b \\ W \\ U \end{bmatrix} - \alpha P P^T \begin{bmatrix} \frac{\partial L}{\partial b} \\ \frac{\partial L}{\partial W} \\ \frac{\partial L}{\partial U} \end{bmatrix}$ ▷ Update parameters

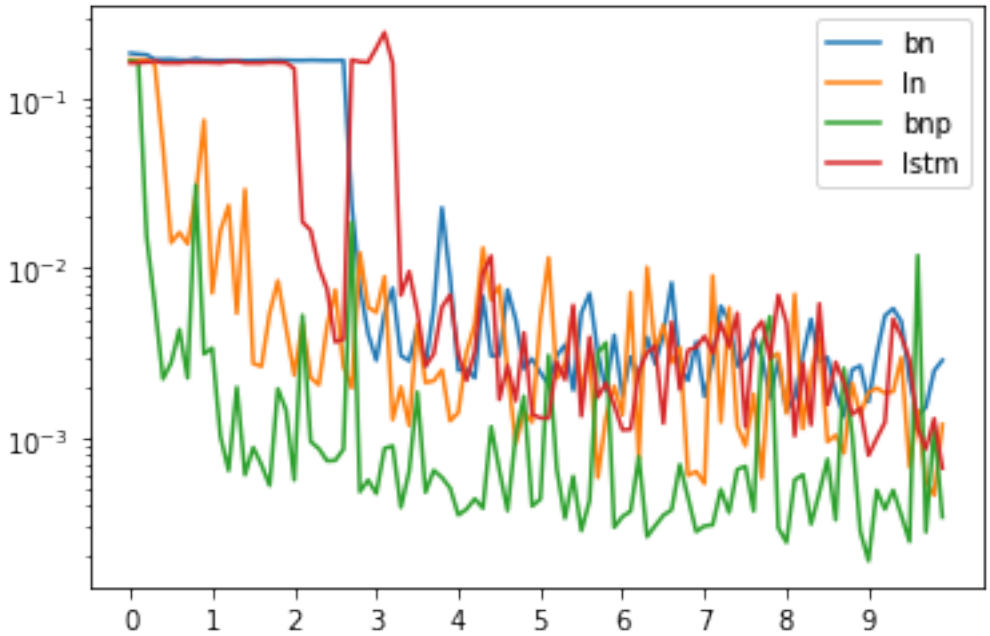


Figure 2.1: Results for Adding Problem: T = 400

Experiments

In this section we will present a number of experiments which provide empirical support for our assertions that the Recurrent BNP model achieves improved acceleration in training with no loss of performance.

Adding Problem: Our first experiment is the adding problem, which is a synthetic task for RNNs originally proposed in [Hochreiter and Schmidhuber, 1997]. In our implementation of this problem, the RNN takes a 2-dimensional input of length T. The first dimension contains a sequence of zeros except for two ones placed randomly in each half of the sequence. The second dimension is a sequence selected uniformly from $[0, 1)$. The goal of the task is to take the numbers from the second dimension in positions corresponding to the ones and to output their sum.

Implementation Details: The model was trained with a batch size of 50, a single LSTM layer with hidden size of 60 and RMSprop optimizer with learning rate of 10^{-3}. We used a T values of 400 for Figure 2.1 and 400 for Figure 2.2. This task is evaluated with mean-squared error.

Results: With $T = 400$ we see that our model shows immediate improvement over the base model as well as both normalized models. This initial acceleration brings the loss to below the best achieved by the other models within the first epoch, which is then maintained throughout the rest of training. For $T = 750$ we see that we have a similar acceleration boost over the LN and LSTM models, but that BN has similar behavior. Our model then stays at comparable losses to those seen by the BN and LSTM model, while LN is not able to converge for a sequence of this length.

Copying Problem: Our second experiment involves the copying problem, a common synthetic task that is used to test RNNs, which was originally proposed

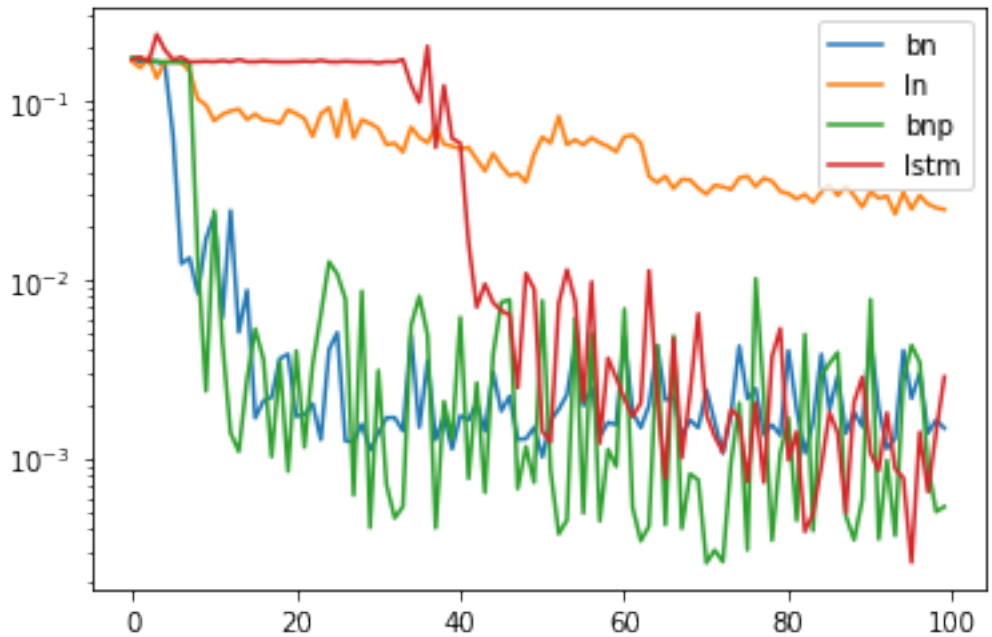


Figure 2.2: Results for Adding Problem: T = 750

in [Hochreiter and Schmidhuber, 1997]. For this problem, a string of 10 digits is fed into the RNN sampled uniformly from the integers between 1 and 8. A sequence of T zeros follows this, and a 9, marking the start of a string of 9 zeros, for a total length of $T + 20$. The objective of the task is to output the initial string of 10 digits beginning at the marker's location, copying the initial string from the front to the back. Cross-entropy loss is used to evaluate this model, with a baseline expected cross-entropy of $\frac{10 \log(8)}{T+20}$ which represents selecting digits 1-8 at random after the 9.

Implementation Details: The models were trained with a batch size of 128, a single LSTM layer with hidden size 68, and SGD optimizer with a learning rate of 10^{-4}. A T value of 1000 was used in Figure 2.3 and 2000 in Figure 2.4.

Results: For both of these experiments, we see that BNP achieves a quick acceleration boost, converging to the expected baseline value in a fraction of the number of iterations needed by the other models. Since BNP is primarily concerned with acceleration of training, the inability of the BNP model to pass the baseline value is not concerning.

2.4 BNP for Graph Convolutional Networks

In this section, we will develop a BNP style preconditioner for graph convolutional networks (GCN). Consider a GCN in which the tth layer is defined by

$$X^{(t)} = \phi\left(AX^{(t-1)}w + b^{(t)}\right) \tag{2.34}$$

Where ϕ is some nonlinear activation function, A is an adjacency matrix, and w is a trainable weight. For the purposes of our derivation, we will combine the weight and

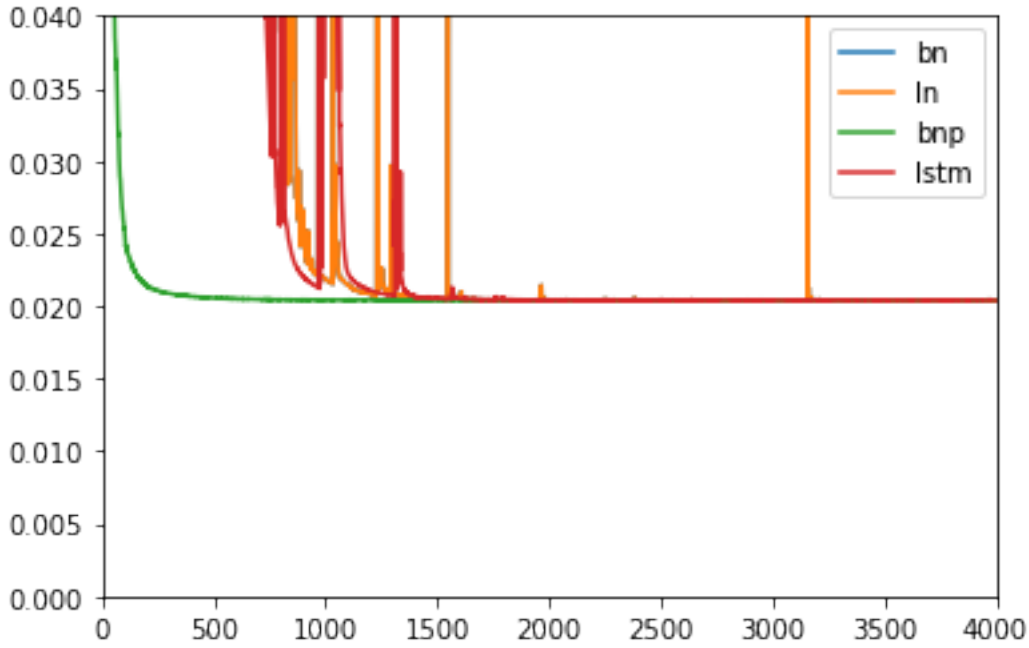


Figure 2.3: Results for Copying Problem: T = 1000

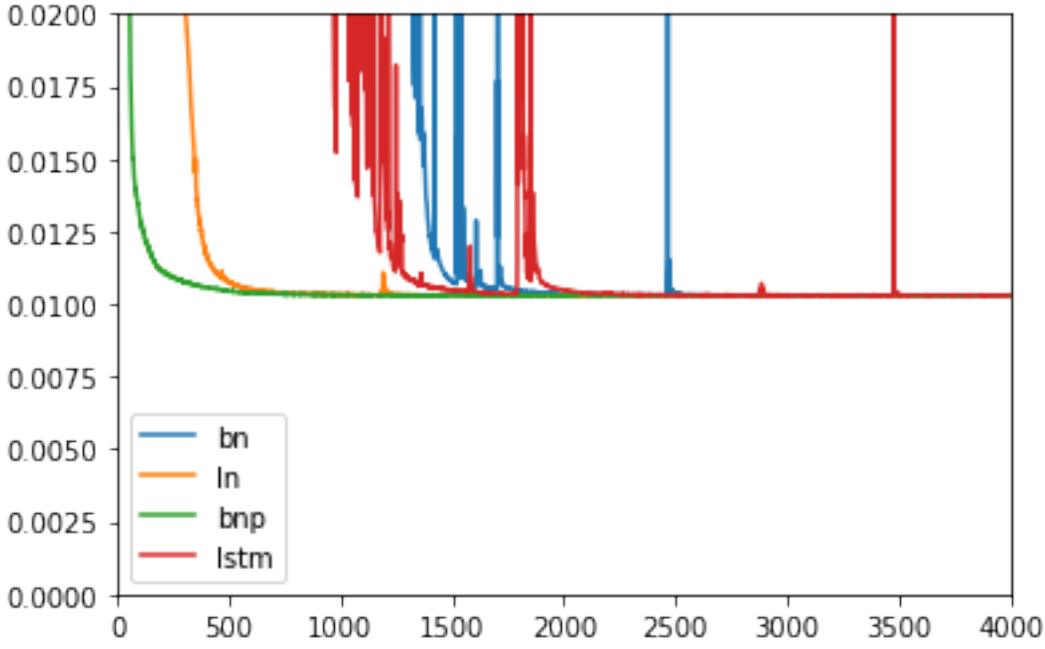


Figure 2.4: Results for Copying Problem: T = 2000

bias into a single augmented weight matrix, like was done for fully connected and recurrent versions. This gives us

$$z_{node}^{(layer)} = \sum_{v_i \in \mathcal{N}_1(v_{node})} \alpha_{node,i} x_i^{(layer-1)} \tag{2.35}$$

$$\hat{z}_{node}^{(layer)} = \begin{bmatrix} 1 \\ z_{node}^{(layer)} \end{bmatrix} \tag{2.36}$$

$$\hat{w}^{(layer)} = \begin{bmatrix} b^{(layer)} & w^{(layer)} \end{bmatrix} \tag{2.37}$$

With this set up, we can reformulate Equation 1.1 to give us a gradient of the loss function with respect to the augmented weight matrix in terms of the augmented hidden states and then do the same for the Hessian matrix.

Proposition 4. *Consider a graph convolutional network loss function L defined for a single input x. Write $L = L(z_i) = L(\hat{w}_i{}^T \hat{z})$ as a function of the corresponding z and hence of the parameter $\hat{w}$. When training over a mini-batch of N inputs, let $\{z_1^{(t-1)}, z_2^{(t-1)}, \ldots, z_N^{(t-1)}\}$ be the associated hidden states, and let $\hat{z}_j^{(t)T} = \begin{bmatrix} 1 & z_j^{(t-1)T} \end{bmatrix}$. Let $L = L(\hat{w}) := \frac{1}{N} \sum_{j=1}^{N} \sum_{z \in \mathcal{N}_{\tau-t}} L(\hat{w}_i^T \hat{z}^{(t)})$ be the mean loss over the mini-batch. Then, its Hessian with respect to $\hat{w}$ is*

$$\nabla^2_{\hat{w}^{(\tau-j)}} L_n(\hat{w}) = \sum_{v_i \in \mathcal{N}_j(v_n)} L''(\hat{z}_i^{(\tau-j)} \hat{w}^{(\tau-j)}) \hat{z}_i^{(\tau-j)} \hat{z}_i^{(\tau-j)T} \tag{2.38}$$

$$= \hat{Z}_n^T S_n \hat{Z}_n \tag{2.39}$$

Where $\mathcal{N}_j(v)$ = All nodes with distance $\leq j$ from node v and

$$\hat{Z}_n^T = \begin{bmatrix} \hat{z}_{i_1}^{(\tau-j)} & \ldots & \hat{z}_{i_{|\mathcal{N}_j(v_n)|}}^{(\tau-j)} \end{bmatrix} \tag{2.40}$$

$$S_n = \begin{bmatrix} L''(\hat{z}_{i_1}^{(\tau-j)} \hat{w}^{(\tau-j)}) & & 0 \\ & \ddots & \\ 0 & & L''(\hat{z}_{i_{|\mathcal{N}_j(v_n)|}}^{(\tau-j)} \hat{w}^{(\tau-j)}) \end{bmatrix} \tag{2.41}$$

In practice, gradients are computed with respect to the aggregated loss, so there is an additional step of the chain rule that is not reflected in the proposition. This results in the implemented methodology computing the mean and variance across the set $M = \bigcup_{n=1}^{N} \mathcal{N}_j(v_n)$ instead of $\bigsqcup_{n=1}^{N} \mathcal{N}_j(v_n)$. This allows us to include every pertinent node without needing to count how many particular neighborhood subgraphs they happen to be a included in.

