Université de Montréal

On Challenges in Training Recurrent Neural Networks

par

Sarath Chandar Anbil Parthipan

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des arts et des sciences
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en informatique

Composition du jury

Président-rapporteur:	Jian-Yun Nie
Directeur de recherche:	Yoshua Bengio
Co-directeur:	Hugo Larochelle
Membre du jury:	Pascal Vincent
Examinateur externe:	Alex Graves

November, 2019

Résumé

Dans un problème de prédiction à multiples pas discrets, la prédiction à chaque instant peut dépendre de l'entrée à n'importe quel moment dans un passé lointain. Modéliser une telle dépendance à long terme est un des problèmes fondamentaux en apprentissage automatique. En théorie, les Réseaux de Neurones Récurrents (RNN) peuvent modéliser toute dépendance à long terme. En pratique, puisque la magnitude des gradients peut croître ou décroître exponentiellement avec la durée de la séquence, les RNNs ne peuvent modéliser que les dépendances à court terme. Cette thèse explore ce problème dans les réseaux de neurones récurrents et propose de nouvelles solutions pour celui-ci.

Le chapitre 3 explore l'idée d'utiliser une mémoire externe pour stocker les états cachés d'un réseau à Mémoire Long et Court Terme (LSTM). En rendant l'opération d'écriture et de lecture de la mémoire externe discrète, l'architecture proposée réduit le taux de décroissance des gradients dans un LSTM. Ces opérations discrètes permettent également au réseau de créer des connexions dynamiques sur de longs intervalles de temps. Le chapitre 4 tente de caractériser cette décroissance des gradients dans un réseau de neurones récurrent et propose une nouvelle architecture récurrente qui, grâce à sa conception, réduit ce problème. L'Unité Récurrente Non-saturante (NRUs) proposée n'a pas de fonction d'activation saturante et utilise la mise à jour additive de cellules au lieu de la mise à jour multiplicative.

Le chapitre 5 discute des défis de l'utilisation de réseaux de neurones récurrents dans un contexte d'apprentissage continuel, où de nouvelles tâches apparaissent au fur et à mesure. Les dépendances dans l'apprentissage continuel ne sont pas seulement contenues dans une tâche, mais sont aussi présentes entre les tâches. Ce chapitre discute de deux problèmes fondamentaux dans l'apprentissage continuel: (i) l'oubli catastrophique d'anciennes tâches et (ii) la capacité de saturation du réseau. De plus, une solution est proposée pour régler ces deux problèmes lors de l'entraînement d'un réseau de neurones récurrent.

Mots clés: Réseaux de Neurones Récurrents, Dépendances à long terme, Unité Récurrente Non-saturante, Réseaux de Neurones à Mémoire Augmentée, Machine Neuronale de Turing, LSTMs, connexions dynamiques, apprentissage continuel, oubli catastrophique, capacité de saturation.

Summary

In a multi-step prediction problem, the prediction at each time step can depend on the input at any of the previous time steps far in the past. Modelling such long-term dependencies is one of the fundamental problems in machine learning. In theory, Recurrent Neural Networks (RNNs) can model any long-term dependency. In practice, they can only model short-term dependencies due to the problem of vanishing and exploding gradients. This thesis explores the problem of vanishing gradient in recurrent neural networks and proposes novel solutions for the same.

Chapter 3 explores the idea of using external memory to store the hidden states of a Long Short Term Memory (LSTM) network. By making the read and write operations of the external memory discrete, the proposed architecture reduces the rate of gradients vanishing in an LSTM. These discrete operations also enable the network to create dynamic skip connections across time. Chapter 4 attempts to characterize all the sources of vanishing gradients in a recurrent neural network and proposes a new recurrent architecture which has significantly better gradient flow than state-of-the-art recurrent architectures. The proposed Non-saturating Recurrent Units (NRUs) have no saturating activation functions and use additive cell updates instead of multiplicative cell updates.

Chapter 5 discusses the challenges of using recurrent neural networks in the context of lifelong learning. In the lifelong learning setting, the network is expected to learn a series of tasks over its lifetime. The dependencies in lifelong learning are not just within a task, but also across the tasks. This chapter discusses the two fundamental problems in lifelong learning: (i) catastrophic forgetting of old tasks, and (ii) network capacity saturation. Further, it proposes a solution to solve both these problems while training a recurrent neural network.

Keywords: Recurrent Neural Networks, Long-term Dependencies, Non-saturating Recurrent Units, Memory Augmented Neural Networks, Neural Turing Machines, LSTMs, skip connections, lifelong learning, catastrophic forgetting, capacity saturation.

Contents

List of Figures

List of Tables

List of Abbreviations

BPC	Bits per character
BPTT	Backpropagation Through Time
CIFAR	Canadian Institute for Advanced Research
DL	Deep Learning
D-NTM	Dynamic Neural Turing Machine
EUNN	Efficient Unitary Neural Network
EURNN	Efficient Unitary Recurrent Neural Network
EWC	Elastic Weight Consolidation
FFT	Fast Fourier Transform
GEM	Gradient Episodic Memory
GORU	Gated Orthogonal Recurrent Unit
GRU	Gated Recurrent Unit
HAT	Hard Attention Target
HMM	Hidden Markov Model
iCaRL	Incremental Classifier and Representation Learning
IMM	Incremental Moment Matching
JANET	Just Another NETwork
KL divergence	Kullback-Leibler divergence
LRU	Least Recently Used
LSTM	Long Short Term Memory
LwF	Learning without Forgetting
MANN	Memory Augmented Neural Network

ML	Machine Learning
MLP	Multi-Layer Perceptron
MNIST	Mixed National Institute of Standards and Technology
NMC	Nearest Mean Classifier
NOP	No Operation
NRU	Non-saturating Recurrent Unit
NTM	Neural Turing Machine
psMNIST	Permuted Sequential MNIST
PTB	Penn Tree Bank
QP	Quadratic Programming
REINFORCE	REward Increment = Nonnegativev Factor times Offset Reinforcement times Characteristic Eligibility
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SNLI	Stanford Natural Language Inference
SOTA	State-of-the-art
SRU	Statistical Recurrent Unit
SSMNIST	Sequential Stroke MNIST
TARDIS	Temporal Automatic Relation Discovery In Sequences

To my mother Premasundari.
For all the sacrifices that she has made!
She is the reason I'm where I am today!!

Acknowledgments

August 2014. I was visiting Hugo Larochelle's lab at the University of Sherbrooke. Hugo and I were discussing my options for a PhD. "If you want to do a Ph.D. in Deep Learning, you have to do it with Yoshua," Hugo had said. I applied only to the University of Montreal and I got an offer to do my Ph.D. with Yoshua Bengio and Hugo Larochelle. I accepted the offer and the next four years were a wonderful journey. If I have to go back and redo my Ph.D., I would do it exactly in the same way!

I should start my acknowledgements by thanking Hugo Larochelle. I know Hugo since NeurIPS 2013. Not only did he show me the right path to do my Ph.D. with Yoshua, but he also agreed to co-supervise me. In the last four years, Hugo has been my adviser, mentor, teacher, and well-wisher. Even though this thesis does not include any of my joint work with Hugo, this thesis is not possible without his advice and mentoring.

My next thanks are to Yoshua Bengio. Yoshua gave me the independence to do my research. However, he was always there every time I got stuck and our discussions always helped me progress. He always encouraged me to do good science and not run after publications. Yoshua and Hugo together are the best combination of advisers one could ask for!

I would like to thank my primary collaborators without whom this thesis is not possible: Caglar Gulcehre, Chinnadhurai Sankar, and Shagun Sodhani. Chapters 3, 4, and 5 in this thesis are joint work with Caglar, Chinna, and Shagun respectively.

I am grateful to be part of Mila, which is a very vibrant and dynamic research lab that one can think of. I would like to thank the following Mila faculty members for several technical and career-related discussions over the last four years: Laurent Charlin, Aaron Courville, Will Hamilton, Simon Lacoste-Julien, Guillaume Lajoie, Roland Memisevic, Ioannis Mitliagkas, Chris Pal, Prakash Panangaden, Liam Paull, Joelle Pineau, Doina Precup, Alain Tapp, Pascal Vincent, Guillaume Rabusseau. I would like to thank Dzmitry Bahdanau, Harm de Vries, William Fedus, Prasanna Parthasarathi, Iulian Vlad Serban, Chiheb Trabelsi, and Eugene Vorontsov for several interesting conversations. I would like to thank my *lecoinnoir* team for all the fun in the workplace. Special thanks to Laura Ball for keeping our corner green!

I would like to thank the Google Brain Team in Montreal for hosting me as a student research scholar for the last 2 years. I have had several stimulating research discussions with Hugo Larochelle, Marc Bellemare, Laurent Dnih, Ross Goroshin, Danny Tarlow and Shibl Mourad (DeepMind) in the Google office. A special thanks to my manager Natacha Mainville for always making sure that I never had any blocks in my research and work at Google.

I would like to thank everyone who participated in Hugo's weekly meetings: David Bieber, Liam Fedus, Disha Srivastava, and Danny Tarlow. Despite being on a friday, these meetings were always fun! I would like to thank the following external researchers for several interesting research discussions and career-related advice: Kyunghyun Cho, Orhan Firat, Marlos Machado, Karthik Narasimhan, Siva Reddy. During my Ph.D., I was lucky to mentor Mohammad Amini, Vardhaan Pahuja, Gabriele Prato, Shagun Sodhani, and Nithin Vasishta. I would like to thank all of them for the knowledge that I gained through mentoring them.

In the last seven years of my research, I was extremely lucky to have several good collaborators. Apart from the ones mentioned above, I would like to thank the following: Ghulam Ahmed Ansari, Alex Auvolat, Sungjin Ahn, Nolan Bard, Michael Bowling, Neil Burch, Alexandre de Brebisson, Murray Campbell, Mathieu Duchesneau, Vincent Dumoulin, Iain Dunning, Jakob N. Foerster, Jie Fu, Alberto Garcia-Duran, Revanth Gengireddy, Mathieu Germain, Edward Hughes, Samira Kahou, Nan Rosemary Ke, Khimya Ketarpal, Taesup Kim, Marc Lanctot, Stanislas Lauly, Zhouhan Lin, Sridhar Mahadevan, Vincent Michalski, Subhodeep Moitra, Alexandre Nguyen, Emilio Parisotto, Prasanna Parthasarathi, Michael Pieper, Olivier Pietquin, Janarthanan Rajendran, Sai Rajeshwar, Vikas Raykar, Subhendu Rongali, Amrita Saha, Karthik Sankaranarayanan, Francis Song, Jose M. R. Sotelo, Florian Strub, Dendi Suhubdy, Sandeep Subramanian, Gerry Tesauro, Saizheng Zhang.

I would like to thank Simon Lacoste-Julien for accepting me to TA for his graphical models course for 2 years. In my Ph.D., I also got the opportunity to teach the Machine Learning course at McGill twice. I would like to thank all the 300+ students who took the course. I learned a lot by teaching this course. Special thanks to all my TAs for supporting me to teach such a large scale course.

Throughout my Ph.D., I was fortunate to hold several fellowships and scholarships. I was supported by an FQRNT-PBEEE fellowship by the Quebec Government for the second and third year. I was supported by an IBM Ph.D, fellowship for the fourth and fifth years. I also received the Antidote scholarship for NLP by Druide Informatique. Google also supported my Ph.D. by giving me a student research scholar position which allowed me to work part-time at Google Brain Montreal. I would like to thank all these institutions for providing financial support which helped me to do my research. Special thanks to all the staff members at

Mila who made my Ph.D. life much easier: Frédéric Bastien, Myriam Cote, Jocelyne Etienne, Mihaela Ilie, Simon Lefrancois, Julie Mongeau, and Linda Peinthiere.

I would like to thank my undergrad adviser Susan Elias for introducing me to research, my Master's adviser Ravindran Balaraman for introducing me to Machine Learning, my long-term collaborator and mentor Mitesh Khapra for introducing me to Deep Learning.

This thesis would not have been possible without the support of my friends and family! I will start by acknowledging my room-mate for the first three years: Chinnadhuari Sankar. We had technical discussions even while cooking and driving. Our pair-programming in late nights has resulted in several interesting projects including the NRU (presented in this thesis). Special thanks to Gaurav Isola, Prasanna Parthasarathi, and Shagun Sodhani for being good friends, and well-wishers, and supporting me whenever I face any personal issues. Thanks to Igor Kozlov, Ritesh Kumar, Disha Srivastava, Sandeep Subramaniam, Nithin Vasishta, and Srinivas Venkattaramanujam for all the fun time in Montreal. I would like to thank my long-time friends Praveen Muthukumaran and Nivas Narayanasamy for supporting me through thick and thin. I would like to thank my music teacher Divya Iyer for accommodating my busy research life and teaching me music whenever I find the time. Music, for me, is a soul recharging experience!

Ph.D. is a challenging endeavour. It is the support of my family which encouraged me to keep going. I would like to thank Sibi Chakravarthi for everything he has done for me. Life would be boring without our silly fights! I also would like to thank Chitra aunty for her support and advice. She always treats me the same way as Sibi. I would like to thank Mamal Amini for all his love and care. Montreal has given me a Ph.D. and a career, but you are the best gift that Montreal gave me. Thank you for being such an awesome friend and brother. Thank you for making sure I go to the gym even if there is a deadline the next day! I would like to thank my sister Parani for her constant support. She always cares for me and encourages me every time I feel down. I would like to thank my wife Sankari for supporting my dreams and understanding every night I was working in front of the desktop (including the night I am writing this acknowledgement section!). Thank you for your love and affection. I would like to thank my mom Premasundari for everything. I am the world for her. This thesis is rightly dedicated to her. Finally, I would like to thank my father Parthipan for everything. He always wanted to see me as a scientist. Now I can finally say that I have achieved your dreams for me dad! You are still living in our memories and I know you will be feeling proud about me right now!

1 Introduction

Designing general-purpose learning algorithms is one of the long-standing goals of artificial intelligence and machine learning. The success of Deep Learning (DL) (LeCun et al., 2015; Goodfellow et al., 2016) demonstrated *gradient descent* as one such powerful learning algorithm. However, it shifted the attention of the machine learning research community from the search for a general-purpose learning algorithm to the search for a general-purpose model architecture. This thesis considers one such general-purpose model architecture class: Recurrent Neural Networks (RNNs). RNNs have become the de-facto models for sequential prediction problems. Variants of RNNs are used in many natural language processing applications like speech recognition (Bahdanau et al., 2016), language modeling (Merity et al., 2017), machine translation (Wu et al., 2016), and dialogue systems (Serban et al., 2017). RNNs are also used for sequential decision-making problems (Hausknecht and Stone, 2015).

While RNNs are more suitable for sequential prediction problems, they can also be useful computation models for one-step prediction problems. A one-step prediction problem like image classification could still benefit from recurrent reasoning steps over the available information to make better predictions as demonstrated by Mnih et al. (2014). Thus any improvements to RNNs should have a significant impact on the general problem of *prediction*. This thesis proposes several architectural and algorithmic improvements in training recurrent neural networks.

RNNs, even though a powerful class of model architectures, are difficult to train due to the problem of vanishing and exploding gradients (Hochreiter, 1991; Bengio et al., 1994). While training an RNN using gradient descent, gradients could vanish or explode as the length of the sequence increases. While exploding gradients destabilize the training, vanishing gradients prohibit the model from learning long-term dependencies that exist in the data. Learning long-term dependencies is one of the central challenges for machine learning. Designing a general-purpose recurrent architecture which has no vanishing or exploding gradients while maintaining all its expressivity remains an open question. The first half of this thesis attempts to answer this question.

Moving away from the single-task setting where one trains a model to perform only one task, the second half of the thesis considers a multi-task lifelong learning

setting. In a lifelong learning setting, the network has to learn a series of tasks rather than just one task. This is beneficial since the network can transfer knowledge from one task to another and hence learn new tasks faster than learning them from scratch. However, this comes with additional challenges. When we train an RNN to learn multiple tasks it might forget the previous tasks while learning the new task and hence its performance in previous tasks might degrade. This is known as *catastrophic forgetting* (McCloskey and Cohen, 1989). Also note that RNNs are finite capacity parametric models. When forced to learn more tasks than its capacity allows, an RNN tends to unlearn previous tasks to free the capacity to learn new tasks. We call this *capacity saturation*. Note that capacity saturation and catastrophic forgetting are related issues. While capacity saturation leads to catastrophic forgetting, it is not the only source of catastrophic forgetting. Another source is the gradient descent algorithm itself which forces the parameter to move to a different region in the parameter space to learn the new task better. The thesis attempts to provide solutions to tackle these additional challenges that would arise in a lifelong learning setting.

1.1 Contributions

The key contributions of this thesis are as follows:

— **Dynamic skip-connections to avoid vanishing gradients.** In a typical recurrent neural network, the state of the network at each time step is a function of the state of the network at previous time step. This creates a linear chain that the gradients has to pass through. We propose a new recurrent architecture called "TARDIS" which learns to create dynamic skip connections to previous time steps so that the gradients can pass directly to a previous time step without passing through all the intermediate time steps. This significantly reduces the rate at which gradients vanish (Chapter 3).

— **A recurrent architecture with no vanishing gradients.** In this thesis, we propose a new recurrent architecture which does not have any vanishing gradients by construction. We call the architecture "Non-saturating Recurrent Unit" (NRU) (Chapter 4).

— **RNNs for lifelong learning.** We study the problem of using RNNs for lifelong learning. While most of the existing work on lifelong learning focused on catastrophic forgetting, this work highlights that the issue of capacity saturation is also important and proposes a hybrid learning algorithm which would tackle both catastrophic forgetting and capacity saturation. Specifically, we

extend the individual solutions for these problems from the feed-forward literature and also merge them to tackle both problems together (Chapter 5).

This thesis is based on the following three publications:

1. Caglar Gulcehre, Sarath Chandar, Yoshua Bengio. *Memory Augmented Neural Networks with Wormhole Connections.* In arXiv, 2017.
   — **Personal Contributions:** The model is inspired from our earlier work (Gulcehre et al., 2016). Caglar came up with the idea of tying the reading and writing head. Caglar performed the language model and SNLI experiments. I created the sequential stroke multi-digit MNIST task and performed the benchmark experiments. Caglar and I wrote the paper with significant contributions from Yoshua.
2. Sarath Chandar*, Chinnadhurai Sankar*, Eugene Vorontsov, Samira Ebrahimi Kahou, Yoshua Bengio. *Towards Non-saturating Recurrent Units for Modelling Long-term Dependencies.* Proceedings of AAAI, 2019.
   — **Personal Contributions:** I was looking for alternative solutions for vanishing gradients after the TARDIS project. Yoshua suggested the idea of using a flat memory vector with ReLU based gating in one of our discussions. I developed NRUs based on this suggestion. I designed the experiments and implemented the tasks. Chinnadhurai and I ran most of the experiments. Eugene helped us by running the unitary RNN baselines. I wrote the paper with feedback from all the authors.
3. Shagun Sodhani*, Sarath Chandar*, Yoshua Bengio. *Towards Training Recurrent Neural Networks for Lifelong Learning.* Neural Computation, 2019.
   — **Personal Contributions:** The idea of studying the capacity saturation issue in lifelong learning was mine. I also came up with the idea of combining Net2Net with methods that mitigate catastrophic forgetting. I designed the tasks and Shagun ran all the experiments and we both wrote the paper together with feedback from Yoshua.

1.2 Thesis layout

The rest of the thesis is organized as follows. Chapter 2 introduces sequential problems and recurrent neural networks. Chapter 2 also introduces the problem of vanishing and exploding gradients and provides necessary background on state-of-the-art recurrent architectures which aim to solve this problem. Chapter 3 and chapter 4 introduce two different solutions to solve the vanishing gradient problem: LSTMs with wormhole connections and Non-saturating Recurrent Units respectively. Chapter 5 discusses the challenges in training RNNs for lifelong learning

and proposes a solution. Chapter 6 concludes the thesis and outlines future research directions.

2 Background

In this chapter, we will provide the necessary background on recurrent neural networks to understand this thesis. We assume that the reader already knows the basics of machine learning, neural networks, gradient descent, and backpropagation. See (Goodfellow et al., 2016) for a relevant textbook.

2.1 Sequential Problems

This thesis focuses on sequential problems and we give the general definition of a sequential problem here.

In a sequential problem, at every time step t, the system receives some input $\mathbf{x}_t$ and has to produce some output $\mathbf{y}_t$. The number of time steps is a variable and it can potentially be infinite (in which case the system receives an infinite stream of inputs and produces an infinite stream of outputs). The input $\mathbf{x}_t$ at any time step t can be optional and so is the output $\mathbf{y}_t$ at any time step t. $\mathbf{y}_t$ at any time step might be dependent on any of the previous $\mathbf{x}_t$ or even all of previous $\mathbf{x}_t$ which makes the problem more challenging than the standard single step prediction problems where the output depends on just the immediately preceding input.

Now we will show several example applications which can be considered in this framework of sequential problems.

2.1.1 Sequence Classification

Given a sequence $\mathbf{x}_1, ..., \mathbf{x}_T$, the task is to predict $\mathbf{y}_T$ which is the class label of the sequence. This can be considered in the general sequential problem framework with no output $\mathbf{y}_t$ at all time steps except for $t = T$. The sequence is defined as $(\mathbf{x}_1, \_), (\mathbf{x}_2, \_), ..., (\mathbf{x}_{T-1}, \_), (\mathbf{x}_T, \mathbf{y}_T)$.

Example applications include sentence sentiment classification where given a sequence of words, the task is to predict whether the sentiment of the sentence is positive, negative or neutral.

2.1.2 Language Modeling

Given a sequence of words $\mathbf{w}_1, ..., \mathbf{w}_T$, the goal of language modeling is to model the probability of seeing this sequence of words or tokens, $P(\mathbf{w}_1, ..., \mathbf{w}_T)$. This can be represented as the conditional probability of the next word given all the previous words:

$$P(\mathbf{w}_1, ..., \mathbf{w}_T) = \prod_{t=1}^{T} P(\mathbf{w}_t | \mathbf{w}_1^{t-1}) \tag{2.1}$$

where $\mathbf{w}_1^{t-1}$ refers to $\mathbf{w}_1, ..., \mathbf{w}_{t-1}$ and $\mathbf{w}_1^0$ is defined as null. This task can be posed as a sequential prediction problem where at every time t, given $\mathbf{w}_1, ..., \mathbf{w}_{t-1}$, the system predicts $\mathbf{w}_t$. The sequence flow for this problem is defined as $(\_, \mathbf{w}_1)$, $(\mathbf{w}_1, \mathbf{w}_2)$, ..., $(\mathbf{w}_{T-1}, \mathbf{w}_T)$. At any point in time t, the system has seen the previous $t-1$ words (over $t-1$ time steps) and hence should be able to model the probability of the t-th word given the previous $t-1$ words. A system trained with a language modeling objective can be used for unconditional language generation by sampling a word given the sequence of previously sampled words. It may also be used to score a word sequence by $P(\mathbf{w}_1, ..., \mathbf{w}_T)$ using equation 2.1.

2.1.3 Conditional Language Modeling

In conditional language modeling, we model the probability of a sequence of words $\mathbf{w}_1, ..., \mathbf{w}_T$ given some object $\mathbf{o}$: $P(\mathbf{w}_1, ..., \mathbf{w}_T | \mathbf{o})$. The conditioning object $\mathbf{o}$ itself can be a sequence. There are several applications for conditional language modeling.

In question answering, given a question (which is a sequence of words), the system has to generate the answer (which is another sequence of words). In machine translation, given a sequence of words in one language, the system has to generate another sequence of words in a different language. In image captioning, given an image (which is a single object), the system has to generate a suitable caption (which is a sequence of words). In dialogue systems, given the previous utterances (each of which is a sequence of sentences), the system has to generate the next utterance (which is a sequence of words). The sequence flow for conditional language modeling is defined as $(\mathbf{o}_1, \_), ..., (\mathbf{o}_{T_1}, \_), (\_, \mathbf{w}_1), ..., (\_, \mathbf{w}_{T_2})$ where T_1 is 1 for non-sequential objects.

2.1.4 Sequential Decision Making

In sequential decision making, at any time step t, the agent perceives a state $\mathbf{s}_t$ and takes an action $\mathbf{a}_t$. Based on the current state and the action, the agent receives some reward $\mathbf{r}_{t+1}$ and moves to a new state $\mathbf{s}_{t+1}$ with probability $P(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$. The goal of the agent is to find a policy (i.e. which action to take in the given state) such that it can maximize the discounted sum of the rewards, where the future rewards are discounted by some discount factor $\gamma \in [0, 1]$. The sequence flow in this setting is $(\{\_, \mathbf{s}_1\}, \mathbf{a}_1), (\{\mathbf{r}_2, \mathbf{s}_2\}, \mathbf{a}_2), ..., (\{\mathbf{r}_T, \mathbf{s}_T\}, \mathbf{a}_T)$. This is the central problem of Reinforcement Learning.

2.2 Vanilla Recurrent Neural Networks

In this section, we will first show the limitations in using feed-forward networks for sequential problems. Then we will introduce recurrent neural networks as an elegant solution for sequential problems. We assume discrete outputs throughout the discussion (i.e. $\mathbf{y}_t$ takes a finite set of values). The discussion can be easily extended to the continuous output setting.

2.2.1 Limitations of Feedforward Neural Networks

Feedforward neural networks are the simplest neural network architectures that take some input $\mathbf{x}$ and predict some output $\mathbf{y}$. Consider a sequential problem where at every time step t, the network receives $\mathbf{x}_t$ and has to predict $\mathbf{y}_t$. The typical architecture of a feedforward one-hidden layer fully-connected neural network is defined as follows:

$$\mathbf{h}_t = f(\boldsymbol{U}\mathbf{x}_t + \mathbf{b}) \tag{2.2}$$

$$\mathbf{o}_t = softmax(\mathbf{V}\mathbf{h}_t + \mathbf{c}) \tag{2.3}$$

where f is some non-linearity function like sigmoid or tanh, and $\mathbf{h}_t$ is the hidden state of the network. $\boldsymbol{U}$ is the input to hidden state connection matrix and $\mathbf{V}$ is the hidden state to output connection matrix. $\mathbf{b}$ and $\mathbf{c}$ are the bias vectors for the hidden layer and the output layer respectively. Softmax gives a valid probability distribution over possible values $\mathbf{y}_t$ can take.