2.4.1 Implementation

We propose the following preconditioner:

$$P := UD \text{ where } U = \begin{bmatrix} 1 & -\mu_Z^T \\ 0 & I \end{bmatrix} \text{ and } D = \begin{bmatrix} 1 & 0 \\ 0 & \text{diag}(\sigma_Z) \end{bmatrix}^{-1}$$

where

$$\mu_Z = \frac{1}{\text{len}(Z)} Z^T e = \frac{1}{\text{len}(Z)} \sum_{n=1}^{N} \sum_{v_i \in \mathcal{N}_j(v_n)} z_i^{(\tau-j)} \tag{2.42}$$

$$\sigma_Z^2 = \frac{1}{\text{len}(Z)} \sum_{n=1}^{N} \sum_{v_i \in \mathcal{N}_j(v_n)} (z_i^{(\tau-j)} - \mu_Z)^2 \tag{2.43}$$

with $\text{len}(Z) = \sum_{n=1}^{N} |\mathcal{N}_j(v_n)|$

This choice of preconditioner satisfies the criteria of Theorem 2, therefore giving us a better conditioned Hessian.

We will use two base models for our experiments, one uses GCNConv layers [Kipf and Welling, 2017] and the other uses SageConv [Hamilton et al., 2018]. GCNConv is a simple graph convolutional operation expressed in equation form as

$$X' = \hat{D}^{-1/2} \hat{A} \hat{D}^{-1/2} X^{(t)} W \tag{2.44}$$

where $\hat{A}$ is the adjacency matrix augmented with self loops, and $\hat{D}$ is the diagonal degree matrix, $X^{(t)}$ is the hidden states of the nodes at layer t, and W is our trainable parameter matrix. Typical implementations of GCNConv include a bias term b but it is usually omitted in writing the general operation.

Algorithm 5 One Step of BNP for GCNConv

Given: $\varepsilon = 10^{-1}$; learning rate α

Input: Mini-batch hidden states of a particular layer for the nodes in the appropriate neighborhood $Z = \{z_{node}^{(t)}\} \subset \mathbb{R}^m$

Compute parameter gradients: $\frac{\partial L}{\partial W} \in \mathbb{R}^{m \times m}$, $\frac{\partial L}{\partial b} \in \mathbb{R}^m$

$\mu_Z \leftarrow \frac{1}{\text{len}(Z)} \sum_{z \in Z} z$ ▷ Compute Hidden State mean

$\sigma_Z^2 \leftarrow \frac{1}{\text{len}(Z)} \sum_{z \in Z} (z - \mu_Z)^2 + \varepsilon$ ▷ Compute Hidden State variance

$P \leftarrow \begin{bmatrix} 1 & -\mu_Z^T \\ 0 & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \text{diag}\left(\frac{1}{\sigma_Z}\right) \end{bmatrix}$ ▷ Compute Preconditioner

$\begin{bmatrix} b \\ W \end{bmatrix} \leftarrow \begin{bmatrix} b \\ W \end{bmatrix} - \alpha P P^T \begin{bmatrix} \frac{\partial L}{\partial b} \\ \frac{\partial L}{\partial W} \end{bmatrix}$ ▷ Update parameters

SAGEConv includes two weight matrices and is expressed as

$$x_i' = W_1 x_i + W_2 \cdot \text{mean}_{j \in \mathcal{N}_1(i)} x_j \tag{2.45}$$

We can view the taking of the mean over the neighborhoods as being multiplication by the adjacency matrix with rows scaled by the degree. Expressing it like that brings the second term into the same form as for the general GCN. Likewise, we can view the first term as having an identity matrix as its adjacency matrix. In this way, we

Table 2.1: Results for Cora

Method	Test Accuracy (Early Stopping)
ADAM (our baseline)	87.02 ± 0.15
SGD	88.8 ± 0.00
SGD+BNP	89.1 ± 0.00

are able to use our BNP for GCNConv process for each term independently without significant changes. Typical implementations of SAGEConv include bias terms b_1, and b_2 for each set of weights, but they are usually omitted in writing the general operation.

Algorithm 6 One Step of BNP for SAGEConv

Given: $\varepsilon = 10^{-1}$; learning rate α
Input: Mini-batch hidden states for the output nodes $Z_1 = \{z_{node}^{(t)}\} \subset \mathbb{R}^m$, mini-batch hidden states for the nodes neighboring the output nodes $Z_2 = \{z_{node}^{(t)}\} \subset \mathbb{R}^m$
BNP(Z_1, W_1, b_1) ▷ perform BNP from algorithm 5
BNP(Z_2, W_2, b_2) ▷ perform BNP from algorithm 5

2.4.2 Experiments

In this section, we will present a number of experiments which display the improved acceleration gained from the use of BNP.

Cora: Our first experiment is a node classification problem using the Cora dataset from [Yang et al., 2016]. This dataset features 2708 scientific publications. There are 5429 edges in the information graph. The nodes contain a binary vector indicating the presence or absence of words from a 1433 word dictionary. The goal of the task is to sort each node into one of seven classes.

Implementation Details: Since this is a relatively small dataset, we are able to perform full-batch training, meaning that our mini-batch is the size of the training set (or validation/testing set if we are in the evaluation stage). The model consisted of two GCNConv layers with hidden size of 16 separated by a ReLU function. We compare our performance against equivalent models trained using ADAM and SGD. Our model uses our BNP implementation with SGD as the underlying optimizer. The models were run on 10 different seeds.

Results: A convergence curve can be seen in Figure 2.5, testing accuracies can be seen in Table 2.1.The BNP model achieves rapid acceleration in the early epochs which causes it to converge much sooner than the other models. The models were evaluated using early stopping based on the best validation accuracy, so the initial bump gives us a higher test accuracy than the other models despite the BNP model eventually suffering from slight overfitting.

OGB Proteins: Our next experiment is a node prediction task on the Proteins dataset from the Open Graph Benchmark collection. This is an undirected, weighted,

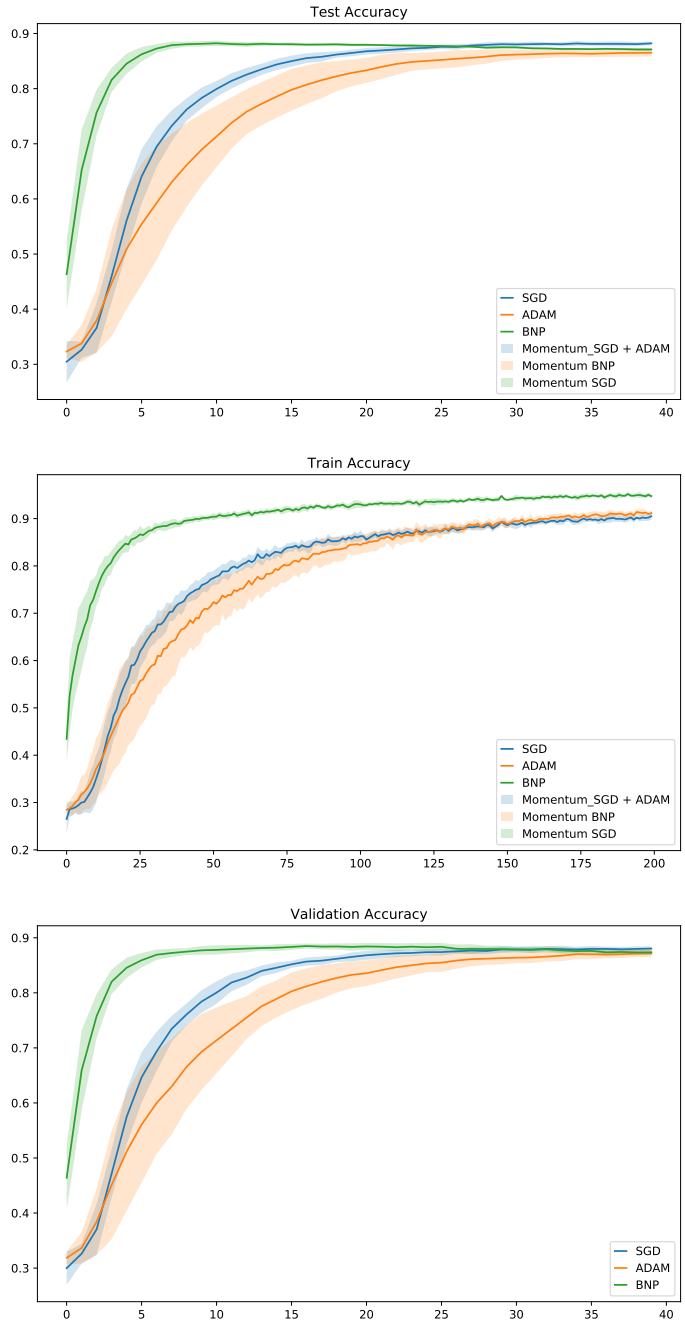


Figure 2.5: Accuracy Curves for Cora

Table 2.2: Results for OGB Proteins

Method	Test Accuracy (Early Stopping)
ADAM (our baseline)	77.31 ± 0.47
SGD	73.25 ± 0.34
SGD+BNP	76.56 ± 0.37

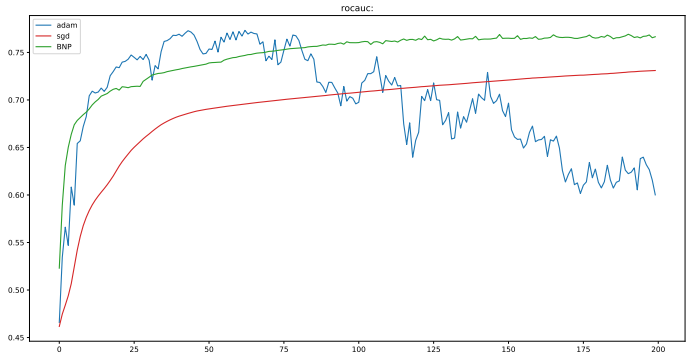


Figure 2.6: Accuracy Curves for OGB Proteins

typed graph where nodes represent different proteins and edges represent types of biological connections. This is a multi-label binary classification problem where the model predicts 112 different labels.

Implementation Details: Our model was trained using a mini-batch size of 1024. It consisted of three SAGEConv layers with hidden size of 256 separated by ReLU nonlinearities. We compare our performance against equivalent models trained using ADAM and SGD. Our model uses our BNP implementation with SGD as the underlying optimizer. The models were run on 3 different seeds.

Results: As seen in Figure 2.6 and Table 2.2 model shows considerable improvement over the model trained using the SGD optimizer in terms of both the test accuracy and the speed of convergence. It does not surpass the performance of the model trained using ADAM. Based on the performance with Cora, we believe that we should be able to match the ADAM results, and there are a number of settings that we are currently testing in order to achieve that.

OGB Products: Our third experiment is a node classification problem using the Products dataset from the Open Graph Benchmark collection. This is an undirected, unweighted graph where nodes represent Amazon products and they are joined by an edge if the two products were bought together. This is a multi-class classification problem where the model seeks to sort the nodes into 47 different bins.

Implementation Details: Our model was trained using a mini-batch size of 1024, with Neighbor sampling. It consists of three SAGEConv layers with hidden size of 256 separated by ReLU nonlinearities. We compare our performance against equivalent models trained using ADAM and SGD. Our model uses our BNP implementation with SGD as the underlying optimizer.

Results: As seen in Figure 2.7 and Table 2.3, our model shows considerable

Table 2.3: Results for OGB Products

Method	Test Accuracy (Early Stopping)
ADAM (our baseline)	79.11 ± 0.12
SGD	72.38 ± 0.03
SGD+BNP	78.36 ± 0.35

improvement over the model trained using the SGD optimizer in terms of both the test accuracy and the speed of convergence. It does not surpass the performance of the model trained using ADAM. Based on the performance with Cora, we believe that we should be able to match the ADAM results, and there are a number of settings that we are currently testing in order to achieve that.

2.4.3 Ablation Studies

In this section we will present a number of performed to test various attributes of our method and its interactions with the base model and some standard techniques.

The first of these is a study on BNP's performance when combined with layer normalization and batch normalization. In the LSTM experiments, we found that there was a performance improvement when we included normalization in the forward pass, so it is natural to try this with the GCN version. We performed these tests on the Cora dataset with a two layer GCN model. There are three places where normalization can take place in this model, as can be seen in the following code for the forward pass:

```
def forward(self, data):
        x, edge_index = data.x, data.edge_index

        x = self.norm_1(x)

        x, bnp_in = self.conv1(x, edge_index)
        self.bnp1.collect_stats(bnp_in)

        x = F.relu(x)
        x = F.dropout(x, training=self.training)

        x = self.norm_2(x)

        x, bnp_in = self.conv2(x, edge_index)
        self.bnp2.collect_stats(bnp_in)

        x = self.norm_3(x)

        return F.log_softmax(x, dim=1)
```

To see the influence of normalization throughout the model, we trained using each combination of having the normalization layers turned on and off, with and without BNP. In Tables 2.5 and 2.4 and Figure 2.8, we can see that the inclusion of Layer Normalization helps accelerate the training accuracy and provides some acceleration over the standard GCN model, but the difference it provides does not provide much

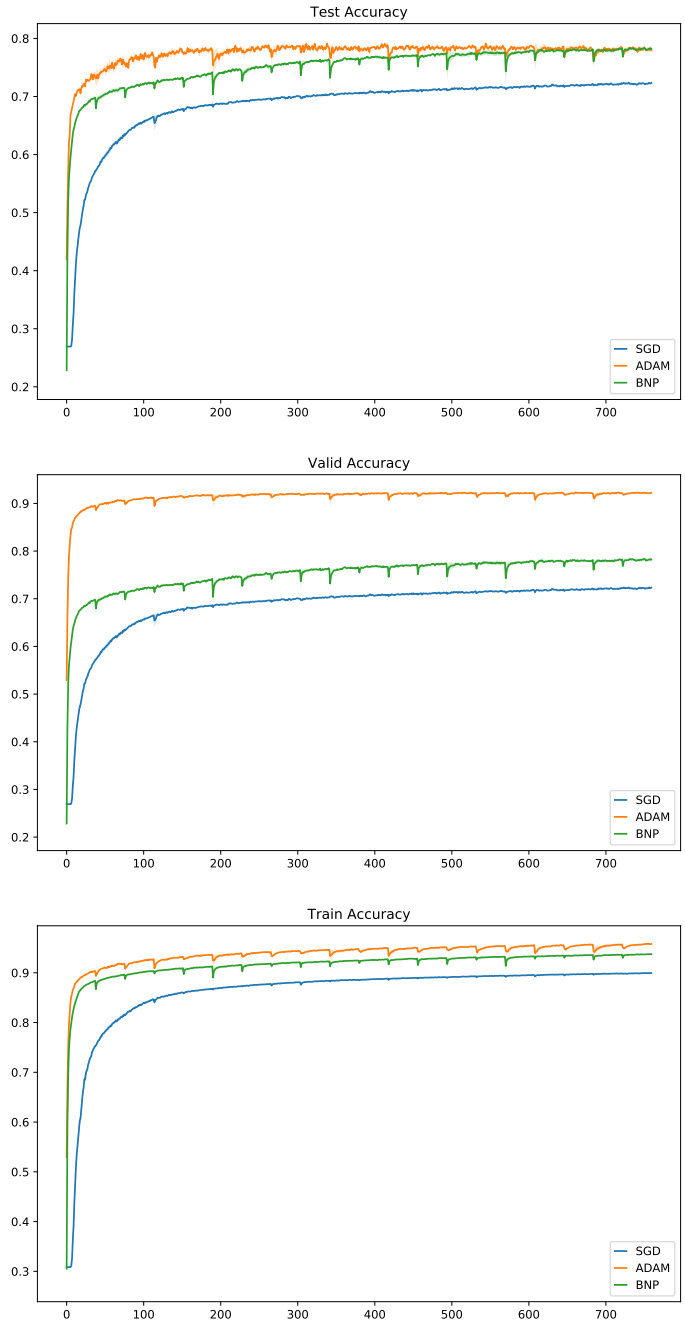


Figure 2.7: Accuracy Curves for OGB Products

Table 2.4: Testing Loss for Normalization Ablation Study

	BN	BN + BNP	LN	LN + BNP
000	0.524	0.511	0.524	0.508
001	0.675	0.594	0.512	0.556
010	0.584	0.571	0.627	0.602
011	0.657	0.645	0.636	0.619
100	0.522	0.639	0.524	0.597
101	0.797	0.758	0.538	0.748
110	0.569	0.694	0.633	0.736
111	0.689	0.748	0.622	0.748

Table 2.5: Testing Accuracy for Normalization Ablation Study

	BN	BN + BNP	LN	LN + BNP
000	88.8	87.2	88.8	87.1
001	87.6	87.4	88.8	87.6
010	86.2	86.2	87.1	85.6
011	87.1	86.6	86.3	87.4
100	86.7	85.2	87.7	85.5
101	84.7	84.6	86.8	85.7
110	86.1	84.8	85.8	83.5
111	84.7	84.2	86.7	83.8

difference in testing accuracies and loss. Adding BNP with LN keeps in check the fluctuations seen with the plain LN model, but it still does not outperform the inclusion of just LN. We further see that the inclusion of Batch Normalization accelerates the model over the standard GCN model, but provides little benefit beyond that, also there seems to be a harmful effect to having normalization in the last position. When adding BNP, we only see worsened performance.

In our second ablation study, we look at the influence that using BNP has on the different layers. This was trained using the Cora dataset with a two layer GCN model. In Figure 2.9 and Table 2.6 we can see the performance with 0 denoting that BNP was turned off for that layer, and 1 denoting that it was on. We can see that applying BNP anywhere in the model provides acceleration over the base model and gives us improved accuracy and loss. Determining which position is best is less obvious. Of the BNP models, having BNP on every level provides the most acceleration, but the model achieves the worst testing accuracy. Having BNP only on the first or second layer provides similar performance in the loss and accuracy metrics, but the first layer model has faster acceleration.

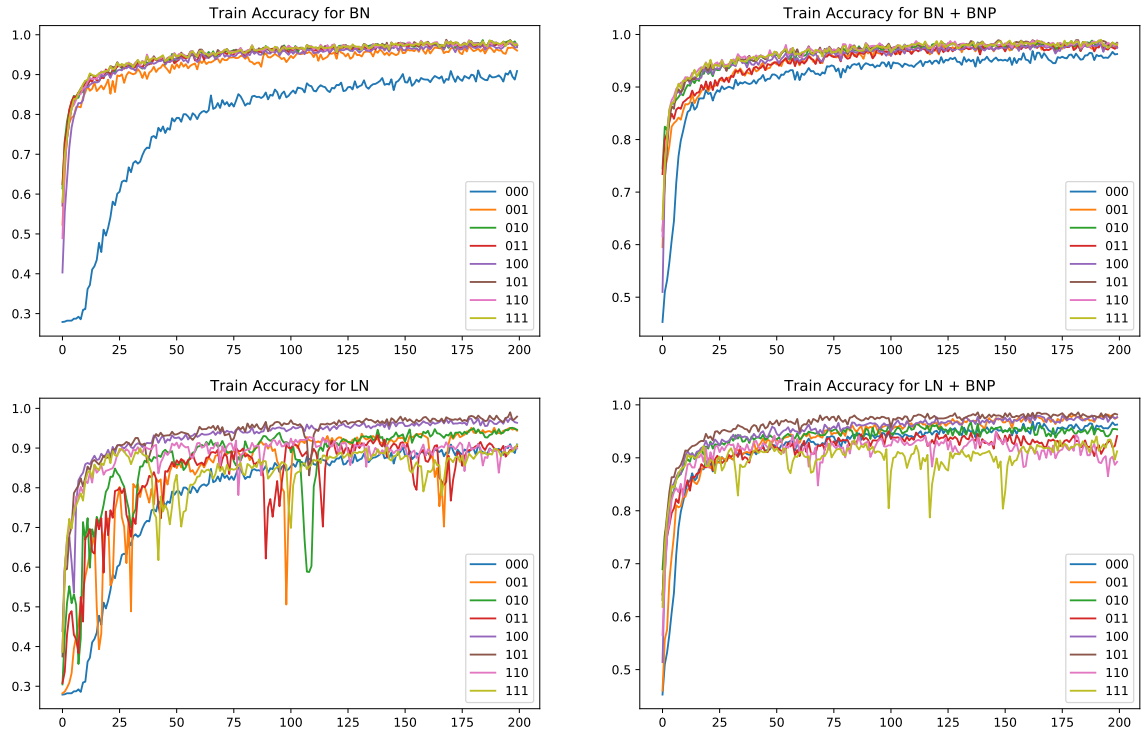


Figure 2.8: Normalization Ablation Study

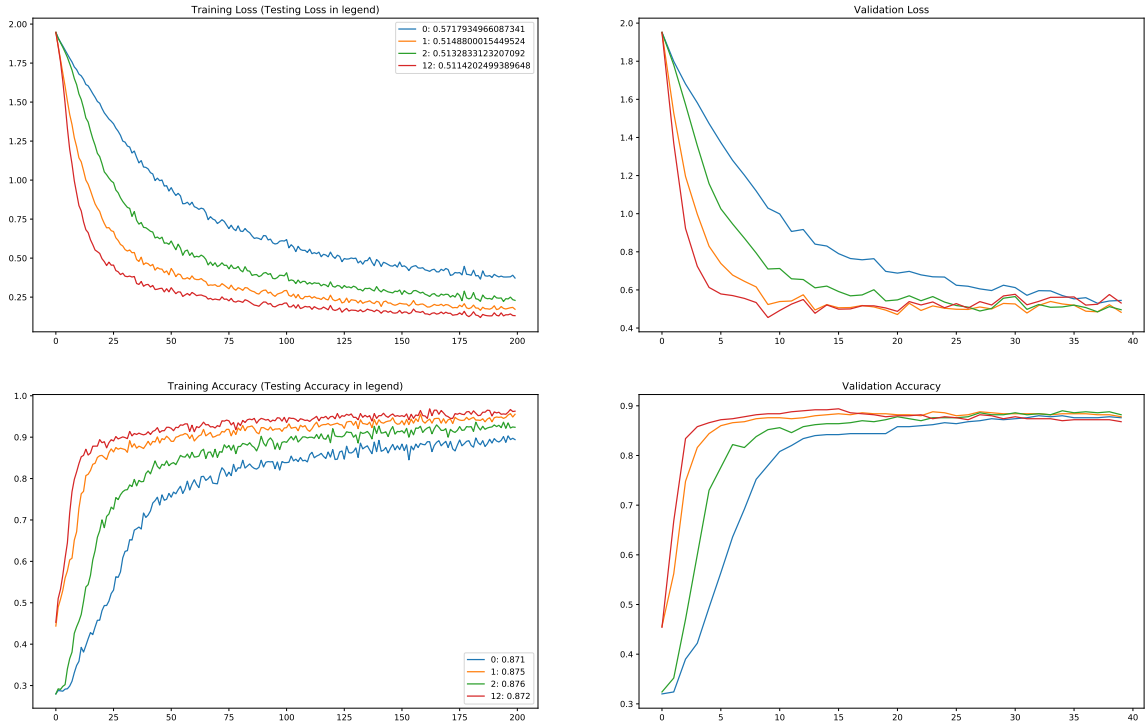


Figure 2.9: BNP on different layers

Table 2.6: Testing Loss and Accuracy for BNP ablation study

	Testing Loss	Testing Accuracy
00	0.572	87.1
01	0.513	87.6
10	0.515	87.5
11	0.511	87.2

2.5 Conclusion

In this chapter, we derived batch normalized preconditioning methods for recurrent neural networks and for graph convolutional networks. This is an expansion of the work done in [Lange et al., 2021] into classes of neural network architecture that are of particular interest in the field. We formulated our methods to use the theoretical results from [Lange et al., 2021] as justification for their use and presented a variety of experiments to support our claims empirically.

Copyright© Cole M. Pospisil, 2023.

Chapter 3 Assorted Temporal Normalization

Methods such as Layer Normalization (LN) and Batch Normalization (BN) have proven to be effective in improving the training of Recurrent Neural Networks (RNN). These existing methods normalize using only the information at one particular time step, and the end result of the normalization process is a preactivation state with a time-independent distribution. This implementation fails to account for the temporal differences inherent in the problem type and the architecture of RNNs. Since these networks share weights across time steps, it may also be desirable to account for these connections between time steps in the normalization methodology. In this chapter, we develop a normalization method called Assorted-Time Normalization (ATN), which preserves information from multiple consecutive time steps and normalizes using them. This setup allows us to introduce longer time dependencies into the traditional normalization methods without introducing any new trainable parameters.

3.1 Introduction

The Recurrent Neural Network (RNN) [Rumelhart et al., 1986, Hopfield, 1982], and RNN's commonly used variant models, such as the Long Short-Term Memory cell (LSTM) [Hochreiter and Schmidhuber, 1997] or the more recent Gated Recurrent Unit cell (GRU) [Cho et al., 2014, Jing et al., 2019], are some of the core architectures used for modeling time-series data in Deep Learning today. While LSTMs and GRUs are effective in avoiding problems with vanishing gradients, all of these recurrent models are still subject to issues with exploding gradients, as well as over-fitting. One of the most successful ideas that have been introduced over the years is the normalization of RNNs using methods such as Layer Normalization (LN) [Ba et al., 2016] and Batch Normalization (BN) [Cooijmans et al., 2017]. These methods recenter and rescale the preactivation information using the statistics of that time step. This allows for the norm of the model's states and gradients to be controlled, which speeds up training and prevents exploding gradients.

While these normalization methods have been successful, their applications to RNNs do not involve adaptations to the variation across time and therefore loses the usable information which it imparts. For example, the LN or BN models are invariant to the scaling in the input at any time step and are, therefore, independent of the norm of the input vector at each time step. Depending on the applications, this may have devastating consequences. Additionally, LN and BN produce a preactivation state with a distribution that is invariant across time. Such time invariance properties may impede the architectural structure of RNN's ability to exploit the temporal dependencies fully. Since RNNs share weights across time steps, it would be natural to introduce this dependency into the normalization method as well. An attempted version of this involving averaging statistics across time was mentioned in [Cooijmans et al., 2017] but was unsuccessful and was presented without much de-

tail. It appears that simply averaging over every time step is an overcorrection that makes the statistics susceptible to diluted averages and loses effectiveness further into the sequence. Instead, we argue that by collecting the mean and variance across a smaller subsequence, one can gain benefit from these time dependencies without overly weakening the impact of a single time step.

Assorted-Time Normalization (ATN) is a normalization technique which preserves information from multiple consecutive time steps and normalizes using them. Our ATN method can be combined with other normalization methods, such as LN and BN, that normalize input information along some dimensions but not time. It maintains a short-term memory of the previous k time steps, which allows it to account for the temporal dependencies in a way in which previous methods were incapable. We use that memory to calculate the statistics used in normalization, giving us an output that has a controlled mean and variance while still being capable of changing between time steps. By using just a limited subsequence at each point in time, we can avoid the problems that come from using all or none of the sequences and find the length best suited to the dataset. Since this process works by changing which set statistics are collected from and not adding any weights, it can be used without the introduction of any new learnable parameters.

We present theoretical derivations for the gradient propagation and prove the weight scaling invariance property. Our experiments demonstrate that the inclusion of our method provides consistent improvement on variou's tasks, such as Adding, Copying, and Denoise Problems as well as Language Modeling Problems.

3.2 Related Work

One of the earliest attempts to use some normalization technique throughout model layers was Batch Normalization (BN) [Ioffe and Szegedy, 2015a]. It was proposed for Fully Connected (FC) and Convolutional (CNN) Neural Networks to normalize network activations across the batch dimension. BN is known often to provide a more stable and accelerated training process while improving generalizations. The Instance Normalization (IN) [Ulyanov et al., 2017] method, contrary to BN, acts like contrast normalization and has primarily been used for image-containing datasets. The paper points out that the output stylized images should not rely on the contrast of the input image content, and hence normalizing the instances helps. The Group Normalization (GN) [Wu and He, 2018] method, which is primarily used for CNNs, normalizes a 3D feature in a convolutional layer by dividing its channels into groups and then normalizing the features in the group in all three dimensions.

Consider the typical structure of an RNN, also known as an RNN cell:

$$h^{(t)} = f\left(W_h h^{(t-1)} + W_x x^{(t)} + \beta_h\right) \tag{3.1}$$

$$y^{(t)} = W_y h^{(t)} + \beta_y \tag{3.2}$$

where f is a nonlinear activation function.

Recurrent Batch Normalization [Cooijmans et al., 2017] applies BN to the hidden-to-hidden and memory cell parts of the LSTM model, which aims to reduce the internal covariate shift between consecutive time steps. [Salimans and Kingma, 2016] proposed Weight Normalization (WN) which seeks to decouple the magnitude from the direction of the weight vector to change the parameters of the network and help speeding up learning. Unfortunately, WN appears not widely used in practice due to its limited stability compared to BN [Gitman and Ginsburg, 2017].

Layer Normalization (LN) was proposed in [Ba et al., 2016] to normalize activations along the hidden dimension for both FC networks and RNNs and has since become very popular in RNNs. LN normalizes the preactivation state as follows:

$$h^{(t)} = f\left(LN\left(W_h h^{(t-1)}\right) + LN\left(W_x x^{(t)}\right) + \beta_h\right) \tag{3.3}$$

$$y^{(t)} = LN\left(W_y h^{(t)}\right) + \beta_y \tag{3.4}$$

where the LN operator is defined by

$$\mu_t = \frac{1}{n}\sum_{i=1}^{n} a_i^{(t)} \quad \sigma_t^2 = \frac{1}{n}\sum_{i=1}^{n}\left(a_i^{(t)} - \mu_t\right)^2 \tag{3.5}$$

$$y^{(t)} = LN(a^{(t)}; \gamma, \beta) = \gamma \odot \frac{a^{(t)} - \mu_t}{\sqrt{\sigma_t^2 + \varepsilon}} + \beta \tag{3.6}$$

Such a setup helps eliminate the BN dependency on batch size and selection and simplifies the application to RNNs.

More recently, Adaptive Normalization (AdaNorm) [Xu et al., 2019] made a thorough analysis of LN and concluded that the rescaling and recentering factors, γ and β in (3.6), are not as essential as the backward gradients of the mean and variance inside of the LN method. In addition, they proposed a new method, AdaNorm, which replaces weight and bias with some new transformation function.