Given a sequence of $\mathbf{x}_t, \mathbf{y}_t$, $\{(\mathbf{x}_t, \mathbf{y}_t)\}_{t=1}^{T}$, the network is trained by minimizing

the negative log-likelihood of $\mathbf{y}_t$ given $\mathbf{x}_t$.

$$\mathcal{L} = -\sum_{t=1}^{T} \log\ P_{model}(\mathbf{y}_t|\mathbf{x}_t) \tag{2.4}$$

This loss function can be minimized by using the backpropagation algorithm. This simple model learns to predict $\mathbf{y}_t$ solely based on $\mathbf{x}_t$. Hence, it cannot capture the dependency of $\mathbf{y}_t$ with some previous $\mathbf{x}_{t'}$ with $t' < t$. In other words, this is a stateless architecture since it cannot remember previous $\mathbf{x}_{t'}$ while making prediction for the current $\mathbf{y}_t$.

The straightforward extension of this architecture to model the dependencies with previous inputs is to consider previous k inputs while predicting the next output. The architecture of such a network is defined as follows:

$$\mathbf{h}_t = f(\boldsymbol{U}_1\mathbf{x}_t + \boldsymbol{U}_2\mathbf{x}_{t-1} + ... + \boldsymbol{U}_k\mathbf{x}_{t-k+1} + \mathbf{b}) \tag{2.5}$$

$$\mathbf{o}_t = softmax(\mathbf{V}\mathbf{h}_t + \mathbf{c}) \tag{2.6}$$

The loss function in this case is

$$\mathcal{L} = -\sum_{t=1}^{T} \log\ P_{model}(\mathbf{y}_t|\mathbf{x}_t, \mathbf{x}_{t-1}, ..., \mathbf{x}_{t-k+1}) \tag{2.7}$$

While we simulate states in this architecture by explicitly feeding the previous k inputs, this model has limited expressive power in the sense that it can only model the dependencies within the previous k inputs. In addition, the number of parameters of the model grows linearly as the value of k increases. Ideally we would expect to model the dependencies with all the previous inputs and also avoid this linear growth in the number of parameters as the length of the sequence increases.

2.2.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a family of neural network architectures specialized for processing sequential data. RNNs are similar to feedforward networks except that the hidden states have a recurrent connection. RNNs can use this hidden state to remember the previous inputs and hence the hidden state at any point of time acts like a summary of the entire history. The typical architecture of a simple, or *vanilla* RNN is defined as follows:

$$\mathbf{h}_t = f(\boldsymbol{U}\mathbf{x}_t + \boldsymbol{W}\mathbf{h}_{t-1} + \mathbf{b}) \tag{2.8}$$

$$\mathbf{o}_t = softmax(\mathbf{V}\mathbf{h}_t + \mathbf{c}) \tag{2.9}$$

where the matrix $\boldsymbol{W}$ corresponds to hidden to hidden connections. The initial hidden state $\mathbf{h}_0$ can be initialized to a zero vector or can be learned like other parameters. Figure 2.1 shows the architecture of this RNN. RNNs when unrolled across time correspond to a feedforward neural network with shared weights in every time step (see Figure 2.2). Hence one can learn the parameters of the RNN by using backpropagation in this unrolled network. This is known as backpropagation through time (BPTT). The memory required for BPTT increases linearly with the length of the sequence. Hence we often approximate BPTT with truncated BPTT in which we truncate the backprop after some k time steps.

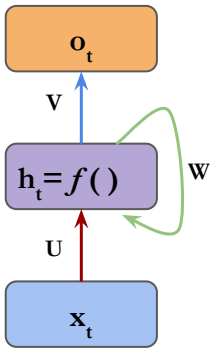


Figure 2.1 – A *vanilla* Recurrent Neural Network. Biases are not shown in the illustration.

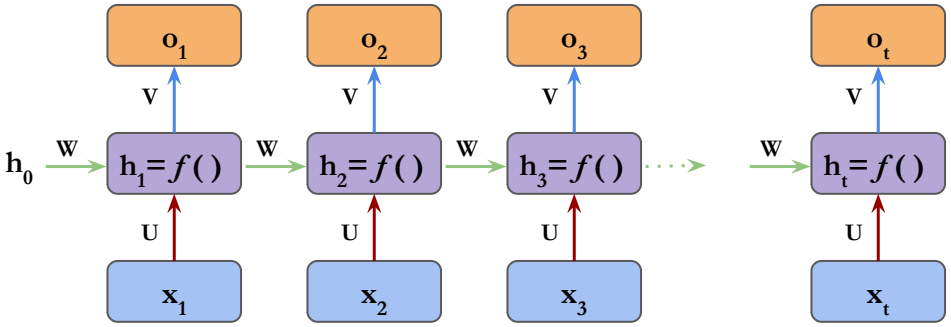


Figure 2.2 – RNN unrolled across the time steps. An unrolled RNN can be considered as a feed-forward neural network with weights $\boldsymbol{W}, \boldsymbol{U}, \mathbf{V}$ shared across every layer.

The loss function in this case could be

$$\mathcal{L} = -\sum_{t=1}^{T} \log\, P_{model}(\mathbf{y}_t | \mathbf{x}_1^t) \tag{2.10}$$

RNNs have several nice properties suitable for sequential problems.

1. Prediction for any $\mathbf{y}_t$ is dependent on all the previous inputs.

2. The number of parameters is independent of the length of the sequences. This is achieved by sharing the same set of parameters across the time steps.
3. RNNs can naturally handle variable length sequences.

2.3 Problem of vanishing and exploding gradients

Even though RNNs are capable of capturing long term dependencies, they often learn only short term dependencies. This is mainly due to the fact that gradients vanish as the length of the sequence increases. This is known as vanishing gradient problem and has been well studied in the literature (Hochreiter, 1991; Bengio et al., 1994).

Consider an RNN which at each timestep t takes an input $\mathbf{x}_t \in \mathbb{R}^d$ and produces an output $\mathbf{o}_t \in \mathbb{R}^o$. The hidden state of the RNN can be written as,

$$\mathbf{z}_t = \boldsymbol{W}\mathbf{h}_{t-1} + \boldsymbol{U}\mathbf{x}_t, \tag{2.11}$$

$$\mathbf{h}_t = f(\mathbf{z}_t). \tag{2.12}$$

where $\boldsymbol{W}$ and $\boldsymbol{U}$ are the recurrent and the input weights of the RNN respectively and $f(\cdot)$ is a non-linear activation function. Let $\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}_t$ be the loss function that the RNN is trying to minimize. Given an input sequence of length T, we can write the derivative of the loss $\mathcal{L}$ with respect to parameters $\boldsymbol{\theta}$ as,

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \sum_{1 \le t_1 \le T} \frac{\partial \mathcal{L}_{t_1}}{\partial \boldsymbol{\theta}} = \sum_{1 \le t_1 \le T} \sum_{1 \le t_0 \le t_1} \frac{\partial \mathcal{L}_{t_1}}{\partial \mathbf{h}_{t_1}} \frac{\partial \mathbf{h}_{t_1}}{\partial \mathbf{h}_{t_0}} \frac{\partial \mathbf{h}_{t_0}}{\partial \boldsymbol{\theta}}. \tag{2.13}$$

The multiplication of many Jacobians in the form of $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$ to obtain $\frac{\partial \mathbf{h}_{t_1}}{\partial \mathbf{h}_{t_0}}$ is the main reason of the vanishing and the exploding gradients (Pascanu et al., 2013b):

$$\frac{\partial \mathbf{h}_{t_1}}{\partial \mathbf{h}_{t_0}} = \prod_{t_0 < t \le t_1} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} = \prod_{t_0 < t \le t_1} \text{diag}[f'(\mathbf{z}_t)]\boldsymbol{W}. \tag{2.14}$$

Let us assume that the singular values of a matrix $\mathbf{M}$ are ordered as, $\sigma_1(\mathbf{M}) \ge \sigma_2(\mathbf{M}) \ge \cdots \ge \sigma_n(\mathbf{M})$. Let α be an upper bound on the singular values of $\boldsymbol{W}$, s.t. $\alpha \ge \sigma_1(\boldsymbol{W})$, then the norm of the Jacobian will satisfy (Zilly et al., 2016),

$$||\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}|| \le ||\boldsymbol{W}|| \;\; ||\text{diag}[f'(\mathbf{z}_t)|| \le \alpha \; \sigma_1(\text{diag}[f'(\mathbf{z}_t)]), \tag{2.15}$$

Pascanu et al. (2013b) showed that for $||\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}|| \leq \sigma_1(\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}) \leq \eta$, the following inequality holds:

$$|| \prod_{t_0 \leq t \leq t_1} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}|| \leq \sigma_1 \left(\prod_{t_0 \leq t \leq t_1} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \right) \leq \eta^{t_1 - t_0}. \tag{2.16}$$

where $\alpha||f'(\mathbf{z}_t)|| \leq \eta$ for all t. We see that the norm of the product of Jacobians grows exponentially on $t_1 - t_0$. Hence, if $\eta < 1$, the norm of the gradients will vanish exponentially fast. Similarly, if $\eta > 1$, the norm of the gradients may explode exponentially fast. This is the problem of vanishing and exploding gradients in RNNs. While gradient clipping can limit the effect of exploding gradients, vanishing gradients are harder to prevent and so limit the network's ability to learn long term dependencies.

The second cause of vanishing gradients is the activation function f. If the activation function is a saturating function like a sigmoid or tanh, then its Jacobian $\text{diag}[f'(\mathbf{z}_t)]$ has eigenvalues less than or equal to one, causing the gradient signal to decay during backpropagation. Long term information may decay when the spectral norm of $(\boldsymbol{W}\text{diag}[f'(\mathbf{z}_t)])$ is less than 1 but this condition is actually necessary in order to store this information reliably (Bengio et al., 1994; Pascanu et al., 2013b).

Designing a recurrent neural network architecture which has no vanishing or exploding gradients is an open question in the RNN literature. While there are certain recurrent architectures which have no vanishing or exploding gradients, their parameterizations are rather less expressive (as we will see in this chapter).

2.4 Long Short-Term Memory (LSTM) Networks

Long Short-Term Memory (LSTM) Networks were introduced by Hochreiter and Schmidhuber (1997) to mitigate the issue of vanishing and exploding gradients. The LSTM is a recurrent neural network which maintains two recurrent states: a hidden state $\mathbf{h}_t$ (similar to a *vanilla* RNN) and a cell state $\mathbf{c}_t$. The cell state is updated in an additive manner instead of the usual multiplicative manner, which is the main source of vanishing gradients. Further, the information flow to the cell state and out of the cell state are controlled by sigmoidal gates.

Specifically, the LSTM has three different gates: forget gate $\mathbf{f}_t$, input gate $\mathbf{i}_t$, and output gate $\mathbf{o}_t$. All the three gates are functions of the current input $\mathbf{x}_t$ and

the previous hidden state $\mathbf{h}_{t-1}$.

$$\mathbf{i}_t = \text{sigmoid}(\boldsymbol{W}_{xi}\mathbf{x}_t + \boldsymbol{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i) \tag{2.17}$$
$$\mathbf{f}_t = \text{sigmoid}(\boldsymbol{W}_{xf}\mathbf{x}_t + \boldsymbol{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f) \tag{2.18}$$
$$\mathbf{o}_t = \text{sigmoid}(\boldsymbol{W}_{xo}\mathbf{x}_t + \boldsymbol{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o). \tag{2.19}$$

The cell state $\mathbf{c}_t$ is updated as follows:

$$\tilde{\mathbf{c}}_t = \tanh(\boldsymbol{W}_{xc}\mathbf{x}_t + \boldsymbol{W}_{hc}\mathbf{h}_{t-1} + \mathbf{b}_c) \tag{2.20}$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \tag{2.21}$$

where $\odot$ denotes element-wise product and $\tilde{\mathbf{c}}_t$ is the new information to be added to the cell state. Intuitively, $\mathbf{f}_t$ controls how much of $\mathbf{c}_{t-1}$ should be forgotten and $\mathbf{i}_t$ controls how much of $\tilde{\mathbf{c}}_t$ has to be added to the cell state.

The hidden state is computed using the current cell state as follows:

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t). \tag{2.22}$$

The LSTM maintains a cell state that is updated by addition rather than multiplication (see Equation 2.21) and serves as an "error carousel" that allows gradients to skip over many transition operations. The gates determine which information is skipped. Figure 2.3 shows the architecture of LSTM.