3.3 Assorted Time Normalization

One undesirable property of the adaptation of LN to RNNs is that the statistics for the normalization are calculated at each time step, resulting in a post-normalization state which has mean and variance that are invariant across time. This prevents the model from effectively representing the shifting distributions across time that might be critical in modeling sequential data. For example, the normalization $LN\left(W_x x^{(t)}\right)$ in (3.3) is invariant to scaling in $x^{(t)}$, which restricts the model from learning the changing norm of $x^{(t)}$. This may be mitigated by including a bias in the linear term, which is often used in implementations; see section 3.5.1 for more discussion. Most of the above discussions also apply to BN.

We propose a new normalization method to break this time invariance. Consider a sequence $\mathbf{a} = \left\{a^{(t)}\right\} \subset \mathbb{R}^n$ produced in an RNN, such as the preactivation state that we wish to normalize. At time step t of the RNN, we maintain a memory of

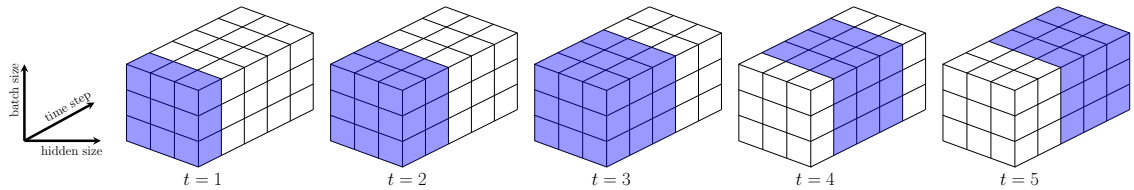


Figure 3.1: Illustration of the ATN method combined with LN using $k = 3$ time steps: Consider the preactivation state tensor in $\mathbb{R}^{5\times3\times3}$. At $t = 1$, we use a standard LN; at $t = 2$, we normalize using information from time steps $1, 2$; at $t = 3$ ATN method uses information from time step $t = 1, 2, 3$; at $t = 4$, we normalize with respect to time steps $t = 2, 3, 4$; and so on after that.

the previous k entries, $\mathbf{a}_k^{(t)} = \left\{a^{(t-k+1)}, \ldots, a^{(t-1)}, a^{(t)}\right\} \subset \mathbf{a}$, in the normalization layer, using this extended set to compute the mean and variance to be used for normalization. This can be combined with other normalization methods. Combining with Layer Normalization, for example, these statistics are calculated at time-step t as follows:

$$\mu_{t,k} = \frac{1}{nk}\sum_{j=0}^{k-1}\sum_{s=1}^{n} a_s^{(t-j)} \tag{3.7}$$

$$\sigma_{t,k}^2 = \frac{1}{nk}\sum_{j=0}^{k-1}\sum_{s=1}^{n} \left(a_s^{(t-j)} - \mu_{t,k}\right)^2 \tag{3.8}$$

See Figure 3.1 for a visual depiction of our method.

Once these statistics are calculated, we then normalize only the current term $a^{(t)}$ and optionally recenter and rescale using γ and β, two trainable parameters shared across time while adding a small epsilon to the variance to prevent division by zero, similar to the LN method in (3.6).

$$y^{(t)} = ATN(\mathbf{a}_k^{(t)}; \gamma, \beta) := \gamma \odot \frac{a^{(t)} - \mu_{t,k}}{\sqrt{\sigma_{t,k}^2 + \varepsilon}} + \beta \tag{3.9}$$

This differs from the process in (3.6) in that we include multiple time steps in our statistic calculations, giving us a double sum instead of the single one for LN. This definition of the statistics is more stable in time, at least for large k, changing modestly at each time step with only one term in the set being replaced. This results in a normalized output that is not expected to have a uniform mean and variance across time steps. We argue that this is desirable for sequential problems. Having this potential for variation allows for the model to account for changing norms of the inputs across the sequence, providing additional information about the distribution that is lost with previous methods.

We may consider using all previous terms in the sequence to compute the statistics, but they will have more variation in early time steps than in later ones. By keeping

only k time steps and not the entire sequence, the statistics will vary gradually across time, and we are able to limit the memory and computational costs, which could be significant for long sequences.

Using information from multiple time steps also effectively provides a larger set on which to calculate statistics. This allows a more accurate approximation to gain a clearer glimpse at the underlying distribution of the dataset. In other words, ATN uses statistics over a larger set that is more stable across time so that the normalized state can retain more variations in time. In contrast, the traditional normalization methods use high-frequency statistics at each time step to produce a normalized state that becomes time-invariant. In particular, the ATN network depends on the scaling of the input vector at a time step, while LN and BN do not. However, ATN preserves the desirable weight scaling invariant property, which we show as follows:

Let H and $\tilde{H}$ be weight matrices for two sets of model parameters, θ and $\tilde{\theta}$ respectively, which differ by a scaling factor of δ, i.e. $\tilde{H} = \delta H$. Then the outputs of ATN are the same:

$$\tilde{y}^{(t)} = \frac{\gamma}{\tilde{\sigma}_{t,k}} \odot \left(\tilde{H}a^{(t)} - \tilde{\mu}_{t,k}\right) + \beta \ = \frac{\gamma}{\sigma_{t,k}} \odot \left(Ha^{(t)} - \mu_{t,k}\right) + \beta = y^{(t)} \tag{3.10}$$

where $\tilde{\sigma}_{t,k} = \delta\sigma_{t,k}$ and $\tilde{\mu}_{t,k} = \delta\mu_{t,k}$. This invariance property makes the ATN network independent of the norm of H, mitigating the exploding/vanishing gradient problems.

It is also easy to see that ATN is also invariant to the rescaling of the whole input sequence but not invariant under the rescaling of an individual element in the sequence.

During training, we backpropagate the gradients with respect to the model parameters. With ATN, a key step is to propagate the gradient through the normalization layer, i.e., $\frac{\partial y_i^{(t)}}{\partial a_i^{(t-m)}}$. The following proposition gives the formulas for computing these derivatives.

Proposition 5. *Consider ATN for a sequence* $\mathbf{a} = \{a^{(t)}\} \subset \mathbb{R}^n$ *produced in a RNN and let* $y^{(t)} = ATN(\mathbf{a}_k^{(t)}; \gamma, \beta)$*. Then, for* $0 \leq m \leq k-1$*, we have:*

$$\frac{\partial y_i^{(t)}}{\partial a_i^{(t-m)}} = \gamma \odot \frac{\frac{\partial a_i^{(t)}}{\partial y_i^{(t-m)}} \frac{\partial y_i^{(t-m)}}{\partial a_i^{(t-m)}} - \frac{\partial \mu_{t,k}}{\partial a_i^{(t-m)}}}{\sqrt{\sigma_{t,k}^2 + \varepsilon}} - \gamma \odot \frac{a_i^{(t)} - \mu_{t,k}}{2\left(\sigma_{t,k}^2 + \varepsilon\right)^{3/2}} \frac{\partial \sigma_{t,k}^2}{\partial a_i^{(t-m)}} \tag{3.11}$$

where

$$\frac{\partial \mu_{t,k}}{\partial a_i^{(t-m)}} = \frac{1}{nk} \sum_{j=0}^{m} \frac{\partial a_i^{(t-j)}}{\partial a_i^{(t-m)}} \tag{3.12}$$

$$\frac{\partial \sigma_{t,k}^2}{\partial a_i^{(t-m)}} = \frac{2}{nk} \sum_{j=0}^{m} \left(a_i^{(t-j)} - \mu_{t,k} \right) \frac{\partial a_i^{(t-j)}}{\partial a_i^{(t-m)}} - \sum_{j=0}^{k-1} \sum_{s=1}^{n} \left(a_s^{(t-j)} - \mu_{t,k} \right) \frac{\partial \mu_{t,k}}{\partial a_i^{(t-m)}}. \tag{3.13}$$

Proof. Suppose $0 \leq m \leq k-1$ then

$$\mu_{t,k} = \frac{1}{nk} \sum_{j=0}^{k-1} \sum_{s=1}^{n} a_s^{(t-j)} \tag{3.14}$$

and

$$\frac{\partial \mu_{t,k}}{\partial a_i^{(t-m)}} = \frac{1}{nk} \sum_{j=0}^{k-1} \sum_{s=1}^{n} \frac{\partial a_s^{(t-j)}}{\partial a_i^{(t-m)}} \tag{3.15}$$

$$= \frac{1}{nk} \left(\frac{\partial a_i^{(t)}}{\partial a_i^{(t-m)}} + \frac{\partial a_i^{(t-1)}}{\partial a_i^{(t-m)}} + \cdots + \frac{\partial a_i^{(t-m+1)}}{\partial a_i^{(t-m)}} + 1 \right) \tag{3.16}$$

$$= \frac{1}{nk} \sum_{j=0}^{m} \frac{\partial a_i^{(t-j)}}{\partial a_i^{(t-m)}}; \tag{3.17}$$

$$\sigma_{t,k}^2 = \frac{1}{nk} \sum_{j=0}^{k-1} \sum_{s=1}^{n} \left(a_s^{(t-j)} - \mu_{t,k} \right)^2 \tag{3.18}$$

and

$$\frac{\partial \sigma_{t,k}^2}{\partial a_i^{(t-m)}} = \frac{1}{nk} \sum_{j=0}^{k-1} \sum_{s=1}^{n} 2 \left(a_s^{(t-j)} - \mu_{t,k} \right) \left(\frac{\partial a_s^{(t-j)}}{\partial a_i^{(t-m)}} - \frac{\partial \mu_{t,k}}{\partial a_i^{(t-m)}} \right) \tag{3.19}$$

$$= \frac{2}{nk} \left(\sum_{j=0}^{m} \left(a_i^{(t-j)} - \mu_{t,k} \right) \frac{\partial a_i^{(t-j)}}{\partial a_i^{(t-m)}} - \sum_{j=0}^{k-1} \sum_{s=1}^{n} \left(a_s^{(t-j)} - \mu_{t,k} \right) \frac{\partial \mu_{t,k}}{\partial a_i^{(t-m)}} \right); \tag{3.20}$$

$$y^{(t)} = \gamma \odot \frac{a^{(t)} - \mu_{t,k}}{\sqrt{\sigma_{t,k}^2 + \varepsilon}} + \beta \tag{3.21}$$

and

$$\frac{\partial y_i^{(t)}}{\partial a_i^{(t-m)}} = \gamma \odot \frac{\dfrac{\partial a_i^{(t)}}{\partial a_i^{(t-m)}} - \dfrac{\partial \mu_{t,k}}{\partial a_i^{(t-m)}}}{\sqrt{\sigma_{t,k}^2 + \varepsilon}} - \gamma \odot \frac{1}{2} \frac{a_i^{(t)} - \mu_{t,k}}{\left(\sigma_{t,k}^2 + \varepsilon \right)^{3/2}} \frac{\partial \sigma_{t,k}^2}{\partial a_i^{(t-m)}} \tag{3.22}$$

where

$$\frac{\partial a_i^{(t)}}{\partial a_i^{(t-m)}} = \frac{\partial a_i^{(t)}}{\partial y_i^{(t-m)}} \frac{\partial y_i^{(t-m)}}{\partial a_i^{(t-m)}}. \tag{3.23}$$

□

Note that the computations of $\frac{\partial y_i^{(t)}}{\partial \beta}$ and $\frac{\partial y_i^{(t)}}{\partial \gamma}$ are straightforward and are omitted.

In our experiments, we will use ATN on LSTM networks. Following the methodology of [Ba et al., 2016] and [Cooijmans et al., 2017], our ATN method for LSTM is as follows :

$$\begin{pmatrix} f^{(t)} \\ i^{(t)} \\ o^{(t)} \\ g^{(t)} \end{pmatrix} = ATN(W_h h^{(t-1)}) + ATN(W_x x^{(t)}) + b \tag{3.24}$$

$$c^{(t)} = \sigma(f^{(t)}) \odot c^{(t-1)} + \sigma(i^{(t)}) \odot \tanh(g^{(t)}) \tag{3.25}$$

$$h^{(t)} = \sigma(o^{(t)}) \odot \tanh(ATN(c^{(t)})) \tag{3.26}$$

where $\odot$ is the Hadamard product and $\sigma(\cdot)$ is the sigmoid function.

3.4 Experiments

We have performed a series of sequential data experiments which include the Copying Problem and Adding Problems [Hochreiter and Schmidhuber, 1997], and the Denoise Problem [Jing et al., 2019, Foerster et al., 2016] as well as Language Modeling on character level Penn Treebank dataset [Marcus et al., 1993].

All experiments were run using Python 3.7.0, PyTorch 1.1.0, and CUDA 9.0 on a single NVIDIA Tesla V100 GPU.

3.4.1 Synthetic Tasks

Copying

The copying problem is a common synthetic task that is used to test RNNs, which was originally proposed in [Hochreiter and Schmidhuber, 1997]. For this problem, a string of 10 digits is fed into the RNN sampled uniformly from the integers between 1 and 8. A sequence of T zeros follows this, and a 9, marking the start of a string of 9 zeros, for a total length of $T + 20$. The objective of the task is to output the initial string of 10 digits beginning at the marker's location, copying the initial string from the front to the back. Cross-entropy loss is used to evaluate this model, with a baseline expected cross-entropy of $\frac{10 \log(8)}{T+20}$ which represents selecting digits 1-8 at random after the 9.