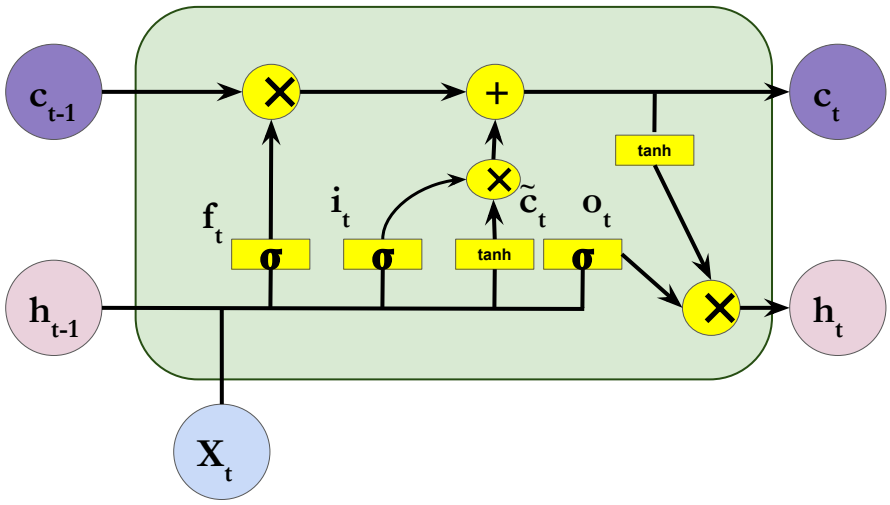


Figure 2.3 – Long Short-Term Memory (LSTM) Network.

The cell state and the hidden state in an LSTM play two different roles. While the cell state is responsible for information from the past that will be useful for future predictions, the hidden state is responsible for information from the past that

will be useful for the current prediction. Note that in a vanilla RNN, the hidden state played both these roles together. While a vanilla RNN has all the expressive power that an LSTM has, the explicit parameterization of the LSTM makes the learning problem easier. This idea of constructing explicit parameterizations to make learning easier is one of the central themes of this thesis.

The LSTM, even though introduced in 1997, is still the de-facto model that is used for most of the sequential problems. There has been few improvements to the original version of LSTM proposed by Hochreiter and Schmidhuber (1997). The initial version of the LSTM did not have forget gates. The idea of adding forgetting ability to free memory from irrelevant information was introduced in Gers et al. (2000). Gers et al. (2002) introduced the idea of peepholes connecting the gates to the cell state so the network can learn precise timing and counting of the internal states. The parameterization of an LSTM with peephole connections is given below:

$$\mathbf{i}_t = \text{sigmoid}(\boldsymbol{W}_{xi}\mathbf{x}_t + \boldsymbol{W}_{hi}\mathbf{h}_{t-1} + \boldsymbol{W}_{ci}\mathbf{c}_{t-1} + \mathbf{b}_i) \tag{2.23}$$

$$\mathbf{f}_t = \text{sigmoid}(\boldsymbol{W}_{xf}\mathbf{x}_t + \boldsymbol{W}_{hf}\mathbf{h}_{t-1} + \boldsymbol{W}_{cf}\mathbf{c}_{t-1} + \mathbf{b}_f) \tag{2.24}$$

$$\mathbf{c}_t = \mathbf{f}_t\mathbf{c}_{t-1} + \mathbf{i}_t \tanh(\boldsymbol{W}_{xc}\mathbf{x}_t + \boldsymbol{W}_{hc}\mathbf{h}_{t-1} + \mathbf{b}_c) \tag{2.25}$$

$$\mathbf{o}_t = \text{sigmoid}(\boldsymbol{W}_{xo}\mathbf{x}_t + \boldsymbol{W}_{ho}\mathbf{h}_{t-1} + \boldsymbol{W}_{co}\mathbf{c}_t + \mathbf{b}_o) \tag{2.26}$$

$$\mathbf{h}_t = \mathbf{o}_t \tanh(\mathbf{c}_t) \tag{2.27}$$

where the weight matrices from the cell to gate vectors (e.g. $\boldsymbol{W}_{cf}$) are diagonal. When we refer to LSTMs in this thesis, it is usually LSTMs with peephole connections.

While the additive cell updates avoided the first source of vanishing gradients, the gradients still vanish in LSTMs due to the saturating activation functions used for gating. When the gating unit activations saturate, the gradient on the gating unit themselves vanishes. However, locking the gates to ON (or OFF) is necessary for distance gradient propagation across memory. This introduces an unfortunate trade-off where either the gate mechanism receives updates or gradients are skipped across many transition operators. Due to this issue, LSTMs often end up learning only short term dependencies. However, the short term dependencies learned by LSTMs are still longer than the dependencies learned by a vanilla RNN and hence the name *long short-term* memory.

2.4.1 Forget gate initialization

Gers et al. (2000) proposed to initialize the bias of the forget gate to 1 which helped in modelling medium term dependencies when compared with zero initialization. This was also recently highlighted in the extensive exploration study by

Józefowicz et al. (2015). Tallec and Ollivier (2018) proposed chrono-initialization to initialize the bias of the forget gate ($\mathbf{b}_f$) and input gate ($\mathbf{b}_i$). Chrono-initialization requires the knowledge of maximum dependency length ($T_{\max}$) and it initializes the gates as follows:

$$\mathbf{b}_f \sim \log(\mathcal{U}[1, T_{\max} - 1]) \tag{2.28}$$
$$\mathbf{b}_i = -\mathbf{b}_f \tag{2.29}$$

This initialization method encourages the network to remember information for approximately $T_{\max}$ time steps. While these forget gate bias initialization techniques encourage the model to retain information longer, the model is free to unlearn this behaviour.

2.5 Other gated architectures

A simpler gated architecture called Gated Recurrent Unit (GRU) was introduced by Cho et al. (2014). Similar to LSTMs, GRUs also have gating units to control the information flow. However, unlike LSTMs which has three types of gates, GRUs has only two type of gates: an update gate $\mathbf{z}_t$ and a reset gate $\mathbf{r}_t$. Both the gates are functions of the current input $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$.

$$\mathbf{z}_t = \text{sigmoid}(\boldsymbol{W}_{xz}\mathbf{x}_t + \boldsymbol{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_z) \tag{2.30}$$
$$\mathbf{r}_t = \text{sigmoid}(\boldsymbol{W}_{xr}\mathbf{x}_t + \boldsymbol{W}_{hr}\mathbf{h}_{t-1} + \mathbf{b}_r) \tag{2.31}$$

At any time step t, first the candidate hidden state $\tilde{\mathbf{h}}_t$ is computed as follows:

$$\tilde{\mathbf{h}}_t = \tanh(\boldsymbol{W}_{xh}\mathbf{x}_t + \boldsymbol{W}_{hh}(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h) \tag{2.32}$$

When the reset gate is close to zero, the unit essentially forgets the previous state. This is similar to the forget gate in LSTM.

Now the hidden state $\mathbf{h}_t$ is computed as a linear interpolation between the previous hidden state $\mathbf{h}_{t-1}$ and the current candidate hidden state $\tilde{\mathbf{h}}_t$:

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t \tag{2.33}$$

where the update gate $\mathbf{z}_t$ determines how much the unit updates its content based on the newly available content. This is very similar to the cell state update in an LSTM. However, GRU does not maintain a separate cell state. The hidden state serves the purpose of both the cell state and the hidden state in an LSTM. While the cell state helps the LSTM to remember long-term information, in GRU, it is the

duty of the update gate to learn to remember long-term information. GRUs while simpler than LSTMs, are comparable in performance to LSTMs (Chung et al., 2014) and hence are widely used.

While GRUs reduced the number of gates in LSTM from three to two, Just Another NETwork (JANET) (van der Westhuizen and Lasenby, 2018) reduces the number of gates further to only one: the forget gate $\mathbf{f}_t$.

$$\mathbf{f}_t = \text{sigmoid}(\boldsymbol{W}_{xf}\mathbf{x}_t + \boldsymbol{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f) \tag{2.34}$$

At any time step t, first the candidate hidden state $\tilde{\mathbf{h}}_t$ is computed as follows:

$$\tilde{\mathbf{h}}_t = \tanh(\boldsymbol{W}_{xh}\mathbf{x}_t + \boldsymbol{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h) \tag{2.35}$$

Then the hidden state $\mathbf{h}_t$ is computed as follows:

$$\mathbf{h}_t = \mathbf{f}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{f}_t) \odot \tilde{\mathbf{h}}_t \tag{2.36}$$

JANET couples the input gate and the forget gate while removing the output gate. Like GRUs, JANET also does not contain an explicit cell state. van der Westhuizen and Lasenby (2018) hypothesized that it will be beneficial to allow slightly more information to accumulate than the amount of information forgotten. This can be implemented by subtracting a pre-specified value β from the input control component. The modified JANET equations are given as follows:

$$\mathbf{s}_t = \boldsymbol{W}_{xf}\mathbf{x}_t + \boldsymbol{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f \tag{2.37}$$

$$\tilde{\mathbf{h}}_t = \tanh(\boldsymbol{W}_{xh}\mathbf{x}_t + \boldsymbol{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h) \tag{2.38}$$

$$\mathbf{h}_t = \text{sigmoid}(\mathbf{s}_t) \odot \mathbf{h}_{t-1} + (1 - \text{sigmoid}(\mathbf{s}_t - \beta)) \odot \tilde{\mathbf{h}}_t \tag{2.39}$$

van der Westhuizen and Lasenby (2018) showed that JANET with chrono-initialization performs better than LSTMs and LSTMs with chrono-initialization. van der Westhuizen and Lasenby (2018) call this the unreasonable effectiveness of the forget gate.

The fact that GRUs and JANETs, with lesser number of gates, work better than LSTMs support our hypothesis that saturating gates make learning long-term dependencies difficult. We will get back to this crucial observation in Chapter 4.

2.6 Orthogonal RNNs

Better initialization of the recurrent weight matrix has been explored in the literature of RNNs. One standard approach is orthogonal initialization where the recurrent weight matrices are initialized using orthogonal matrices. Orthogonal initialization makes sure that the spectral norm of the recurrent matrix is one during initialization and hence there is no vanishing gradient due to the recurrent weight matrix. This approach has two caveats. Firstly, the gradient would still vanish due to saturating activation functions. Secondly, this is only an initialization method. Hence, RNNs are free to move away from orthogonal weight matrices and the norm of the recurrent matrix can become less than or greater than one which would result in vanishing or exploding gradients respectively.

Le et al. (2015) addressed the first caveat using identity initialization (which results in a recurrent weight matrix with the spectral norm of one) to train RNNs with ReLU activation functions. While the authors showed good performance in a few tasks, ReLU RNNs are notoriously difficult to train due to exploding gradients. A simple way to avoid the second caveat is to add a regularization term to ensure that the recurrent weight matrix $\boldsymbol{W}$ stays close to an orthogonal matrix throughout the training. For example, one can add:

$$\lambda||\boldsymbol{W}^T\boldsymbol{W} - \mathbf{I}||^2. \tag{2.40}$$

Vorontsov et al. (2017) proposed a direct parameterization of $\boldsymbol{W}$ which permits a direct control over the spectral norms of the matrix. Specifically, they consider the singular value decomposition of the $\boldsymbol{W}$ matrix:

$$\boldsymbol{W} = \boldsymbol{U}\mathbf{S}\mathbf{V}^T \tag{2.41}$$

where $\boldsymbol{U}$ and $\mathbf{V}$ are the orthogonal basis matrices and $\mathbf{S}$ is a diagonal spectral matrix that has the singular values of $\boldsymbol{W}$ as the diagonal entries. The basis matrices $\boldsymbol{U}$ and $\mathbf{V}$ are kept orthogonal by doing geodesic gradient descent along the set of weights that satisfy $\boldsymbol{U}\boldsymbol{U}^T = \mathbf{I}$ and $\mathbf{V}\mathbf{V}^T = \mathbf{I}$ respectively. If we fix the $\mathbf{S}$ matrix to be an identity matrix, we can ensure the orthogonality of $\boldsymbol{W}$ matrix throughout the training. However, Vorontsov et al. (2017) proposes to parameterize the $\mathbf{S}$ matrix in such a way that the singular values can deviate from 1 by a margin of m. This is achieved with the following parameterization:

$$s_i = 2m(\sigma(p_i) - 0.5) + 1, \qquad s_i \in \{\text{diag}(\mathbf{S})\}, m \in [0, 1] \tag{2.42}$$

This parameterization guarantees that the singular values are restricted to the range $[1 - m, 1 + m]$ while the underlying parameters p_i are updated by using the regular gradient descent. Experiments by Vorontsov et al. (2017) conclude that

maintaining the hard orthogonal constraint can be overall detrimental. Authors hypothesise that orthogonal RNNs cannot forget information and this might be problematic in tasks that require only short-term dependencies. On the other hand, allowing the singular values to deviate from 1 by a small margin consistently improved the performance. Deviating by a large margin can lead to vanishing and exploding gradients.

2.7 Unitary RNNs

Another line of work starting from (Arjovsky et al., 2016) explored the idea of using unitary matrix parameterization for recurrent weight matrix. Unitary matrices generalize orthogonal matrices to the complex domain. The spectral norm of the unitary matrices is also one. Arjovsky et al. (2016) proposed a simple parameterization of unitary matrices based on the observation that the product of unitary matrices is a unitary matrix. Their parameterization ensured that one can do gradient descent on the parameters without deviating from the unitary recurrent weight matrix.