Implementation Details: The models were trained with a batch size of 128, a single LSTM layer with a hidden size of 68, an RMSProp [Tieleman and Hinton, 2012]

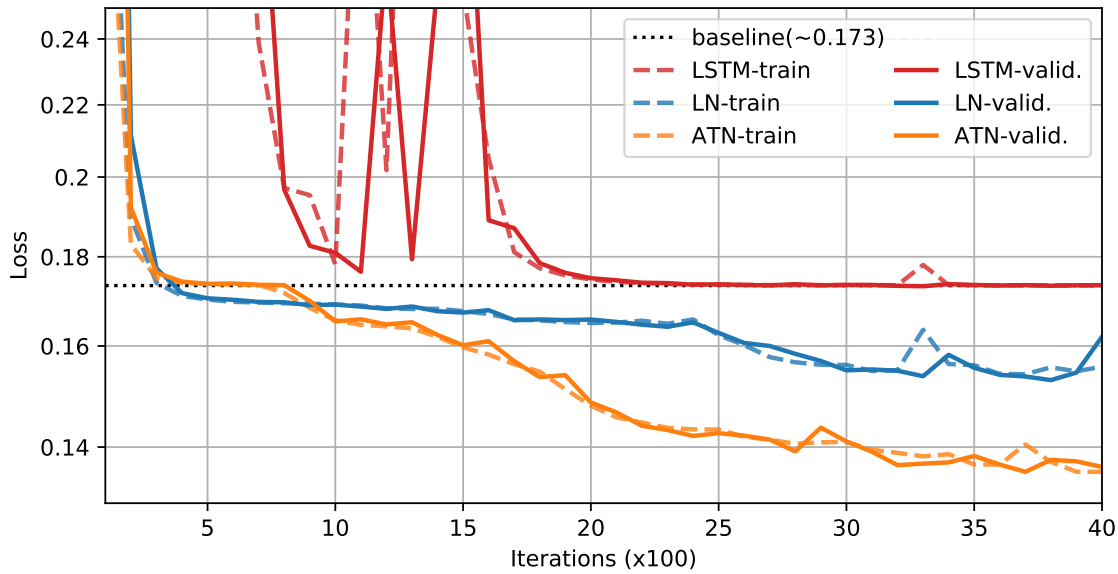


Figure 3.2: Copying problem with $T = 100$

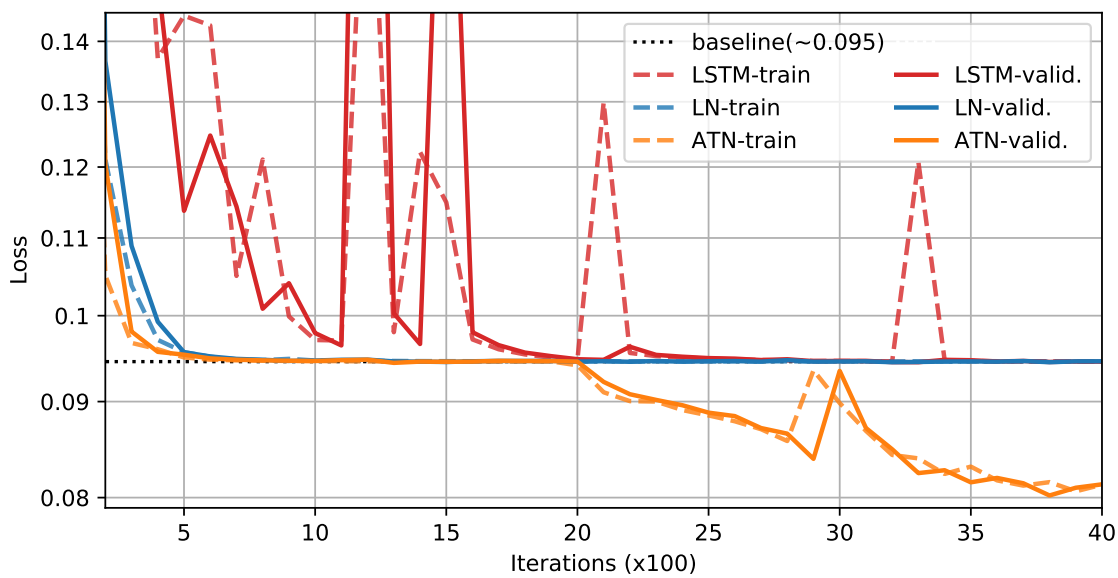


Figure 3.3: Copying problem with $T = 200$

optimizer with a learning rate of 10^{-4}, and T values of 100 and 200. The ATN model is implemented with $k = 45$ for both T values.

Results: For each of the sequence lengths tested, the plain LSTM is incapable of achieving losses below the baseline. While the LN-LSTM is able to do so to some extent on the $T = 100$ version, see Figure 3.2, it also gets stuck at the baseline loss on the $T = 200$ task, Figure 3.3. For both of these tasks, our ATN-LSTM model demonstrates eventual losses below those reached by the LN-LSTM model, Figures 3.2 and 3.3. We also note that the initial rate of convergence is at least as steep if

Table 3.1: Copying Results: Attained minimum values. ↓ - denotes the smaller, the better result.

	Loss $\times 10^{-1} \downarrow$		Loss $\times 10^{-2} \downarrow$	
Sequence Length	$T = 100$		$T = 200$	
	train	validation	train	validation
LSTM	1.739	1.731	9.445	9.445
LN	1.542	1.529	9.445	9.445
ATN	1.354	1.354	8.020	8.020

not steeper than that of the LN-LSTM model, demonstrating that the ATN-LSTM positively contributes to training in both the short and long term.

Adding

The adding problem is another synthetic task for RNNs which was originally proposed in [Hochreiter and Schmidhuber, 1997]. Our implementation of this problem is a variation of the original problem. The RNN takes a 2-dimensional input of length T. The first dimension consists of a sequence of zeros except for two ones placed randomly in the first and second half of the sequence. The second dimension is a sequence of numbers selected uniformly from $[0, 1)$. The goal of the task is to take the numbers from the second dimension in positions corresponding to the ones and to output their sum.

Implementation Details: The models were trained with a batch size of 50, a single LSTM layer with a hidden size of 60, and an RMSprop [Tieleman and Hinton, 2012] optimizer with a learning rate of 10^{-3}. We use T values of 100 and 200. This task is evaluated with a mean-squared error. The ATN model is implemented with k values of 25 for $T = 100$ and 5 for $T = 200$.

Results: Our model shows consistent improvement over the LSTM and LN-LSTM models. For each example, the ATN shows a rapid initial convergence before settling into a slower rate which is roughly parallel to that of the LN-LSTM. In Figure 3.4, this initial conversion almost manages to take the model to the same loss as is achieved by the LN-LSTM after the entirety of the training. In Figure 3.5, the LN-LSTM is able to separate itself further from the LSTM than in Figure 3.4 but is still at a higher loss than the ATN for all but the very beginning of training.

Denoise Task

The Denoise Task [Jing et al., 2019, Foerster et al., 2016] is another synthetic problem that requires filtering out the noise out of a noisy sequence. This problem requires the forgetting ability of the network as well as learning long-term dependencies coming from the data [Jing et al., 2019]. The input sequence of length T contains 10 randomly located data points, and the other $T - 10$ points are considered noise data.

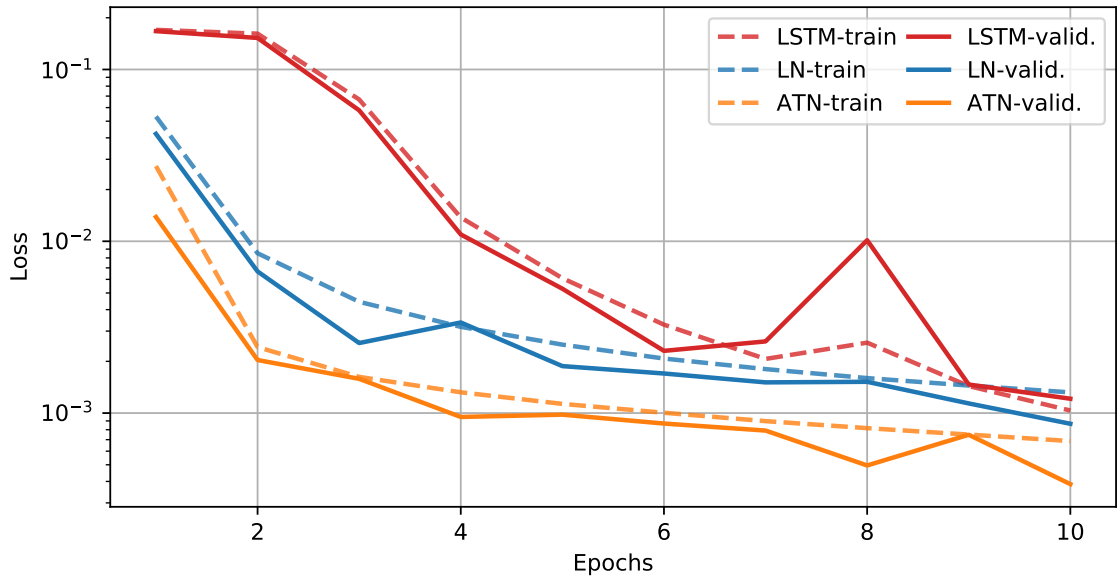


Figure 3.4: Adding problem with $T = 100$

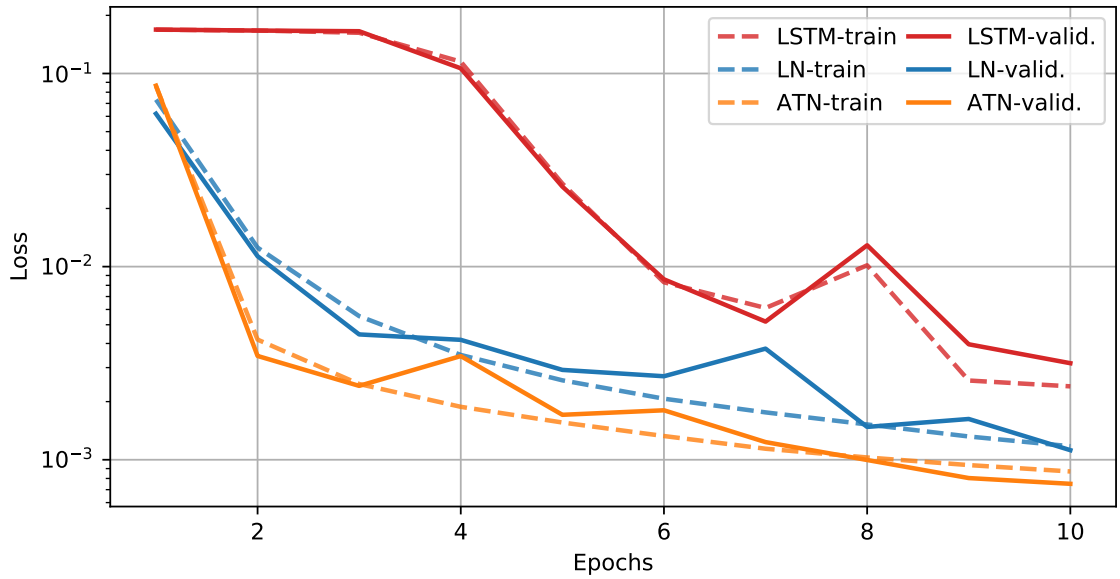


Figure 3.5: Adding problem with $T = 200$

Table 3.2: Adding Results: Attained minimum values. ↓ - denotes the smaller, the better result.

	Loss $\times 10^{-3}$ ↓		Loss $\times 10^{-3}$ ↓	
Sequence Length	$T = 100$		$T = 200$	
	train	validation	train	validation
LSTM	1.034	1.212	2.390	3.161
LN	1.319	0.866	1.174	1.121
ATN	0.687	0.385	0.869	0.750

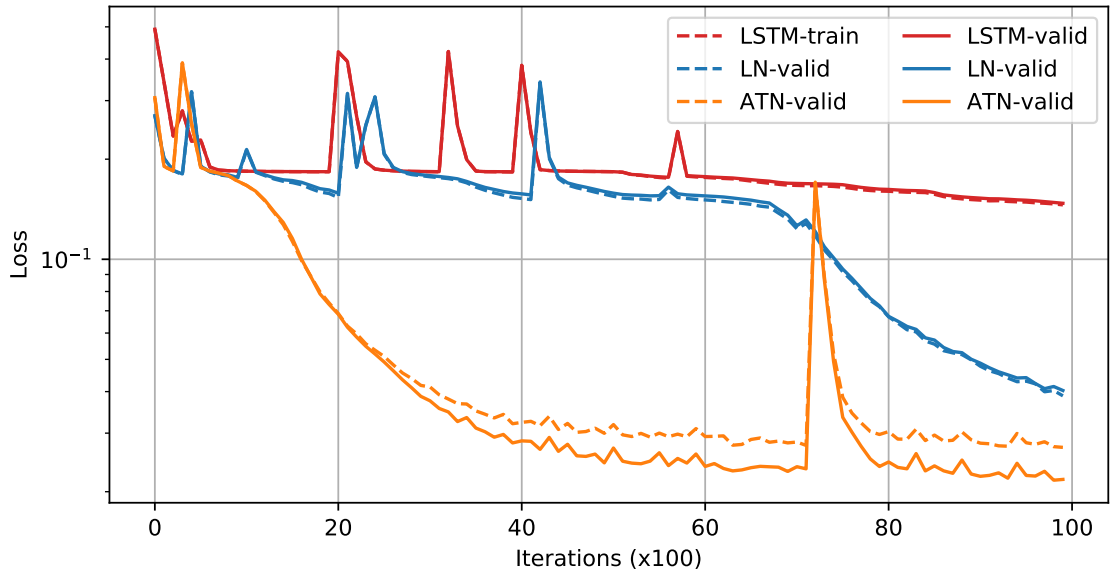


Figure 3.6: Denoise task with $T = 100$

These 10 points are selected from a dictionary $\{a_i\}_{i=0}^{n+1}$, where the first n elements are data points, and the other two are the "noise" and the "marker" respectively. The output data consists of the list of the data points from the input, and it should be outputted as soon as it receives the "marker". The goal is to filter out the noise and output the random 10 data points chosen from the input.

Implementation Details: The models were trained using a batch size of 128, a single LSTM layer with a hidden size of 100, and Adam [Kingma and Ba, 2014] optimizer with a learning rate of 10^{-2}. We use T values of 100 and 200. The ATN model is implemented with k values of 20 and 60 for $T = 100$ and $T = 200$, respectively.

Results: For both sequence lengths, our models outperform the LSTM and the LN-LSTM throughout training. While the LN-LSTM model can surpass the baseline set by the LSTM, it does so later than the ATN model, and its convergence curve flattens out at a higher loss than the ATN model.

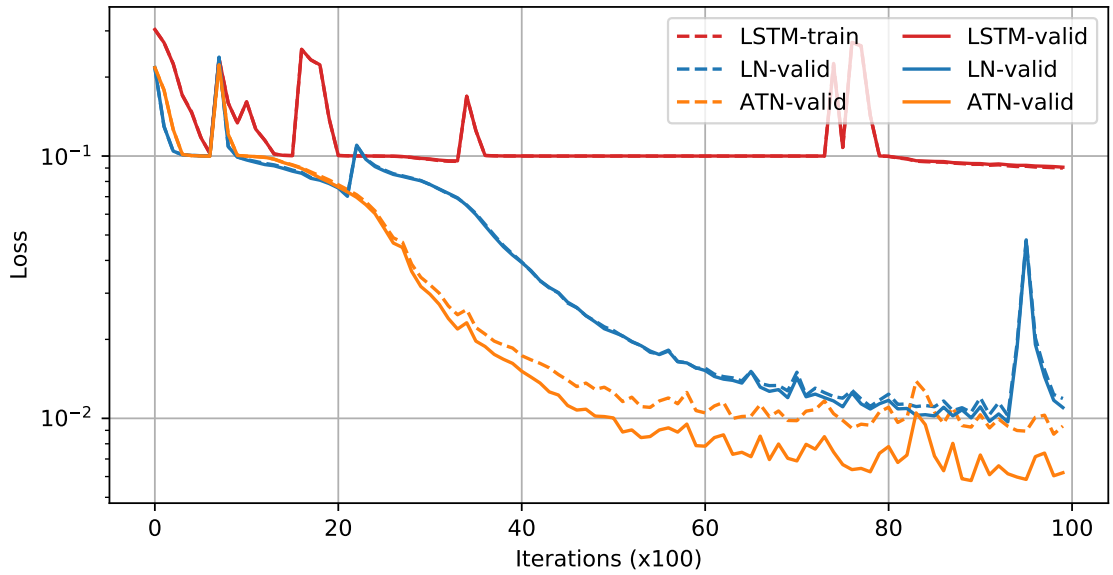


Figure 3.7: Denoise task with $T = 200$

Table 3.3: Denoise Results: Attained minimum values. ↓ - denotes the smaller, the better result.

	Loss $\times 10^{-2}$ ↓		Loss $\times 10^{-3}$ ↓	
Sequence Length	$T = 100$		$T = 200$	
	train	validation	train	validation
LSTM	14.22	14.64	88.30	89.88
LN	2.489	3.169	6.108	9.057
ATN	1.733	2.073	4.70	5.433

3.4.2 Language Models

Language modeling is one of many natural language processing tasks. It is the development of probabilistic models that are capable of predicting the next word or character in a sequence using information that has preceded it. For both of the Language Modeling problems, we based our experiments on the AWD-LSTM model [Merity et al., 2018].

Character Level Penn Treebank

The models were tested on their suitability for language modeling tasks using the character level Penn Treebank dataset [Marcus et al., 1993] also known as character-PTB or simply cPTB dataset. This dataset is a collection of English-language Wall Street Journal articles. The dataset consists of a vocabulary of 10,000 words with other words replaced as `<unk>`, resulting in approximately 6 million characters that

are divided into 5.1 million, 400 thousand, and 450 thousand character sets for training, validation, and testing, respectively with a character alphabet size of 50. The goal of the character-level Language Modeling task is to predict the next character given the preceding sequence of characters.

Implementation Details: For this task, we partitioned the training sequence into 220 character length subsequences. The models were trained using a batch size of 32, a single LSTM layer with a hidden size of 1,000, an Adam [Kingma and Ba, 2014] optimizer with a learning rate of 10^{-2}, gradient clipping by norm at 3, and learning rate decay by a factor of 10 at epoch 80 and 90. The ATN model is implemented with a k value of 10.

Results: Our model shows improvement over the LSTM and the LN-LSTM models, the comparison results are presented in Table 3.4.

Table 3.4: Character Level Penn Treebank Results: Attained minimum values. ↓ - denotes the smaller, the better result

	bpc ↓	
	train	validation
LSTM	1.692	1.743
LN	1.390	1.520
ATN	1.381	1.511

3.5 Ablation Studies

3.5.1 Input Statistic Invariance Across Time

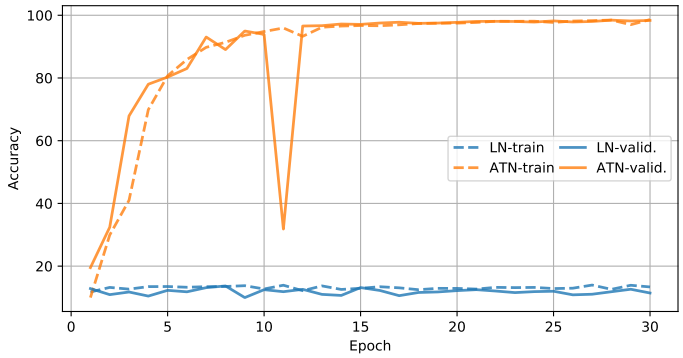


Figure 3.8: Pixel-by-pixel MNIST

In most implementations of LN-LSTM, including the one used in the experiments above, the inputs to the normalization method are the results of a linear layer, including both weight and bias. This differs slightly from the model proposed in [Ba et al., 2016] in that their version placed the LSTM bias outside of the normalization. Using that original architecture, we can clearly demonstrate the underlying

problem with Layer Normalization that we aim to solve, the loss of input information, by setting the statistics to constant values across time.

To show this, we use the MNIST dataset [Lecun et al., 1998] after applying Gaussian noise with variance 0.1 for the pixel-by-pixel task [Le et al., 2015]. This task takes the pixel values of a handwritten digit and inputs them as an unpermuted sequence of length 784 in order to predict the digit class. Due to the high probability of pixels having near zero values, we needed to use ε values of 1 in both normalization schemes. With this task, we can see in Figure 3.8 that the use of Layer Normalization renders the model completely incapable of training. Because LN takes the information from each pixel and normalizes it to the exact same distribution, it erases everything the model could use to learn, making it no better than guessing. The ATN method with $k = 10$ solves this problem by using multiple time steps in calculating the mean and variance, meaning that the normalized outputs will not all have identical statistics. This change allows ATN to perform quite well, even when Layer Normalization cannot.

3.5.2 Post Normalization Statistics

In Figures 3.9, 3.10, and 3.11, we present the statistics of the post-normalization components, $W_h h^{(t-1)}$, $W_x x^{(t)}$, and $c^{(t)}$, from a single iteration of training for the Adding Problem [Hochreiter and Schmidhuber, 1997] described in Section 3.4.1 with $T = 75$. We present the statistics from four different models, an LN-LSTM, and three ATN(k) models with k values of 5, 25, and 55. All of the models did not include the use of trainable bias and gain parameters inside the normalization methods.

In Figure 3.9, we show the mean and variance after normalization of the product of the hidden-to-hidden weight and the hidden state, $W_h h^{(t-1)}$. While Layer Normalization produces constant mean and variance, the ATN method allows for the statistics to vary at each time step, resulting in curves that do not differ too much from those for LN in terms of scale but do demonstrate the natural fluctuations in the hidden states. From this, we can see that we are achieving the combination of a controlled output that is still capable of reflecting the temporal changes of the network.

In Figure 3.10, we show the statistics from the product of the input-to-hidden weight and the input, $W_x x^{(t)}$. The ATN model provides highly variable means and variances, showcasing the amount of information about the dataset which is lost when LN resets the statistics to these constant values.

In Figure 3.11, we show the post-normalization statistics of the memory cell, $c^{(t)}$. These statistics clearly demonstrate the effect of a shorter k value as opposed to a longer one in the mean. In the early iterations for the $k = 5$ model, the mean has a larger spike which flattens to a bit above zero by the end of the iteration. For the larger k values, this initially increased mean gets maintained throughout a larger portion of the iteration, causing the lower values further along to have less influence on the statistics.

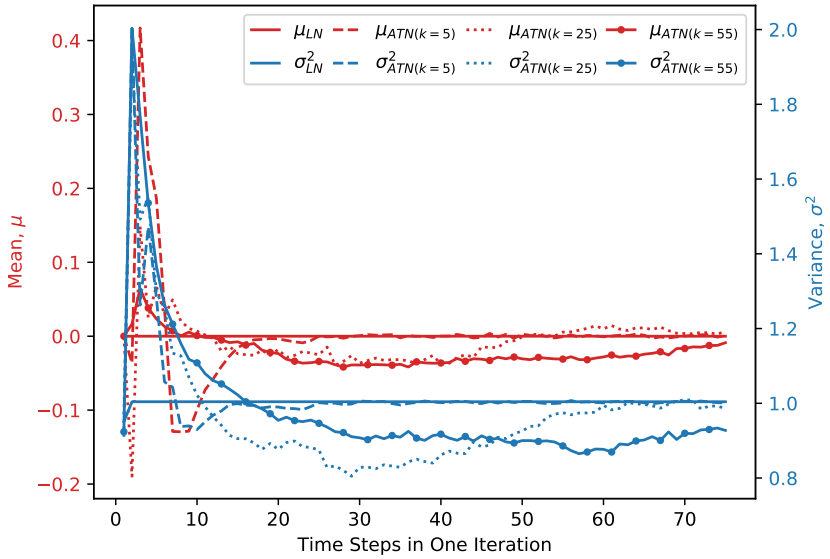


Figure 3.9: Hidden-to-Hidden

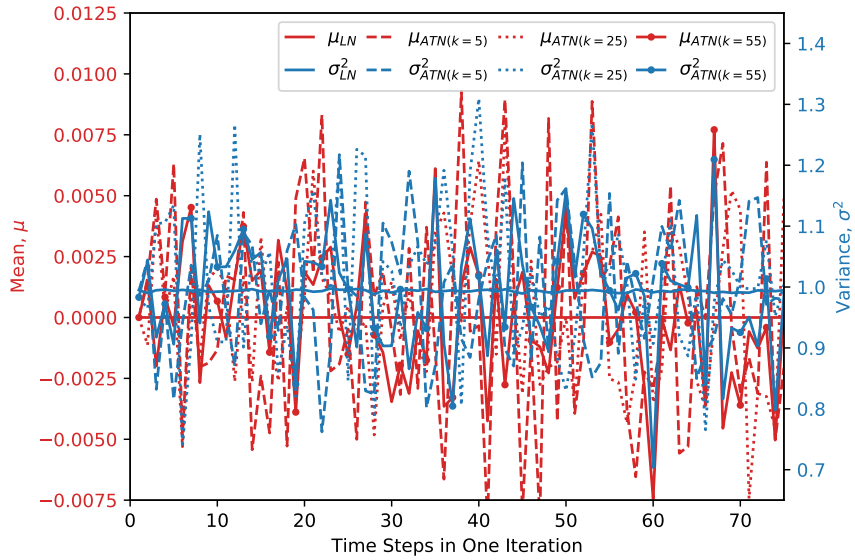


Figure 3.10: Input-to-Hidden

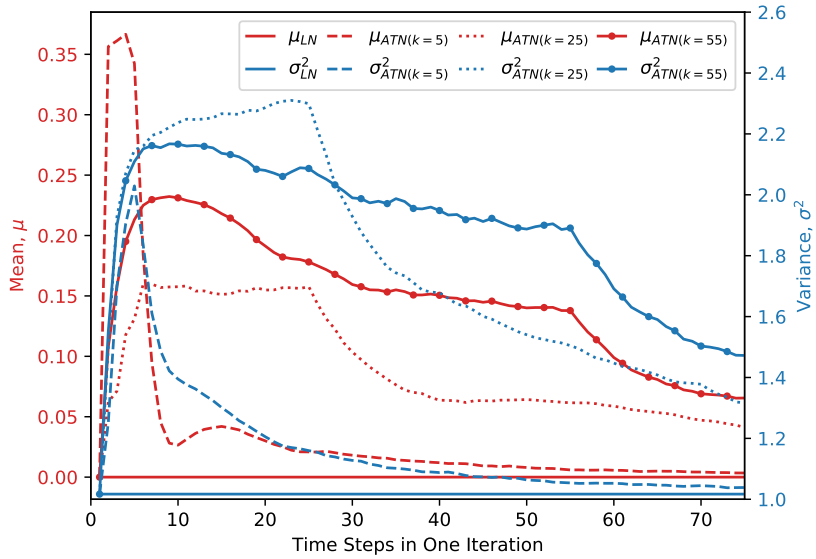


Figure 3.11: Memory Cell

3.5.3 Optimal k Value for ATN method

To highlight the importance of normalizing with respect to k time steps instead of just one or all of them, we present a study on various k values. In Figure 3.12, we present results on the Copying Problem [Hochreiter and Schmidhuber, 1997] described in Section 3.4.1 with $T = 100$. For this experiment, we have trained LSTM, LN, and three ATN(k) models with values of k being 25, 45, and 65 under the same conditions.

All ATN models perform better than both LSTM and LN. The ATN($k = 45$) model performs better than ATN($k = 25$), which should not be a surprise since the larger k value would mean we are normalizing with respect to a larger set and getting better statistics for the mean and variance; however, ATN($k = 65$) performs poorer than ATN($k = 45$) and even poorer than ATN($k = 25$) which suggests that too large a k value may actually degrade the result. This may be due to numerical difficulties in propagating derivative through k steps in ATN for a large k.

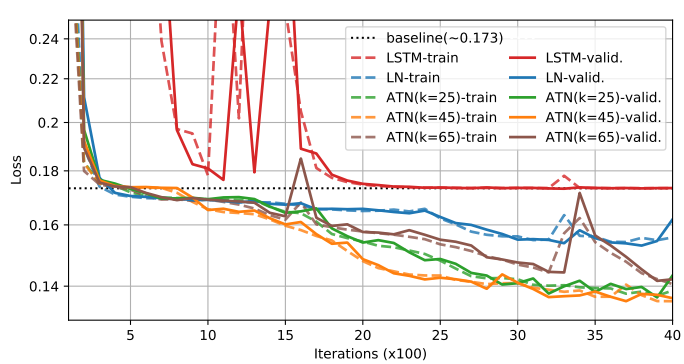


Figure 3.12: k value study in ATN method

3.6 Invariance Properties

Table 3.5: Invariance properties under different normalization methods

	BN	WN	LN	ATN-BN	ATN-LN
Weight matrix re-scaling	Yes	Yes	Yes	Yes	Yes
Weight matrix re-centering	No	No	Yes	No	No
Weight vector re-scaling	Yes	Yes	No	Yes	No
Dataset re-scaling	Yes	No	Yes	Yes	Yes
Dataset re-centering	Yes	No	No	Yes	No
Single training case re-scaling	No	No	Yes	No	Yes
Input at a single time re-scaling	No	No	Yes	No	No

In Table 3.5 we provide a summary of invariance properties for several normalization methods. The methods we compare are all commonly used throughout the field and include BN - Batch Normalization [Ioffe and Szegedy, 2015a], WN - Weight Normalization [Salimans and Kingma, 2016], LN - Layer Normalization [Ba et al., 2016]. We also include two versions of our proposed method: ATN-BN - Assorted-Time Normalization built on BN method, and ATN-LN - Assorted-Time Normalization built on LN method.

This is an expansion of Table 1 in [Ba et al., 2016]. Weight matrix re-scaling and re-centering are the adjustments of the weight matrix by multiplying a constant scaling factor or adding a constant re-scaling factor. Weight vector re-scaling is similar to weight matrix re-scaling, but only adjusts a single vector instead of the entire matrix. Dataset re-centering and re-scaling consist of changing every input example by multiplying or adding a constant. Single training case re-scaling is when the dataset adjustments are applied to just one example. Of particular interest is the invariance with respect to the scaling of an input at a single time point, which was referenced in Section 3.3. This is one of the invariance properties which LN has that its ATN adaptation does not, and we argue that this is one of the reasons that our method improves on LN.

3.7 Conclusion

In this chapter, we have introduced a method for adapting statistics-based normalization methods to recurrent neural networks to break the time invariance of the traditional normalization methods. We have presented theoretical results on the impact this method has on the model's gradients, as well as showing the preservation of invariance to the rescaling of the weight matrix. Our experiments demonstrate that our ATN-LSTM improves over LN for LSTM in both training and testing results. In

light of the popularity of LN in practical applications, our method offers an important alternative for further improving RNN performance.

Copyright© Cole M. Pospisil, 2023.

Chapter 4 NC-GRU

In this chapter, we present a method of reparameterizing the recurrent weights of the Gated Recurrent Unit sequential model to produce orthogonal weights.

4.1 Introduction

As discussed elsewhere in this work, one of the most common neural network models for sequential data is the Recurrent Neural Network (RNN) [Rumelhart et al., 1986, Hopfield, 1982]. RNNs can efficiently model sequential data through their use of repeated weights and sequences of hidden states. Training vanilla RNNs does have several obstacles [Rumelhart et al., 1986, Hopfield, 1982], one of the most studied of these is their susceptibility to vanishing and exploding gradients [Bengio et al., 1993]. In the case of vanishing gradients, the optimization algorithm is prevented from learning due to there being only extremely small changes in the model parameters. In the case of exploding gradients, the training suffers from instabilities such as divergence or oscillations.

There has been significant work done to find remedies for these issues. One popular method is the introduction of gates into the architecture of RNNs: Long Short-Term Memory (LSTM) [Hochreiter and Schmidhuber, 1997] and Gated Recurrent Units (GRU) [Cho et al., 2014]. This allows these models to pass along long-term information which makes them more resistant to the vanishing gradient problem. Despite this, GRU and LSTM models are still vulnerable to having exploding gradients.