Wisdom et al. (2016) observed that the parameterization of Arjovsky et al. (2016) does not span the entire space of unitary matrices and proposed an alternate parameterization that has full capacity. To stay in the manifold of unitary matrices, Wisdom et al. (2016) has to re-project the recurrent weight matrix to the unitary space after every gradient update. Jing et al. (2017b) avoids the re-projection cost by using an efficient unitary RNN parameterization with tunable representation capacity which does not require any re-projection step. This model is called the Efficient Unitary Neural Network (EUNN).

One issue with orthogonal and unitary matrices is that they do not filter out information, preserving gradient norm but making forgetting impossible. Gated Orthogonal Recurrent Units (GORUs) (Jing et al., 2017a) addressed this issue by combining Unitary RNNs with a forget gate to learn to filter the information. It is worth noting that Unitary RNNs are, in general, slow to train due to the inherent sequential computations in the parameterizations. Restricting the recurrent weight matrices to be orthogonal or unitary also restricts the representation capacity of an RNN.

2.8 Statistical Recurrent Units

Oliva et al. (2017) proposed a new ungated recurrent architecture called Statisti-

cal Recurrent Units (SRUs). SRUs model sequential information by using recurrent statistics which are generated at multiple time scales. Specifically, SRUs maintain exponential moving averages $\mathbf{m}^{(\alpha)}$ defined as follows:

$$\mathbf{m}_t^{(\alpha)} = \alpha \mathbf{m}_{t-1}^{(\alpha)} + (1-\alpha)\hat{\mathbf{m}}_t \tag{2.43}$$

where $\alpha \in [0, 1)$ defines the scale of the moving average and $\hat{\mathbf{m}}_t$ is the recurrent statistics at time t computed as follows:

$$\mathbf{r}_t = f(\boldsymbol{W}_r \mathbf{m}_{t-1} + \mathbf{b}_r) \tag{2.44}$$

$$\hat{\mathbf{m}}_t = f(\boldsymbol{W}_m \mathbf{r}_t + \boldsymbol{W}_x \mathbf{x}_t + \mathbf{b}_m) \tag{2.45}$$

where f is a ReLU activation function. As we can see, the recurrent statistics at time t is conditioned on the current input and on the recurrent statistics at the previous time step. Oliva et al. (2017) proposed to consider m different values for α (where m is a hyper-parameter) to maintain the recurrent statistics at multiple time scales. At any time-step t, let $\mathbf{m}_t$ denote the concatenation of all such recurrent statistics.

$$\mathbf{m}_t = [\mathbf{m}_t^{\alpha_1}; \mathbf{m}_t^{\alpha_2}; ...; \mathbf{m}_t^{\alpha_m}] \tag{2.46}$$

Now the hidden state of the SRU is computed as follows:

$$\mathbf{h}_t = f(\boldsymbol{W}_h \mathbf{m}_t + \mathbf{b}_h) \tag{2.47}$$

SRUs handle the vanishing gradients by having an ungated architecture with ReLU activation function and simple recurrent statistics. However, it is worth noting that the recurrent statistics are computed by using exponential moving averages which will shrink the gradients.

2.9 Memory Augmented Neural Networks

Memory augmented neural networks (MANNs) (Graves et al., 2014; Weston et al., 2015) are neural network architectures that have access to an external memory (usually a matrix) which it can read from or write to. The external memory matrix can be considered as a generalization of the cell state vector in an LSTM. In this section, we will introduce two memory-augmented architectures: Neural Turing Machines (Graves et al., 2014) and Dynamic Neural Turing Machines (Gulcehre et al., 2016).

2.9.1 Neural Turing Machines

Memory

Memory in Neural Turing Machine (NTM) is a matrix $\mathbf{M}_t \in \mathbb{R}^{k \times q}$ where k is the number of memory cells and q is the dimensionality of each memory cell. The controller neural network can use this memory matrix as a scratch pad to write to and read from.

Model Operation

The controller in an NTM can either be a feed-forward network or an RNN. The controller has multiple read heads, write heads, and erase heads to access the memory. For this discussion, we will assume that the NTM has only one head per operation. However, this description can be easily extended to the multi-head setting.

At each time step t, the controller receives an input $\mathbf{x}_t$. Then it generates the read weight $\mathbf{w}_t^r \in \mathbb{R}^{k \times 1}$. The read weights are used to generate the content vector $\mathbf{r}_t$ as a weighted combination of all the cells in the memory:

$$\mathbf{r}_t = \mathbf{M}_t^T \mathbf{w}_t^r \in \mathbb{R}^{q \times 1} \tag{2.48}$$

This content vector is used to condition the hidden state of the controller. If the controller is a feed-forward network, then the hidden state of the controller is defined as follows:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{r}_t) \tag{2.49}$$

where f is any non-linear function. If the controller is an RNN, then the hidden state of the controller is computed as:

$$\mathbf{h}_t = g(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{r}_t) \tag{2.50}$$

where g can be a vanilla RNN or a GRU network or an LSTM network.

The controller also updates the memory with a projection of the current hidden state. Specifically, it computes the content to write as follows:

$$\mathbf{a}_t = f(\mathbf{h}_t, \mathbf{x}_t) \tag{2.51}$$

It also generate write weights $\mathbf{w}_t^w \in \mathbb{R}^{k \times 1}$ in similar way as read weights and erase vector $\mathbf{e}_t \in \mathbb{R}^{q \times 1}$ whose elements are in the range (0,1). It updates each cell $\mathbf{M}_t[i]$

as follows:

$$\mathbf{M}_t[i] = \mathbf{M}_{t-1}[i](\mathbf{1} - \mathbf{w}_t^w[i]\mathbf{e}_t) + \mathbf{w}_t^w[i]\mathbf{a}_t \tag{2.52}$$

Intuitively, the writer erases some content from every cell in the memory and adds a weighted version of the new content to every cell in the memory.

Thus in every time step, the controller computes the current hidden state as a function of the content vector generated from the memory. Also, at every time step, the controller writes the current hidden state to the memory. Thus, even if the controller is a feed-forward network, the architecture is recurrent due to the conditioning on the memory.

Addressing Mechanism

Now we will describe how the read head generates the read weights to address the memory cells. The write heads generate their write weights in a similar manner.

In NTMs, memory addressing is a combination of content-based addressing and location-based addressing.

The read head first generates a key for memory access:

$$\mathbf{k}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}) \in \mathbb{R}^{q \times 1} \tag{2.53}$$

The key is compared with each cell in the memory $\mathbf{M}_t$ by using some similarity measure S to generate the content-based weights as follows:

$$\mathbf{w}_t^c = \frac{\exp(\beta_t S[\mathbf{k}_t, \mathbf{M}_t[i]])}{\sum_j \exp(\beta_t S[\mathbf{k}_t, \mathbf{M}_t[j]])} \tag{2.54}$$

where β_t is a positive key strength vector that can amplify or attenuate the precision of the focus.

Next, the read head generates a scalar interpolation gate $g_t \in (0, 1)$ which is used to interpolate between the content-based addressing and the previous address $\mathbf{w}_{t-1}^r$. The interpolated weight $\mathbf{w}_t^g$ is defined as follows:

$$\mathbf{w}_t^g = g_t \mathbf{w}_t^c + (1 - g_t)\mathbf{w}_{t-1}^r \tag{2.55}$$

If the gating weight is one, then the previous weight is ignored and only the content-based weight is considered. If the gating weight is zero, then the content-based weight is ignored and only the previous weight is considered. This interpolated weight is passed as input to the location-based addressing.

After interpolation, the read head generates a shifting weight $\mathbf{s}_t$ which defines a normalized distribution over the allowed integer shifts. If the memory locations are indexed from 0 to k-1, then the rotation applied to $\mathbf{w}_t^g$ by $\mathbf{s}_t$ can be expressed as a circular convolution:

$$\tilde{\mathbf{w}}_t[i] = \sum_{j=0}^{k-1} \mathbf{w}_t^g[j]\mathbf{s}_t[i-j] \tag{2.56}$$

where the indices are computed modulo N. The convolution operation may cause leakage of weight over time if the shift weight is not sharp. This is addressed by generating a scalar $\gamma_t \geq 1$ which is used to sharpen $\tilde{\mathbf{w}}_t$ to produce the final weights:

$$\mathbf{w}_t^r[i] = \frac{\tilde{\mathbf{w}}_t[i]^{\gamma_t}}{\sum_j \tilde{\mathbf{w}}_t[j]^{\gamma_t}} \tag{2.57}$$

This addressing mechanism can learn to operate in three modes: pure content-based addressing, pure location-based addressing, content-based addressing following by location-based shifting. As we can see, the entire addressing mechanism is continuous and hence the architecture is fully differentiable. Graves et al. (2014) showed that NTMs can easily solve several algorithmic tasks which are very difficult for LSTMs. NTMs inspired a series of memory augmented architectures: Dynamic NTMs (Gulcehre et al., 2016), Sparse Access Memory (Rae et al., 2016), and Differentiable Neural Computer (Graves et al., 2016). We will describe only the Dynamic NTM architecture, which is more relevant to this thesis.

2.9.2 Dynamic Neural Turing Machines

Memory

The memory matrix $\mathbf{M}_t \in \mathbb{R}^{k \times q}$ in the Dynamic NTM (D-NTM) is partitioned into two parts: the address part $\mathbf{A}_t \in \mathbb{R}^{k \times d_a}$ and the content part $\mathbf{C}_t \in \mathbb{R}^{k \times d_c}$ such that $d_a + d_c = q$.

$$\mathbf{M}_t = [\mathbf{A}_t; \mathbf{C}_t] \tag{2.58}$$

The address part is considered to be a model parameter and is learned during training. During inference, the address part is frozen and is not updated. On the other hand, the content part is updated during both training and inference. Also, the content part is reset to zero for every example, $\mathbf{C}_0 = 0$. The learnable address part allows the model to learn more sophisticated location-based strategies when compared to the location-based addressing used in NTMs.

Model Operation

Like NTMs, the controller in D-NTMs can either be a feed-forward network or an RNN. At each time step t, the controller receives an input $\mathbf{x}_t$. Then it generates the read weights $\mathbf{w}_t^r \in \mathbb{R}^{k\times 1}$. The read weights are used to generate the content vector $\mathbf{r}_t$ as a weighted combination of all the cells in the memory:

$$\mathbf{r}_t = \mathbf{M}_t^T \mathbf{w}_t^r \in \mathbb{R}^{q\times 1}. \tag{2.59}$$

This content vector is used to condition the hidden state of the controller. If the controller is a feed-forward network, then the hidden state of the controller is defined as follows:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{r}_t), \tag{2.60}$$

where f is any non-linear function. If the controller is an RNN, then the hidden state of the controller is computed as:

$$\mathbf{h}_t = g(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{r}_t), \tag{2.61}$$

where g can be a vanilla RNN or a GRU network or an LSTM network.

Similar to the NTM, the controller in D-NTM also updates the memory with a projection of the current hidden state. Specifically, it computes the content vector $\mathbf{c}_t \in \mathbb{R}^{d_c\times 1}$ to write as follows:

$$\mathbf{c}_t = \text{ReLU}(\boldsymbol{W}_m \mathbf{h}_t + \alpha_t \boldsymbol{W}_x \mathbf{x}_t), \tag{2.62}$$

where α_t is a scalar gate controlling the input. It is defined as follows:

$$\alpha_t = f(\mathbf{h}_t, \mathbf{x}_t). \tag{2.63}$$

The controller also generates write weights $\mathbf{w}_t^w \in \mathbb{R}^{k\times 1}$ in similar way as read weights and erase vector $\mathbf{e}_t \in \mathbb{R}^{d_c\times 1}$ whose elements are in the range (0,1). It updates the content of each cell $\mathbf{C}_t[i]$ as follows:

$$\mathbf{C}_t[i] = \mathbf{C}_{t-1}[i](\mathbf{1} - \mathbf{w}_t^w[i]\mathbf{e}_t) + \mathbf{w}_t^w[i]\mathbf{c}_t. \tag{2.64}$$

Intuitively, the writer erases some content from every cell in the memory and adds a weighted version of the new content to every cell in the memory. Unlike NTMs, D-NTMs have a designated *No Operation* (NOP) cell in the memory. Reading from or writing to this NOP cell is ignored. This is used to add the flexibility of not accessing the memory at any time step.

Addressing Mechanism

We describe the addressing mechanism for the read head here. But the same description holds for the write heads as well.

The read head first generates a key for the memory access:

$$\mathbf{k}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}) \in \mathbb{R}^{q \times 1}. \tag{2.65}$$

The key is compared with each cell in the memory $\mathbf{M}_t$ by using some similarity measure S to generate the logits for the address weights as follows:

$$\mathbf{z}_t[i] = \beta_t S[\mathbf{k}_t, \mathbf{M}_t[i]], \tag{2.66}$$

where β_t is a positive key strength vector that can amplify or attenuate the precision of the focus. Up to this point, the addressing mechanism looks like the content-based addressing in the NTM. But note that the memory also has a location part and hence this adressing mechanism is a mix of content-based addressing and location-based addressing.