In a similar line of research, there have been several RNN based models recently proposed which enforce the use of unitary or orthogonal matrices for the shared recurrent weights [Helfrich et al., 2018, Dorado-Rojas et al., 2020, Vorontsov et al., 2017a, Mhammedi et al., 2017, Jing et al., 2017, Arjovsky et al., 2016, Wisdom et al., 2016, Jing et al., 2019, Maduranga et al., 2018]. One of the main benefits behind these developments is the existence of a theoretical explanation for the improved performance of these models [Arjovsky et al., 2016]. A key component in these methods is the preservation of the weight matrices' orthogonal or unitary properties at every training step. There have been several different techniques for updating the recurrent weights to preserve either orthogonal or unitary properties: Cayley transforms [Helfrich et al., 2018, Maduranga et al., 2018, Helfrich and Ye, 2019, Lezcano-Casado and Martínez-Rubio, 2019], Givens rotations [Jing et al., 2017], Householder reflections [Mhammedi et al., 2017], multiplicative gradient update steps [Wisdom et al., 2016], and other similar works that have shown the efficacy of orthogonal and unitary matrices [Saxe et al., 2014, Arjovsky et al., 2016, Henaff et al., 2017, Tagare, 2011, Hyland and Gunnar, 2017, Vorontsov et al., 2017b].

In this chapter, we study the application of orthogonal matrices to one of the most widely used RNN models, the Gated Recurrent Unit (GRU) [Cho et al., 2014], and its theoretical and empirical benefits. We propose the usage of orthogonal matrices in several hidden state weights of the GRU model based on gradient analysis. We

also introduce a Neumann series-based Scaled Cayley transform for training the orthogonal weights. Our method utilizes a method of Scaled Cayley transforms, which was studied in [Helfrich et al., 2018, Maduranga et al., 2018] for training orthogonal weights for RNNs. In addition, we propose the use of a Neumann series approximation for the matrix inverse inside the Cayley transform. Such substitution performs similarly or improves on the traditional inverse (see Section 4.4) while decreasing computation time which is very desirable when working with larger networks. We call our method *Neumann-Cayley Orthogonal Gated Recurrent Unit* or *NC-GRU* for simplicity.

4.2 Related Work

In the past few decades, there has been considerable research done to improve upon the vanilla RNN model [Rumelhart et al., 1986, Hopfield, 1982]. This includes the establishment of gates [Hochreiter and Schmidhuber, 1997, Cho et al., 2014], normalization methods [Ioffe and Szegedy, 2015b, Cooijmans et al., 2017, Wu and He, 2018, Ulyanov et al., 2017, Salimans and Kingma, 2016, Ba et al., 2016, Xu et al., 2019], as well as the introduction of unitary and orthogonal matrices into the weights of the RNN architecture [Arjovsky et al., 2016, Jing et al., 2017, Mhammedi et al., 2017, Jing et al., 2019, Vorontsov et al., 2017b, Dorado-Rojas et al., 2020]. In this section, we will discuss some of the most relevant developments to our proposed method.

Unitary RNNs (uRNNs) [Arjovsky et al., 2016] presented an architecture that learns a unitary weight matrix. The construction of the recurrent weight matrix is composed of diagonal matrices, reflection matrices in the complex domain, and Fourier transforms. The uRNN model presented in [Wisdom et al., 2016] optimizes over the space of all unitary matrices rather than operating as a product of several parameterized matrices. EUNN [Jing et al., 2017] utilizes the product of unitary matrices by parameterizing its recurrent matrix with products of Givens Rotations. Also, the representation capacity of the unitary space is fully tunable and ranges from a subspace of unitary matrices to the entire unitary space.

Work by [Mhammedi et al., 2017] proposed orthogonal RNNs (oRNNs), using Householder reflections. This parameterization of the transition matrix allows for efficient training while maintaining the orthogonality of the recurrent weights throughout training. [Vorontsov et al., 2017b] proposes a weight matrix factorization by bounding the matrix norms to encourage orthogonality. This method also controls the degree of gradient expansion during backpropagation.

GORU [Jing et al., 2019] presented an RNN, which extends unitary RNNs with a gating mechanism. Orthogonality is preserved via a similar methodology to EURNN [Jing et al., 2017]. The results compared to GRU [Cho et al., 2014] are mixed and task-dependent. More recently, [Dorado-Rojas et al., 2020] presented an embedding of a linear time-invariant system containing Laguerre polynomials.

4.2.1 Gated Recurrent Unit (GRU)

In this section, we present an in-depth study of the architecture of the Gated Recurrent Unit (GRU) which was proposed in [Cho et al., 2014] as an alternative to the LSTM cell [Hochreiter and Schmidhuber, 1997].

The basic structure of a GRU cell is

$$\begin{aligned} r^{(t)} &= \sigma\left(W_r x^{(t)} + U_r h^{(t-1)} + b_r\right) \\ u^{(t)} &= \sigma\left(W_u x^{(t)} + U_u h^{(t-1)} + b_u\right) \\ c^{(t)} &= \tanh\left(W_c x^{(t)} + U_c\left(r^{(t)} \odot h^{(t-1)}\right) + b_c\right) \\ h^{(t)} &= \left(1 - u^{(t)}\right) \odot h^{(t-1)} + u^{(t)} \odot c^{(t)} \end{aligned} \tag{4.1}$$

where W_r, W_u, and W_c are input weights in $\mathbb{R}^{n\times m}$, U_r, U_u, and U_c are recurrent weights in $\mathbb{R}^{n\times n}$, and b_r, b_u and b_c are the bias parameters in $\mathbb{R}^n$.m is the dimension of the input data and n is the dimension of the hidden state. In (4.1), the activation functions σ and tanh are the sigmoid and hyperbolic tangent functions respectively, and $\odot$ is the Hadamard product. The initial hidden state h_0 is typically initialized to zero.

The primary difference between GRU and LSTM is in their treatment of long-term memory. LSTM implements its memory term as a separate channel while GRU includes it inside the hidden state, $h^{(t)}$, itself. Where LSTM has both a forget and an input gate, GRU puts this functionality in a single gate, $u^{(t)}$. If the output of $u^{(t)}$ is 1, then the forget gate is open, and the input gate must therefore be closed. Similarly, if $u^{(t)}$ is 0. This structure allows GRUs to discard random noise or insignificant information while grasping the important details.

4.2.2 Cayley Transform Orthogonal RNN

In this section, we present analysis of the effects of orthogonal matrices inside the GRU cell based on Cayley transforms [Tagare, 2011]. Some initial work on the use of Cayley transforms to preserve orthogonal weights in RNNs was presented in [Helfrich et al., 2018] as part of the scoRNN model. This model includes a skew-symmetric matrix A, which determines an orthogonal matrix W via the Scaled Cayley transform

$$W = (I + A)^{-1}(I - A)\,D, \tag{4.2}$$

where matrix D is a diagonal matrix of ones and negative ones which scales the traditional Cayley transform [Tagare, 2011]. It is proved in [Kahan, 2006] that a suitable tuning of the number of negative ones in the matrix D can prevent the problem of eigenvalues of A being negative ones, making $I + A$ singular. It further guarantees that the skew-symmetric matrix A is bounded.

[Helfrich et al., 2018] presents the following training process for the scoRNN model using Scaled Cayley transforms:

$$A^{(t+1)} = A^{(t)} - \lambda \nabla_A L\left(U_{sco}\left(A^{(t)}\right)\right) \tag{4.3}$$

$$U_{sco}^{(t+1)} = \left(I + A^{(t+1)}\right)^{-1}\left(I - A^{(t+1)}\right) D \tag{4.4}$$

where $\nabla_A L\left(U_{sco}\left(A\right)\right)$ is computed using

$$\nabla_A L\left(U_{sco}\left(A\right)\right) = V^T - V, \tag{4.5}$$

with

$$V = \left(I + A\right)^{-T} \nabla_{U_{sco}} L\left(U_{sco}\left(A\right)\right)\left(D + U_{sco}^T\right) \tag{4.6}$$

in which $\nabla_{U_{sco}} L\left(U_{sco}\left(A\right)\right)$ is computed via standard backpropogation methods.

Despite rounding errors that may accumulate due to repeated matrix multiplications, scoRNN [Helfrich et al., 2018] maintains orthogonality up to machine precision. This property helps provide significant improvements over other orthogonal/unitary RNNs for long sequences on several benchmark tasks; see [Helfrich et al., 2018] for more details.

4.3 Efficient Orthogonal Gated Recurrent Unit

We now present an efficient orthogonal GRU model. The proofs of all theoretical results can be found in [Mucllari et al., 2022].

4.3.1 Gradient Analysis of Hidden States in GRU

Since Neural Network training is based on stochastic gradient descent, gradient behavior plays a very important role in convergence, stability, and most importantly performance. When it comes to backpropagation through time for the GRU model from (4.1), the gradients of the loss function $\mathcal{L}$ with respect to intermediate hidden states, weights, and biases can be found from the respective gradients of the final hidden state h_τ, which is simplified to finding the gradient of $h^{(t)}$ with respect to $h^{(t-1)}$ for t between 1 and τ. Namely,

$$\frac{\partial \mathcal{L}}{\partial h^{(i)}} = \frac{\partial \mathcal{L}}{\partial h^{(\tau)}} \prod_{t=i+1}^{\tau} \frac{\partial h^{(t)}}{\partial h^{(t-1)}}. \tag{4.7}$$

Thus, to analyze the gradients, we need to consider the gradient of $h^{(t)}$ with respect to $h^{(t-1)}$ as well as its upper bound in the following theorem.

Theorem 3. *Let $h^{(t-1)}$ and $h^{(t)}$ be two consecutive hidden states from the GRU model stated in (4.1). Then*

$$\left\| \frac{\partial h^{(t)}}{\partial h^{(t-1)}} \right\|_2 \leq \alpha + \beta \|U_c\|_2 \tag{4.8}$$

where

$$\begin{aligned} \alpha = \delta_u \left(\max_i \left\{ [h^{(t-1)}]_i \right\} + \max_i \left\{ [c^{(t)}]_i \right\} \right) \|U_u\|_2 \\ + \max_i \left\{ (1 - [u^{(t)}]_i) \right\} \end{aligned} \tag{4.9}$$

and

$$\beta = \max_i \left\{ [u^{(t)}]_i \right\} \Big(\delta_r \|U_r\|_2 \max_i \left\{ [h^{(t-1)}]_i \right\} + \max_i \left\{ [r^{(t)}]_i \right\} \Big), \tag{4.10}$$

with constants δ_u and δ_r defined as follows:

$$\delta_u = \max_i \left\{ \left[u^{(t)}\right]_i \left(1 - \left[u^{(t)}\right]_i\right) \right\} \tag{4.11}$$

and

$$\delta_r = \max_i \left\{ \left[r^{(t)}\right]_i \left(1 - \left[r^{(t)}\right]_i\right) \right\}. \tag{4.12}$$

We can provide simple upper bounds for α, β in the following corollary of the above theorem.

Corollary 2. *For the hyperbolic tangent activation function in (4.1), we have $\delta_u, \delta_r \leq \frac{1}{4}$, $[h^{(t)}]_i \leq 1$ for any i and t as well as*

$$\alpha \leq \frac{1}{2}\|U_u\|_2 + 1 \quad \textit{and} \quad \beta \leq \frac{1}{4}\|U_r\|_2 + 1. \tag{4.13}$$

These bounds may be slightly pessimistic since it is reasonable to expect the gate elements to be near 0 or 1. In that case, the following corollary presents the relationship between the constants α and β presented in Theorem 3. Below we use the notation $x \lesssim y$ to denote that x is bounded by a quantity that is approximately equal to y.

Corollary 3. *When elements of GRU gates $u^{(t)}$ and $r^{(t)}$ are nearly either 0 or 1, then constants α and β from Theorem 3 satisfy the following inequality:*

$$\alpha + \beta \lesssim 2. \tag{4.14}$$

Moreover, if $u^{(t)}$ and $r^{(t)}$ are nearly either the zero vector or the vector of all ones, then

$$\alpha + \beta \lesssim 1. \tag{4.15}$$

4.3.2 Neumann-Cayley Orthogonal Transformation

Based on the theoretical support of Theorem 3 and Corollary 3, we propose using orthogonal weights in the hidden weights of GRU to better condition its gradients. As discussed in section 4.2, there have been several different techniques proposed to initialize and preserve orthogonal weights during training. In this work, we implement a version of the Scaled-Cayley transformation method from 4.2.2 with a key difference. The Scaled Cayley transform method requires the calculation of the inverse of $I + A^{(k)}$ in order to update the orthogonal matrix $U^{(k)}$ in (4.4). The inversion of matrices using classical numerical techniques such as LU-decomposition or the Least Squares method can be implemented and work well when the dimension of the matrix is small. However, when the dimension of the matrix is large, then these methods are

prohibitively expensive in terms of both memory and computational time. Moreover, classical methods pose a risk of overflowing, entirely preventing convergence. We propose to solve this complication through the use of Neumann Series to approximate the inverse of $I + A^{(k)}$.

To derive the Neumann series approximation for the inverse of $I + A^{(k)}$ in (4.4), we consider the following:

$$\left(I + A^{(t)}\right)^{-1} = \left(I + A^{(t-1)} - \delta A^{(t)}\right)^{-1} \tag{4.16}$$

$$= \left(I - \left(I + A^{(t-1)}\right)^{-1} \delta A^{(t)}\right)^{-1} \left(I + A^{(t-1)}\right)^{-1} \tag{4.17}$$

$$= \left(\sum_{i=0}^{\infty} \left(\left(I + A^{(t-1)}\right)^{-1} \delta A^{(t)}\right)^{i}\right) \left(I + A^{(t-1)}\right)^{-1} \tag{4.18}$$

where $\delta A^{(t)} := opt_A \left(\nabla_A \mathcal{L} = V^{(t)^T} - V^{(t)}\right)$, here opt_A includes a learning rate λ inside of it. Note that the equality in Equations (4.17) and (4.18) relies on the assumption that $\| \left(I + A^{(t-1)}\right)^{-1} \delta A^{(t)} \| < 1$ for some operator norm $\| \cdot \|$; see [Demmel, 1997] for more details.

In our experiments, we have considered the first and the second-order Neumann series approximations for (4.18) with two ($i = 0, 1$) and three ($i = 0, 1, 2$) terms respectively. The model's performance is slightly improved with the use of the second-order approximation, at a marginally increased cost in computational time. For the second-order approximation, the error is of the order $\mathcal{O}\left(\left(\left(I + A^{(k-1)}\right)^{-1} \delta A^{(k)}\right)^{3}\right)$. Even though this error is quite small, there is a chance that the approximation errors will accumulate over time and the weight matrix will lose orthogonality. To avoid this, we reset orthogonality occasionally through an explicit computation of the matrix inverse using a factorization method at the beginning of each epoch. It might be necessary to do it more often in the earlier stages of training due to more fluctuations in the gradients.

Algorithm 8 Update Rule for Orthogonal Weight U

Given: D, $A^{(0)}$, $U^{(0)}$, $\nabla_U \mathcal{L}\left(U^{(0)}\left(A^{(0)}\right)\right)$, opt_A
Define: $\tilde{A}^{(0)} := \left(I + A^{(0)}\right)^{-1}$
for $t = 1, 2, \ldots$ **do**
  $V^{(t)} := \tilde{A}^{(t-1)^T} \nabla_U \mathcal{L}\left(U^{(t-1)}\left(A^{(t-1)}\right)\right)\left(D + U^{(t-1)^T}\right)$
  $\delta A^{(t)} := opt_A \left(\nabla_A \mathcal{L} = V^{(t)^T} - V^{(t)}\right)$
  $A^{(t)} := A^{(t-1)} - \delta A^{(t)}$
  $\tilde{A}^{(t)} := \left(I + \tilde{A}^{(t-1)} \delta A^{(t)} + \left(\tilde{A}^{(t-1)} \delta A^{(t)}\right)^2\right) \tilde{A}^{(t-1)}$
  $U^{(t)} := \tilde{A}^{(t)}\left(I - A^{(t)}\right) D$
end for

Algorithm 7 outlines the Neumann-Cayley Orthogonal Transformation method for training the weight A and updating the corresponding orthogonal weight U. The weight $A^{(0)}$ is initialized to be skew-symmetric as in [Helfrich et al., 2018] which is based on the idea from [Henaff et al., 2017]. We apply the Cayley transform to $A^{(0)}$ in order to initialize $U^{(0)}$. Algorithm 7 includes opt_A, which is a standard optimizer such as stochastic gradient descent (SGD), RMSProp [Tieleman and Hinton, 2012], or Adam [Kingma and Ba, 2014], that recieves $\nabla_A \mathcal{L} = V^{(k)^T} - V^{(k)}$ as an input. An important feature to observe here is that the skew-symmetric property of the weight A and its gradient are preserved under such an optimizer.

4.3.3 Neumann-Cayley Orthogonal GRU (NC-GRU)

In this section, we introduce our Neumann-Cayley Orthogonal GRU (NC-GRU) model that utilizes the proposed Neumann-Cayley Orthogonal Transform.

The structure of the NC-GRU cell is

$$\begin{aligned} r^{(t)} &= \sigma\left(W_r x^{(t)} + U_r(A_r) h^{(t-1)} + b_r\right) \\ u^{(t)} &= \sigma\left(W_u x^{(t)} + U_u h^{(t-1)} + b_u\right) \\ c^{(t)} &= \Phi\left(W_c x^{(t)} + U_c(A_c)\left(r^{(t)} \odot h^{(t-1)}\right)\right) \\ h^{(t)} &= \left(1 - u^{(t)}\right) \odot h^{(t-1)} + u^{(t)} \odot c^{(t)} \end{aligned} \tag{4.19}$$

here σ - sigmoid function, $\odot$ - Hadamard product, and Φ - modReLU function defined in [Arjovsky et al., 2016] as

$$\Phi(x) := \text{modReLU}(x) := \text{sgn}(x) \cdot \text{ReLU}\left(|x| + b\right) \tag{4.20}$$

with trainable bias b.

From empirical studies, we have determined that the best performance is generally achieved using orthogonality in the U_c and U_r hidden weights. Similarly to the original GRU cell, W_r, W_u, W_c, U_u, b_r, b_u, b, are trainable parameters which are updated using standard backpropagation algorithms such as Stochastic Gradient Descent (SGD), RMSProp [Tieleman and Hinton, 2012], or Adam [Kingma and Ba, 2014] while U_r, A_r, U_c, and A_c are trained using Algorithm 7.

The use of orthogonal weights leads to a better-conditioned gradient as shown in the following Corollary.

Corollary 4. *Let h_{t-1} and h_t be two consecutive hidden states from the NC-GRU model defined in (4.19). Then $\|U_r\|_2 = \|U_c\|_2 = 1$ and if elements of the gates $u^{(t)}$ and $r^{(t)}$ are nearly 0 or 1, then the following inequality is satisfied:*

$$\left\| \frac{\partial h_t}{\partial h_{t-1}} \right\|_2 \lesssim 2. \tag{4.21}$$

Furthermore, if $u^{(t)}$ and $r^{(t)}$ are nearly either zero vector or vector of all ones,

$$\left\| \frac{\partial h^{(t)}}{\partial h^{(t-1)}} \right\|_2 \lesssim 1. \tag{4.22}$$

4.4 Experiments

We have performed various experiments to demonstrate the robustness and efficiency of our NC-GRU method. In this work, we present a Language Modeling task for the word-level Penn TreeBank dataset. For additional experiments, see [Mucllari et al., 2022].

All the experiments were trained with equal numbers of trainable parameters, a setting referred to as parameter-matching architecture.

4.5 Word Level Penn TreeBank

We also tested our proposed Neumann-Cayley method on the word level Penn TreeBank dataset [Marcus et al., 1993]. The dataset takes the same underlying corpus as the character-level task, but with tokens representing words instead of characters. This results in a smaller dataset with a larger vocabulary size, with 888 thousand, 70 thousand, and 79 thousand words as training, validation, and testing sets, and a vocabulary of 10 thousand words.

Implementation Details: We trained NC-GRU with three layers with dimensions (400, 1150, 400). We have a learning rate set to 5×10^{-4} for both the A matrix and the rest of the model, optimized using Adam [Kingma and Ba, 2014]. The dropout after the NC-GRU cell has a coefficient of 0.4, the embedding layer dropout has a coefficient of 0.4, and the output dropout has a coefficient of 0.25. The number of negative ones for matrix D for each layer was 50.

Results: Results were evaluated using the perplexity (PPL) metric and are shown in table 4.1. We show improved performance over both baseline GRU and LSTM models.

Table 4.1: Word Level PTB Results: Evaluated perplexity (PPL) for every model. * - result obtained from our experiments; † - result quoted from [Bai et al., 2018]

	PPL ↓
LSTM	$78.93^\dagger$
GRU	$92.48^\dagger$ (80.73^*)
scoRNN	123.90^*
NC-GRU(U_c) (**ours**)	77.00

4.6 Ablation Studies

We performed a number of ablation studies to help provide empirical justification for the use of the Neumann series, orthogonal matrices, and scaled-Cayley transforms. These include a study on the sharpness and computational costs of the Neumann approximation of the inverse compared to standard methods, a study on the effect of orthogonalizing different sets of weights in the GRU, and verifying that the necessary

condition for the use of the Neumann approximation is satisfied in our experimental settings. These studies can be found in [Mucllari et al., 2022]

4.7 Conclusion

In this chapter, we have presented an analysis of the Gated Recurrent Unit (GRU) model's gradient behavior. Based on this, we proposed the Neumann-Cayley Gated Recurrent Unit model. Our model incorporates orthogonal weights in the hidden states of the GRU model, which are trained using a noval method of Neumann-Cayley transformation which maintains orthogonality in those weights. We have conducted a series of experiments showing the benefits of our proposed method, outperforming GRU, LSTM, and scoRNN models.

Bibliography

[Arjovsky et al., 2016] Arjovsky, M., Shah, A., and Bengio, Y. (2016). Unitary evolution recurrent neural networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1120–1128. JMLR.org.

[Ba et al., 2016] Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization.

[Bai et al., 2018] Bai, S., Kolter, J. Z., and Koltun, V. (2018). An empirical evaluation of generic convolutional and recurrent networks for sequence modeling.

[Bengio et al., 1993] Bengio, Y., Frasconi, P., and Simard, P. (1993). The problem of learning long-term dependencies in recurrent networks. In *Proceedings of 1993 IEEE International Conference on Neural Networks (ICNN '93)*, pages 1183–1195, San Francisco, CA. IEEE Press.

[Cho et al., 2014] Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation.

[Cooijmans et al., 2017] Cooijmans, T., Ballas, N., Laurent, C., Gülçehre, Ç., and Courville, A. C. (2017). Recurrent batch normalization. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

[Demmel, 1997] Demmel, J. W. (1997). *Applied Numerical Linear Algebra*. SIAM.

[Dorado-Rojas et al., 2020] Dorado-Rojas, S. A., Vinzamuri, B., and Vanfretti, L. (2020). Orthogonal laguerre recurrent neural networks. *34th Conference on Neural Information Processing Systems*.

[Foerster et al., 2016] Foerster, J. N., Gilmer, J., Chorowski, J., Sohl-Dickstein, J., and Sussillo, D. (2016). Input switched affine networks: An rnn architecture designed for interpretability.

[Gitman and Ginsburg, 2017] Gitman, I. and Ginsburg, B. (2017). Comparison of batch normalization and weight normalization algorithms for the large-scale image classification.

[Hamilton et al., 2018] Hamilton, W. L., Ying, R., and Leskovec, J. (2018). Inductive representation learning on large graphs.

[Helfrich et al., 2018] Helfrich, K., Willmott, D., and Ye, Q. (2018). Orthogonal Recurrent Neural Networks with Scaled Cayley Transform. *In Proceedings of ICML 2018, Stockholm, Sweden, PMLR 80.*

[Helfrich and Ye, 2019] Helfrich, K. and Ye, Q. (2019). Eigenvalue normalized recurrent neural networks for short term memory.

[Henaff et al., 2017] Henaff, M., Szlam, A., and LeCun, Y. (2017). Recurrent orthogonal networks and long-memory tasks. In *Proceedings of the 33rd International Conference on Machine Learning (ICML 2017)*, volume 48.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

[Hopfield, 1982] Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558.

[Hyland and Gunnar, 2017] Hyland, S. L. and Gunnar, R. (2017). Learning unitary operators with help from u(n). In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAI 2017)*, pages 2050–2058, San Francisco, CA.

[Ioffe and Szegedy, 2015a] Ioffe, S. and Szegedy, C. (2015a). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France. PMLR.

[Ioffe and Szegedy, 2015b] Ioffe, S. and Szegedy, C. (2015b). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

[Jing et al., 2019] Jing, L., Gulcehre, C., Peurifoy, J., Shen, Y., Tegmark, M., Soljacic, M., and Bengio, Y. (2019). Gated orthogonal recurrent units: On learning to forget. *Neural Computation*, 31(4):765–783.

[Jing et al., 2017] Jing, L., Shen, Y., Dubcek, T., Peurifoy, J., Skirlo, S., LeCun, Y., Tegmark, M., and Soljačić, M. (2017). Tunable efficient unitary neural networks (eunn) and their application to rnns. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 1733–1741. JMLR.org.

[Kahan, 2006] Kahan, W. (2006). Is there a small skew cayley transform with zero diagonal? *Linear Algebra and its Applications*, 417(2):335–341. Special Issue in honor of Friedrich Ludwig Bauer.

[Kingma and Ba, 2014] Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Kipf and Welling, 2017] Kipf, T. N. and Welling, M. (2017). Semi-supervised classification with graph convolutional networks.

[Lange et al., 2021] Lange, S., Helfrich, K., and Ye, Q. (2021). Batch normalization preconditioning for neural network training.

[Le et al., 2015] Le, Q. V., Jaitly, N., and Hinton, G. E. (2015). A simple way to initialize recurrent networks of rectified linear units.

[Lecun et al., 1998] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[Lezcano-Casado and Martínez-Rubio, 2019] Lezcano-Casado, M. and Martínez-Rubio, D. (2019). Cheap orthogonal constraints in neural networks: A simple parametrization of the orthogonal and unitary group. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3794–3803. PMLR.

[Maduranga et al., 2018] Maduranga, K. D. G., Helfrich, K. E., and Ye, Q. (2018). Complex unitary recurrent neural networks using scaled cayley transform.

[Marcus et al., 1993] Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.*, 19(2):313–330.

[Merity et al., 2018] Merity, S., Keskar, N. S., and Socher, R. (2018). Regularizing and optimizing LSTM language models. In *International Conference on Learning Representations*.

[Mhammedi et al., 2017] Mhammedi, Z., Hellicar, A. D., Rahman, A., and Bailey, J. (2017). Efficient orthogonal parametrisation of recurrent neural networks using householder reflections. In *Proceedings of ICML 2017*, volume 70, pages 2401–2409. PMLR.

[Mucllari et al., 2022] Mucllari, E., Zadorozhnyy, V., Pospisil, C., Nguyen, D., and Ye, Q. (2022). Orthogonal gated recurrent unit with neumann-cayley transformation.

[Polyak, 1964] Polyak, B. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). *Learning Internal Representations by Error Propagation*, page 318–362. MIT Press.

[Saad, 2003] Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second edition.

[Salimans and Kingma, 2016] Salimans, T. and Kingma, D. P. (2016). Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In Lee, D., Sugiyama, M., Luxburg, U., Guyon, I., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc.

[Santurkar et al., 2018] Santurkar, S., Tsipras, D., Ilyas, A., and Madry, A. (2018). How does batch normalization help optimization?

[Saxe et al., 2014] Saxe, A. M., McClelland, J. L., and Ganguli, S. (2014). Exact solutions to nonlinear dynamics of learning in deep linear neural networks.

[Tagare, 2011] Tagare, H. D. (2011). Notes on optimization on stiefel manifolds. *Yale*. "Technical report, Yale University".

[Tieleman and Hinton, 2012] Tieleman, T. and Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2).

[Ulyanov et al., 2017] Ulyanov, D., Vedaldi, A., and Lempitsky, V. (2017). Instance normalization: The missing ingredient for fast stylization.

[Vorontsov et al., 2017a] Vorontsov, E., Trabelsi, C., Kadoury, S., and Pal, C. (2017a). On orthogonality and learning recurrent networks with long term dependencies.

[Vorontsov et al., 2017b] Vorontsov, E., Trabelsi, C., Kadoury, S., and Pal, C. (2017b). On orthogonality and learning recurrent networks with long term dependencies. *arXiv preprint arXiv:1702.00071*.

[Wisdom et al., 2016] Wisdom, S., Powers, T., Hershey, J., Le Roux, J., and Atlas, L. (2016). Full-capacity unitary recurrent neural networks. In *Advances in Neural Information Processing Systems 29*, pages 4880–4888.

[Wu and He, 2018] Wu, Y. and He, K. (2018). Group normalization. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

[Xu et al., 2019] Xu, J., Sun, X., Zhang, Z., Zhao, G., and Lin, J. (2019). Understanding and improving layer normalization. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 32, page 4381–4391. Curran Associates, Inc.

[Yang et al., 2016] Yang, Z., Cohen, W. W., and Salakhutdinov, R. (2016). Revisiting semi-supervised learning with graph embeddings.

Vita

Cole Morgan Pospisil

Place of Birth:

- Charlotte, NC

Education:

- University of Kentucky, Lexington, KY
  M.S. in Mathematics, May 2020
- Vanderbilt University, Nashville, TN
  B.A. in Mathematics, May 2017

Professional Positions:

- Graduate Teaching Assistant, University of Kentucky Fall 2017–Spring 2021

Honors

- SMART Scholarship, Department of Defense

Publications & Preprints:

- "Breaking Time Invariance: Assorted-Time Normalization for RNNs", Cole Pospisil, Vasily Zadorozhnyy, and Qiang Ye (Preprint)
- "Orthogonal Gated Recurrent Unit with Neumann-Cayley Transformation", Edison Mucllari, Vasily Zadorozhnyy, Cole Pospisil, Duc Nguyen, and Qiang Ye (Preprint)