While writing, sometimes it is effective to put more emphasis on the least recently used (LRU) memory locations so that the information is not concentrated in only few memory cells. To implement this effect, D-NTM computes the moving average of the logits as follows:

$$\mathbf{v}_t = 0.1\mathbf{v}_{t-1} + 0.9\mathbf{z}_t. \tag{2.67}$$

This accumulated $\mathbf{v}_t$ is rescaled by a scalar gate $\gamma_t \in (0, 1)$ and subtracted from the current logits before applying the softmax operator. The final weight is given as follows:

$$\mathbf{w}_t^r = \text{softmax}(\mathbf{z}_t - \gamma_t \mathbf{v}_{t-1}). \tag{2.68}$$

By setting γ_t to 0, the model can choose to use pure content-based addressing. This is known as dynamic LRU addressing since the model can dynamically decide whether to use LRU addressing or not.

D-NTM uses single head with multi-step addressing similar to Sukhbaatar et al. (2015) instead of multiple heads used by the NTM.

Discrete Addressing

$\mathbf{w}_t^r$ in Equation 2.68 is a continuous vector and if used as such, the entire architecture is differentiable. The D-NTM with continuous $\mathbf{w}_t^r$ vector is known as

continuous D-NTM.

Note that $\mathbf{w}_t^r$ is a valid probability distribution over k memory cells. Gulcehre et al. (2016) also propose a discrete D-NTM which samples an index from this probability distribution. The discrete address $\tilde{\mathbf{w}}_t^r$ is defined as follows:

$$\tilde{\mathbf{w}}_t^r[k] = I(k = j) \tag{2.69}$$

where $j \sim \mathbf{w}_t^r$ and I is an indicator function. While we sample the index during training, we can use argmax index during inference. This discrete addressing makes the model non-differentiable and hence Gulcehre et al. (2016) used REINFORCE (Williams, 1992) to approximate the gradient. Training discrete D-NTMs was also difficult. Hence the authors suggested a curriculum based training where the model starts with continuous addressing and moves to discrete addressing over the period of training. It was shown that D-NTMs performed better than NTMs in several tasks.

2.10 Normalization methods

Normalization methods are used in deep neural networks to avoid vanishing gradients due to saturating activation functions. The basic idea is to normalize the pre-activations of a layer such that the activation function in that layer does not saturate. Batch Normalization (Ioffe and Szegedy, 2015) normalized the pre-activation based on the batch-level statistics. Consider the pre-activation $\mathbf{x}$ over a mini-batch $\mathcal{B} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m\}$ of m examples. We first compute the mini-batch mean and variance as follows:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}\mathbf{x}_i \tag{2.70}$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}_i - \mu_{\mathcal{B}})^2 \tag{2.71}$$

Then, we compute the normalized pre-activation $\hat{\mathbf{x}}_i$ by using the mini-batch mean and variance.

$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{2.72}$$

where ϵ is a small non-zero value added for numerical stability. The input to the activation function $\mathbf{y}_i$ is computed as follows:

$$\mathbf{y}_i = \gamma \hat{\mathbf{x}}_i + \beta \tag{2.73}$$

where γ and β are the scaling and the shifting parameters respectively. These parameters are also learnt as part of training. Finally, the hidden layer activations are computed as

$$\mathbf{h} = f(BN(\mathbf{x})) \tag{2.74}$$

where f is the activation function, $\mathbf{x}$ is the pre-activation, and BN is the batch-normalization operation such that $BN(\mathbf{x}) = \mathbf{y}$. Note that the scale and the shift allows the network to cancel the effect of normalization if the task demands it. During testing, we will use a global mean and variance computed from training set.

Batch Normalization has been extended for recurrent neural networks in (Cooijmans et al., 2016; Laurent et al., 2016). However, to obtain the best performance using recurrent batch-normalization, one has to track independent normalization statistics for each time-step. During inference time, it is not clear how to handle sequences longer than the training sequences.

Ba et al. (2016) introduced Layer normalization as an alternative to the recurrent batch normalization. The basic proposal by Ba et al. (2016) is to compute the normalization statistics using the hidden units in the same layer instead of using mini-batch of examples. Specifically, we compute

$$\mu_l = \frac{1}{H} \sum_{i=1}^{H} x_i \tag{2.75}$$

$$\sigma_l^2 = \frac{1}{H} \sum_{i=1}^{H} (x_i - \mu_l)^2 \tag{2.76}$$

where H is the number of hidden units in the layer and x_i is the pre-activation for i-th hidden unit. Now, we compute the normalized pre-activations as follows:

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu_l}{\sqrt{\sigma_l^2 + \epsilon}} \tag{2.77}$$

Similar to the batch normalization, the normalized pre-activation is both scaled and shifted before applying the activation function.

The main advantage of layer normalization over batch normalization is we do not need to compute batch-level statistics. Hence, it is very easy to extend layer normalization to RNNs. We can apply layer normalization for pre-activations of

every activation function used in RNNs, LSTMs, or any other recurrent architectures. Ba et al. (2016) report that layer normalization helps in faster convergence and also achieves better performance for several recurrent architectures.

3 LSTMs with Wormhole Connections

In this chapter, we propose our first solution to avoid vanishing gradients called TARDIS (Gülçehre et al., 2017).

3.1 Introduction

Memory augmented neural networks (MANNs) such as neural Turing machine (NTM) (Graves et al., 2014; Rae et al., 2016), dynamic NTM (D-NTM) (Gulcehre et al., 2016), and Differentiable Neural Computer (DNC) (Graves et al., 2016) use an external memory (usually a matrix) to store information and the MANN's controller can learn to both read from and write into the external memory. As we show here, it is in general possible to use particular MANNs to explicitly store the previous hidden states of an RNN in the memory and that will provide shortcut connections through time, called here ***wormhole connections***, to look into the history of the states of the RNN controller. Learning to read and write into an external memory by using neural networks gives the model more freedom or flexibility to retrieve information from its past, forget or store new information into the memory. However, if the addressing mechanism for read and/or write operations are continuous (like in the NTM and continuous D-NTM), then the access may be too diffuse, especially early on during training. This can hurt especially the *writing* operation, since a diffused write operation will overwrite a large fraction of the memory at each step, yielding fast vanishing of the memories (and gradients). On the other hand, discrete addressing, as used in the discrete D-NTM, should be able to perform this search through the past, but prevents us from using straight backpropagation for learning how to choose the address.

We investigate the flow of the gradients and how the wormhole connections introduced by the controller affects it. Our results show that the wormhole connections created by the controller of the MANN can significantly reduce the effects of the vanishing gradients by shortening the paths that the signal needs to travel between the dependencies. We also discuss how the MANNs can generalize to sequences longer than the ones seen during the training.

In a discrete D-NTM, the controller must learn to read from and write into the external memory by itself and additionally, it should also learn the reader/writer synchronization. This can make learning more challenging. In spite of this difficulty, Gulcehre et al. (2016) reported that the discrete D-NTM can learn faster than the continuous D-NTM on some of the bAbI tasks. We provide a formal analysis of gradient flow in MANNs based on discrete addressing and justify this result. In this chapter, we also propose a new MANN based on discrete addressing called TARDIS (Temporal Automatic Relation Discovery in Sequences). In TARDIS, memory access is based on tying the write and read heads of the model after memory is filled up. When the memory is not full, the write head stores information in memory in the sequential order.

The main characteristics of TARDIS are as follows, TARDIS is a simple memory augmented neural network model which can represent long-term dependencies efficiently by using an external memory of small size. TARDIS represents the dependencies between the hidden states inside the memory. We show both theoretically and experimentally that TARDIS fixes to a large extent the problems related to long-term dependencies. Our model can also store sub-sequences or sequence chunks into the memory. As a consequence, the controller can learn to represent the high-level temporal abstractions as well. TARDIS performs well on several structured output prediction tasks as verified in our experiments.

The idea of using external memory with attention can be justified with the concept of mental-time travel which humans do occasionally to solve daily tasks. In particular, in the cognitive science literature, the concept of chronesthesia is known to be a form of consciousness which allows human to think about time subjectively and perform mental time-travel (Tulving, 2002). TARDIS is inspired by this ability of humans which allows one to look up past memories and plan for the future using the episodic memory.

3.2 TARDIS: A Memory Augmented Neural Network

Neural network architectures with an external memory represent the memory in a matrix form, such that at each time step t the model can both read from and write to the external memory. The whole content of the external memory can be considered as a generalization of hidden state vector in a recurrent neural network. Instead of storing all the information into a single hidden state vector, our model can store them in a matrix which has a higher capacity and with more targeted ability to substantially change or use only a small subset of the memory at each

time step. The neural Turing machine (NTM) (Graves et al., 2014) is such an example of a MANN, with both reading and writing into the memory.

3.2.1 Model Outline

In this subsection, we describe the basic structure of TARDIS[i] (Temporal Automatic Relation Discovery In Sequences). TARDIS is a MANN which has an external memory matrix $\mathbf{M}_t \in \mathbb{R}^{k \times q}$ where k is the number of memory cells and q is the dimensionality of each cell. The model has an RNN controller which can read from and write to the external memory at every time step. To read from the memory, the controller generates the read weights $\mathbf{w}_t^r \in \mathbb{R}^{k \times 1}$ and the reading operation is typically achieved by computing the dot product between the read weights $\mathbf{w}_t^r$ and the memory $\mathbf{M}_t$, resulting in the content vector $\mathbf{r}_t \in \mathbb{R}^{q \times 1}$:

$$\mathbf{r}_t = (\mathbf{M}_t)^\top \mathbf{w}_t^r. \tag{3.1}$$

TARDIS uses discrete addressing and hence $\mathbf{w}_t^r$ is a one-hot vector and the dot-product chooses one of the cells in the memory matrix (Zaremba and Sutskever, 2015; Gulcehre et al., 2016). The controller generates the write weights $\mathbf{w}_t^w \in \mathbb{R}^{1 \times k}$, to write into the memory which is also a one hot vector, with discrete addressing. We will omit biases from our equations for simplicity in the rest of the chapter. Let i be the index of the non-zero entry in the one-hot vector $\mathbf{w}_t^w$, then the controller writes a linear projection of the current hidden state to the memory location $\mathbf{M}_t[i]$:

$$\mathbf{M}_t[i] = \boldsymbol{W}_m \mathbf{h}_t, \tag{3.2}$$

where $\boldsymbol{W}_m \in \mathbb{R}^{d_m \times d_h}$ is the projection matrix that projects the d_h dimensional hidden state vector to a d_m dimensional micro-state vector such that $d_h > d_m$.

At every time step, the hidden state $\mathbf{h}_t$ of the controller is also conditioned on the content $\mathbf{r}_t$ read from the memory. The wormhole connections are created by conditioning $\mathbf{h}_t$ on $\mathbf{r}_t$:

$$\mathbf{h}_t = \phi(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{r}_t). \tag{3.3}$$

As each cell in the memory is a linear projection of one of the previous hidden states, the conditioning of the controller's hidden state with the content read from the memory can be interpreted as a way of creating short-cut connections across time (from the time t' that $\mathbf{h}_{t'}$ was written to the time t when it was read through $\mathbf{r}_t$) which can help with the flow of gradients across time. This is possible because of the discrete addressing used for read and write operations.

i. Name of the model is inspired from the time-machine in a popular TV series Dr. Who.

However, the main challenge for the model is to learn proper read and write mechanisms so that it can write the hidden states of the previous time steps that will be useful for future predictions and read them at the right time step. We call this the reader/writer synchronization problem. Instead of designing complicated addressing mechanisms to mitigate the difficulty of learning how to properly address the external memory, TARDIS side-steps the reader/writer synchronization problem by using the following heuristics. For the first k time steps, our model writes the micro-states into the k cells of the memory in a sequential order. When the memory becomes full, the most effective strategy in terms of preserving the information stored in the memory would be to replace the memory cell that has been read with the micro-state generated from the hidden state of the controller after it is conditioned on the memory cell that has been read. If the model needs to perfectly retain the memory cell that it has just overwritten, the controller can in principle learn to do that by copying its read input to its write output (into the same memory cell). The pseudocode and the details of the memory update algorithm for TARDIS is presented in Algorithm 1.

Algorithm 1 Pseudocode for the controller and memory update mechanism of TARDIS.

Initialize $\mathbf{h}_0$
Initialize $\mathbf{M}_0$
for $t \in \{1, \cdots T_x\}$ **do**
    Compute the read weights $\overline{\mathbf{w}}_t^r \leftarrow \text{read}(\mathbf{h}_t, \mathbf{M}_t, \mathbf{x}_t)$
    Sample from/discretize $\overline{\mathbf{w}}_t^r$ and obtain $\mathbf{w}_t^r$
    Read from the memory, $\mathbf{r}_t \leftarrow (\mathbf{M}_t)^\top \mathbf{w}_t^r$.
    Compute a new controller hidden state, $\mathbf{h}_t \leftarrow \phi(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{r}_t)$
    if $t \leq k$ **then**
        Write into the memory, $\mathbf{M}_t[t] \leftarrow \boldsymbol{W}_m \mathbf{h}_t$
    else
        Select the memory location to write into $j \leftarrow \text{argmax}_j(\mathbf{w}_t^r[j])$
        Write into the memory, $\mathbf{M}_t[j] \leftarrow \boldsymbol{W}_m \mathbf{h}_t$

There are two missing pieces in Algorithm 1: How to generate the read weights? What is the structure of the controller function ϕ? We will answer these two questions in detail in next two sub-sections.

3.2.2 Addressing mechanism

Similar to the D-NTM, memory matrix $\mathbf{M}_t$ of TARDIS has disjoint address section $\mathbf{A}_t \in \mathbb{R}^{k \times a}$ and content section $\mathbf{C}_t \in \mathbb{R}^{k \times c}$, $\mathbf{M}_t = [\mathbf{A}_t; \mathbf{C}_t]$ and $\mathbf{M}_t \in \mathbb{R}^{k \times q}$ for $q = c + a$. However, unlike in the D-NTM, address vectors are fixed to random

sparse vectors. The controller reads both the address and the content parts of the memory, but it will only write into the content section of the memory.

The continuous read weights $\overline{\mathbf{w}}_t^r$ are generated by an MLP which uses the information coming from $\mathbf{h}_t$, $\mathbf{x}_t$, $\mathbf{M}_t$ and the usage vector $\mathbf{u}_t$ (described below). The MLP is parametrized as follows:

$$\pi_t[i] = \mathbf{a}^\top \tanh(\boldsymbol{W}_h^\gamma \mathbf{h}_t + \boldsymbol{W}_x^\gamma \mathbf{x}_t + \boldsymbol{W}_m^\gamma \mathbf{M}_t[i] + \boldsymbol{W}_u^\gamma \mathbf{u}_t) \tag{3.4}$$

$$\overline{\mathbf{w}}_t^r = \text{softmax}(\pi_t), \tag{3.5}$$

where $\{\mathbf{a}, \boldsymbol{W}_h^\gamma, \boldsymbol{W}_x^\gamma, \boldsymbol{W}_m^\gamma, \boldsymbol{W}_u^\gamma\}$ are learnable parameters. $\mathbf{w}_t^r$ is a one-hot vector obtained by either sampling from $\overline{\mathbf{w}}_t^r$ or by using argmax over $\overline{\mathbf{w}}_t^r$.

$\mathbf{u}_t$ is the usage vector which denotes the frequency of accesses to each cell in the memory. $\mathbf{u}_t$ is computed from the sum of discrete address vectors $\mathbf{w}_t^r$ and normalizing them:

$$\mathbf{u}_t = \text{norm}(\sum_{i=1}^{t-1} \mathbf{w}_i^r). \tag{3.6}$$

The norm$(\cdot)$ applied in Equation 3.6 is a simple feature-wise computation of centering and divisive variance normalization. This normalization step makes the training easier with the usage vectors. The introduction of the usage vector can help the attention mechanism to choose between the different memory cells based on their frequency of accesses to each cell of the memory. For example, if a memory cell is very rarely accessed by the controller, for the next time step, it can learn to assign more weights to those memory cells by looking into the usage vector. This way, the controller can learn an LRU access mechanism (Santoro et al., 2016; Gulcehre et al., 2016).

Further, in order to prevent the model to learn deficient addressing mechanisms, for e.g. reading the same memory cell which will not increase the memory capacity of the model, we decrease the probability of the last read memory location by subtracting 100 from the logit of $\overline{\mathbf{w}}_t^r$ for that particular memory location.

3.2.3 TARDIS Controller

We use an LSTM controller, and its gates are modified to take into account the content $\mathbf{r}_t$ of the cell read from the memory:

$$\begin{pmatrix} \mathrm{f}_t \\ \mathrm{i}_t \\ \mathrm{o}_t \end{pmatrix} = \begin{pmatrix} \texttt{sigm} \\ \texttt{sigm} \\ \texttt{sigm} \end{pmatrix} \left(\boldsymbol{W}_h \mathbf{h}_{t-1} \; + \; \boldsymbol{W}_x \mathbf{x}_t \; + \; \boldsymbol{W}_r \mathbf{r}_t \right), \tag{3.7}$$

where f_t, i_t, and o_t are the forget gate, input gate, and output gate respectively.

The cell of the LSTM controller, $\mathbf{c}_t$ is computed according to the Equation 3.8 as follows:

$$\begin{aligned}\tilde{\mathbf{c}}_t &= \tanh(\alpha_t \boldsymbol{W}_h^{\mathrm{J}} \mathbf{h}_{t-1} + \boldsymbol{W}_x^{\mathrm{J}} \mathbf{x}_t + \beta_t \boldsymbol{W}_r^{\mathrm{J}} \mathbf{r}_t), \\ \mathbf{c}_t &= \mathrm{f}_t \mathbf{c}_{t-1} + \mathrm{i}_t \tilde{\mathbf{c}}_t,\end{aligned} \tag{3.8}$$

where α_t, β_t are the scalar RESET gates which control the magnitude of the information flowing from the memory and the previous hidden states to the cell of the LSTM $\mathbf{c}_t$. By controlling the flow of information into the LSTM cell, those gates will allow the model to store the sub-sequences or chunks of sequences into the memory instead of the entire context.

We use Gumbel sigmoid (Maddison et al., 2016; Jang et al., 2016) for α_t and β_t due to its behavior close to binary:

$$\begin{pmatrix}\alpha_t \\ \beta_t\end{pmatrix} = \begin{pmatrix}\text{gumbel-sigmoid} \\ \text{gumbel-sigmoid}\end{pmatrix} \left(\begin{pmatrix}\mathbf{w}_h^{\alpha\top} \\ \mathbf{w}_h^{\beta\top}\end{pmatrix} \mathbf{h}_{t-1} + \begin{pmatrix}\mathbf{w}_x^{\alpha\top} \\ \mathbf{w}_x^{\beta\top}\end{pmatrix} \mathbf{x}_t + \begin{pmatrix}\mathbf{w}_r^{\alpha\top} \\ \mathbf{w}_r^{\beta\top}\end{pmatrix} \mathbf{r}_t \right), \tag{3.9}$$

We find Gumbel-sigmoid to be easier to train than the regular sigmoid. The temperature of the Gumbel-sigmoid is fixed to 0.3 in all our experiments.

The hidden state of the LSTM controller is computed as follows:

$$\mathbf{h}_t = \mathrm{o}_t \tanh(\mathbf{c}_t). \tag{3.10}$$

In Figure 3.1, we illustrate the interaction between the controller and the memory with various heads and components of the controller.

3.2.4 Micro-states and Long-term Dependencies

A micro-state of the LSTM for a particular time step is the summary of the information that has been stored in the LSTM controller of the model. By attending over the cells of the memory which contains previous micro-states of the LSTM, the model can explicitly learn to restore information from its past.

The controller can learn to represent high-level temporal abstractions by creating wormhole connections through the memory as illustrated in Figure 3.2. In this example, the model takes the token x_0 at the first time-step and stores its representation to the first memory cell with address a_0. In the second time-step, the controller takes x_1 as input and writes into the second memory cell with the address a_1. Furthermore, the β_1 gate blocks the connection from $\mathbf{h}_1$ to $\mathbf{h}_2$. At the

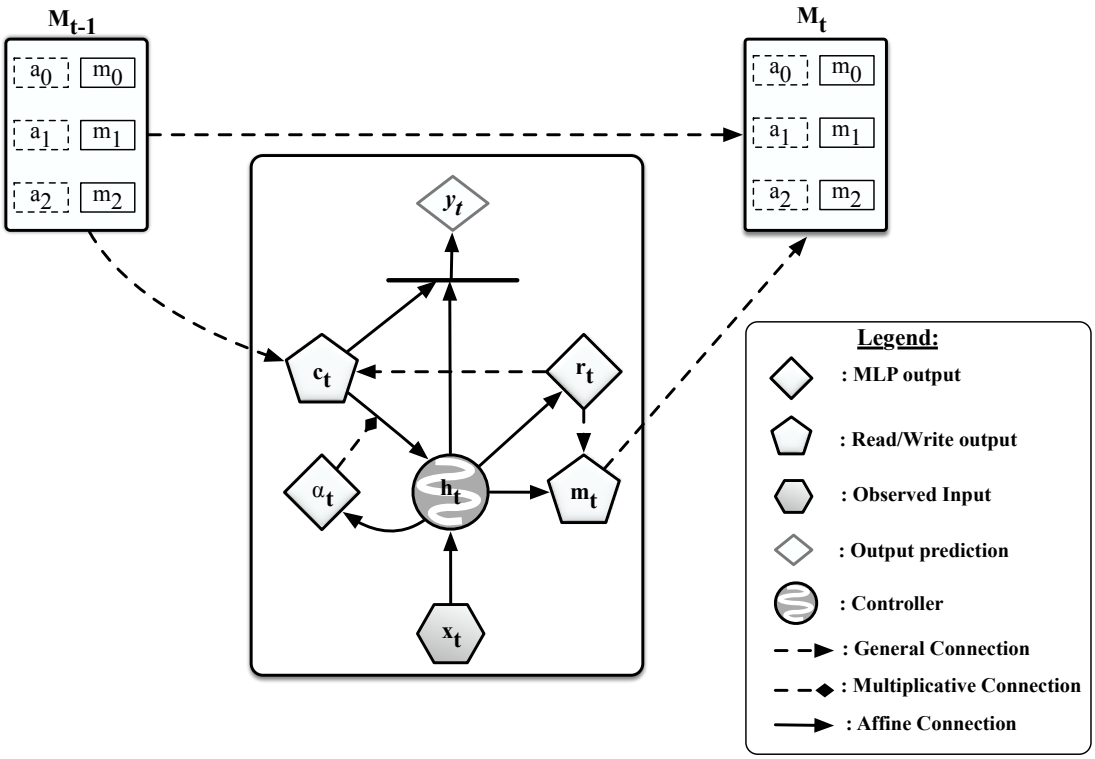


Figure 3.1 – At each time step, controller takes $\mathbf{x}_t$, the memory cell that has been read $\mathbf{r}_t$ and the hidden state of the previous timestep $\mathbf{h}_{t-1}$. Then, it generates α_t which controls the contribution of the $\mathbf{r}_t$ into the internal dynamics of the new controller's state $\mathbf{h}_t$ (We omit the β_t in this visualization). Once the memory $\mathbf{M}_t$ becomes full, discrete addressing weights $\mathbf{w}_t^r$ is generated by the controller which will be used to both read from and write into the memory. To predict the target $\mathbf{y}_t$, the model will have to use both $\mathbf{h}_t$ and $\mathbf{r}_t$.

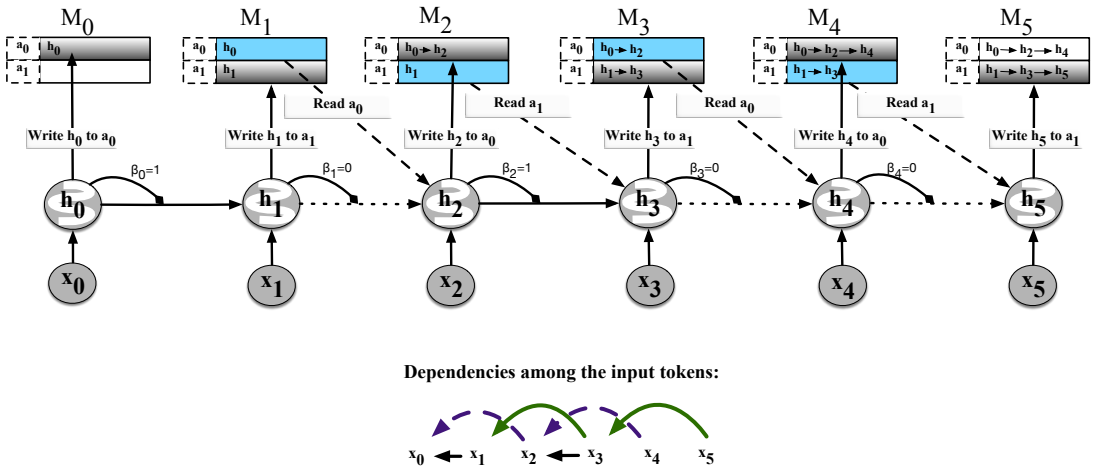


Figure 3.2 – TARDIS's controller can learn to represent the dependencies among the input tokens by choosing which cells to read and write and creating wormhole connections. $\mathbf{x}_t$ represents the input to the controller at timestep t and the $\mathbf{h}_t$ is the hidden state of the controller RNN.

third time-step, the controller starts reading. It receives x_2 as input and reads the first memory cell where micro-state of $\mathbf{h}_0$ was stored. After reading, it computes the hidden-state $\mathbf{h}_2$ and writes the micro-state of $\mathbf{h}_2$ into the first memory cell. The length of the path passing through the micro-states of $\mathbf{h}_0$ and $\mathbf{h}_2$ would be 1. The wormhole connection from $\mathbf{h}_2$ to $\mathbf{h}_0$ would skip a time-step.

A regular single-layer RNN has a fixed graphical representation of a linear-chain when considering only the connections through its recurrent states or the temporal axis. However, TARDIS is more flexible in that it can learn directed graphs with more diverse structures using the wormhole connections and the RESET gates. The directed graph that TARDIS can learn through its recurrent states have at most a degree of 4 at each vertex (maximum 2 incoming and 2 outgoing edges) and it depends on the number of cells (k) that can be stored in the memory.

In this work, we focus on a variation of TARDIS, where the controller maintains a fixed-size external memory. However as in (Cheng et al., 2016), it is possible to use a memory that grows with respect to the length of its input sequences, but that would not scale and can be more difficult to train with discrete addressing.

3.3 Training TARDIS

In this section, we explain how to train TARDIS as a language model. We use language modeling as an example application. However, we would like to highlight

that TARDIS can also be applied to any complex sequence to sequence learning task.

Consider N training examples where each example is a sequence of length T. At every time-step t, the model receives the input $\mathbf{x}_t \in \{0,1\}^{|V|}$ which is a one-hot vector of size equal to the size of the vocabulary $|V|$ and should produce the output $\mathbf{y}_t \in \{0,1\}^{|V|}$ which is also a one-hot vector of size equal to the size of the vocabulary $|V|$.

The output of the model for the i-th example and t-th time-step is computed as follows:

$$\mathbf{o}_t^i = \text{softmax}(\boldsymbol{W}^o \text{g}(\mathbf{h}_t^{(i)}, \mathbf{r}_t^{(i)})), \tag{3.11}$$

where $\boldsymbol{W}^o$ are the learnable parameters and $\text{g}(\mathbf{h}_t, \mathbf{r}_t)$ is a single layer MLP which combines both $\mathbf{h}_t$ and $\mathbf{r}_t$ as in deep fusion by Pascanu et al. (2013a). The task loss would be the categorical cross-entropy between the targets and model outputs. Superscript i denotes that the variable is the output for the i^{th} sample in the training set.

$$\mathcal{L}_{\text{model}}(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\sum_{k=1}^{|V|}\mathbf{y}_t^{(i)}[k]\log(o_t^{(i)}[k]), \tag{3.12}$$

However, the discrete decisions taken for memory access during every time-step makes the model not differentiable and hence we need to rely on approximate methods of computing gradients with respect to the discrete address vectors. In this work we explore two such approaches: REINFORCE (Williams, 1992) and the straight-through estimator (Bengio et al., 2013).

3.3.1 Using REINFORCE

REINFORCE is a likelihood-ratio method, which provides a convenient and simple way of estimating the gradients of the stochastic actions. In this work, we focus on application of REINFORCE on sequential prediction tasks, such as language modelling. For example i, let $R(\mathbf{w}_j^{r(i)})$ be the reward for the action $\mathbf{w}_j^{r(i)}$ at timestep j. We are interested in maximizing the expected return for the whole episode as defined below:

$$\mathcal{J}(\boldsymbol{\theta}) = \mathbb{E}[\sum_{j=0}^{T} R(\mathbf{w}_j^{r(i)})] \tag{3.13}$$

Ideally we would like to compute the gradients for Equation 3.13, however computing the gradient of the expectation may not be feasible. We would have to use a Monte-Carlo approximation and compute the gradients by using REINFORCE for the sequential prediction task which can be written as in Equation 3.14.

$$\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} [\sum_{j=0}^{T} (R(\mathbf{w}_j^{r(i)}) - b_j) \sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log(\overline{\mathbf{w}}_t^{r(i)})], \tag{3.14}$$

where b_j is the reward baseline. However, we can further assume that the future actions do not depend on the past rewards in the episode/trajectory and further reduce the variance of REINFORCE as in Equation 3.15.

$$\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} [\sum_{t=0}^{T} \sum_{j=t}^{T} (R(\mathbf{w}_j^{r(i)}) - b_j) \nabla_{\boldsymbol{\theta}} \log(\overline{\mathbf{w}}_t^{r(i)})], \tag{3.15}$$

In our preliminary experiments, we find that the training of the model is easier with discounted returns, instead of using the centered undiscounted return:

$$\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} [\sum_{t=0}^{T} \sum_{j=t}^{T} [\gamma^{j-t} (R(\mathbf{w}_j^{r(i)}) - b_j)] \nabla_{\boldsymbol{\theta}} \log(\overline{\mathbf{w}}_t^{r(i)})]. \tag{3.16}$$

REINFORCE Training with an Auxiliary Cost Training models with REINFORCE can be difficult, due to the variance imposed into the gradients. In recent years, researchers have developed several tricks in order to mitigate the effect of high-variance in gradients. As proposed by (Mnih and Gregor, 2014), we also use variance normalization on the REINFORCE gradients.

For TARDIS, reward at timestep j ($R(\mathbf{w}_j^{r(i)})$) is the log-likelihood of the prediction at that timestep. Our initial experiments showed that REINFORCE with this reward structure often tends to under-utilize the memory and mainly rely on the internal memory of the LSTM controller. Especially, in the beginning of training, it can just decrease the loss by relying on the memory of the controller and this can cause REINFORCE to increase the log-likelihood of the random actions.

In order to deal with this issue, instead of using the log-likelihood of the model as reward, we introduce an auxiliary cost to use as the reward R' which is computed based on predictions which are only based on the memory cell $\mathbf{r}_t$ read by the controller and not the hidden state of the controller:

$$R'(\mathbf{w}_j^{r(i)}) = \sum_{k=1}^{|V|} \mathbf{y}_j^{(i)}[k] \log(\text{softmax}(\boldsymbol{W}_r^o \bar{\mathbf{r}}_j^{(i)} + \boldsymbol{W}_x^o \mathbf{x}_j^{(i)}))[k], \tag{3.17}$$

In Equation 3.17, we only train the parameters $\{\boldsymbol{W}_r^o \in \mathbb{R}^{d_o \times d_m}, \boldsymbol{W}_x^o \in \mathbb{R}^{d_o \times d_x}\}$ where d_o is the dimensionality of the output size and d_x (for language modelling both d_o and d_x would be $d_o = |V|$) is the dimensionality of the input of the model. We do not backpropagate through $\mathbf{r}_i^{(j)}$ and thus we denote it as $\bar{\mathbf{r}}_i^{(j)}$ in our equations.

3.3.2 Using Gumbel Softmax

Training with REINFORCE can be challenging due to the high variance of the gradients. Gumbel-softmax provides a good alternative with its straight-through estimator to tackle the variance issue. Unlike (Maddison et al., 2016; Jang et al., 2016) instead of annealing the temperature or fixing it, our model learns the inverse-temperature with an MLP $\tau(\mathbf{h}_t)$ which has a single scalar output conditioned on the hidden state of the controller.

$$\tau(\mathbf{h}_t) = \text{softplus}(\mathbf{w}^{\tau\top}\mathbf{h}_t + \mathbf{b}^\tau) + 1. \tag{3.18}$$

$$\text{gumbel-softmax}(\pi_t[i]) = \text{softmax}((\pi_t[i] + \xi)\tau(\mathbf{h}_t)), \tag{3.19}$$

We replace the softmax in Equation 3.5 with gumbel-softmax defined above. During forward computation, we sample from $\overline{\mathbf{w}}_t^r$ and use the generated one-hot vector $\mathbf{w}_t^r$ for memory access. However, during backprop, we use $\overline{\mathbf{w}}_t^r$ for gradient computation and hence the entire model becomes differentiable.

Learning the temperature of the Gumbel-Softmax reduces the burden of performing extensive hyper-parameter search for the temperature.

3.4 Gradient Flow through the External Memory

In this section, we analyze the flow of the gradients through the external memory and will also investigate its efficiency in terms of dealing with the vanishing gradients problem (Hochreiter, 1991; Bengio et al., 1994). In our analysis, we also assume that the weights for the read/write heads are discrete.

We will show that the rate of the gradients vanishing through time for a memory-augmented recurrent neural network is much smaller than of a regular vanilla recurrent neural network.

Consider the MANN where the contents of the memory are linear projections of the previous hidden states as described in Equation 3.2. Let us assume that both

reading and writing operations use discrete addressing. Let the content read from the memory at time step t correspond to some memory location i:

$$\mathbf{r}_t = \mathbf{M}_t[i] = \mathbf{A}\mathbf{h}_{i_t}, \tag{3.20}$$

where $\mathbf{h}_{i_t}$ corresponds to the hidden state of the controller at some previous timestep i_t.

Now the hidden state of the controller in the external memory model can be written as,

$$\begin{aligned} \mathbf{z}_t &= \boldsymbol{W}\mathbf{h}_{t-1} + \mathbf{V}\mathbf{r}_t + \boldsymbol{U}\mathbf{x}_t, \\ \mathbf{h}_t &= \mathrm{f}(\mathbf{z}_t). \end{aligned} \tag{3.21}$$

If the controller reads $\mathbf{M}_t[i]$ at time step t and its memory content is $\mathbf{A}\mathbf{h}_{i_t}$ as described above, then the Jacobians associated with Equation 3.21 can be computed as follows:

$$\begin{aligned} \frac{\partial \mathbf{h}_{t_1}}{\partial \mathbf{h}_{t_0}} &= \prod_{t_0 < t \leq t_1} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \\ &= \left(\prod_{t_0 < t \leq t_1} \mathrm{diag}[\mathrm{f}'(\mathbf{z}_t)]\boldsymbol{W} \right) + \sum_{k=t_0}^{t_1-1} \Big(\prod_{k < t^* < t_1} \mathrm{diag}[\mathrm{f}'(\mathbf{z}_{t^*})]\boldsymbol{W} \Big)\ \mathrm{diag}[\mathrm{f}'(\mathbf{z}_k)]\mathbf{V}\mathbf{A}\frac{\partial \mathbf{h}_{i_k}}{\partial \mathbf{h}_{t_0}} \\ &\qquad + \ \mathrm{diag}[\mathrm{f}'(\mathbf{z}_{t_1})]\mathbf{V}\mathbf{A}\frac{\partial \mathbf{h}_{i_{t_1}}}{\partial \mathbf{h}_{t_0}} \qquad (3.22) \\ &= \mathbf{Q}_{t_1 t_0} + \mathbf{R}_{t_1 t_0}. \qquad (3.23) \end{aligned}$$

where $\mathbf{Q}_{t_1 t_0}$ and $\mathbf{R}_{t_1 t_0}$ are defined as,

$$\mathbf{Q}_{t_1 t_0} = \prod_{t_0 < t \leq t_1} \mathrm{diag}[\mathrm{f}'(\mathbf{z}_t)]\boldsymbol{W}, \tag{3.24}$$

$$\mathbf{R}_{t_1 t_0} = \sum_{k=t_0}^{t-1} \Big(\prod_{k < t^* < t} \mathrm{diag}[\mathrm{f}'(\mathbf{z}_{t^*})]\boldsymbol{W} \Big)\ \mathrm{diag}[\mathrm{f}'(\mathbf{z}_k)]\mathbf{V}\mathbf{A}\frac{\partial \mathbf{h}_{i_k}}{\partial \mathbf{h}_{t_0}} + \mathrm{diag}[\mathrm{f}'(\mathbf{z}_{t_1})]\mathbf{V}\mathbf{A}\frac{\partial \mathbf{h}_{i_{t_1}}}{\partial \mathbf{h}_{t_0}}. \tag{3.25}$$

As shown in Equation 3.23, Jacobians of the MANN can be rewritten as a summation of two matrices, $\mathbf{Q}_{t_1 t_0}$ and $\mathbf{R}_{t_1 t_0}$. The gradients flowing through $\mathbf{R}_{t_1 t_0}$ do not necessarily vanish through time, because it is the sum of jacobians computed over the shorter paths.

The norm of the Jacobian can be lower bounded as follows by using the Minkowski

inequality:

$$||\frac{\partial \mathbf{h}_{t_1}}{\partial \mathbf{h}_{t_0}}|| = || \prod_{t_0 < t \leq t_1} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}|| \tag{3.26}$$

$$= ||\mathbf{Q}_{t_1 t_0} + \mathbf{R}_{t_1 t_0}|| \geq ||\mathbf{R}_{t_1 t_0}|| - ||\mathbf{Q}_{t_1 t_0}||. \tag{3.27}$$

Assuming that the length of the dependency is very long $||\mathbf{Q}_{t_1 t_0}||$ would vanish to 0. Then we will have,

$$||\mathbf{Q}_{t_1 t_0} + \mathbf{R}_{t_1 t_0}|| \geq ||\mathbf{R}_{t_1 t_0}||. \tag{3.28}$$

The rate of the gradients vanishing through time depends on the length of the sequence passing through $\mathbf{R}_{t_1 t_0}$. This is typically less than the length of the sequence passing through $\mathbf{Q}_{t_1 t_0}$. Thus the gradients vanish at a smaller rate than in an RNN. In particular the rate would strictly depend on the length of the shortest paths from t_1 to t_0, because for the long enough dependencies, gradients through the longer paths would still vanish.

We can also derive an upper bound for the norm of the Jacobian as follows:

$$||\frac{\partial \mathbf{h}_{t_1}}{\partial \mathbf{h}_{t_0}}|| = || \prod_{t_0 < t \leq t_1} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}|| \tag{3.29}$$

$$= ||\mathbf{Q}_{t_1 t_0} + \mathbf{R}_{t_1 t_0}|| \leq \sigma_1(\mathbf{Q}_{t_1 t_0} + \mathbf{R}_{t_1 t_0}) \tag{3.30}$$

where $\sigma_1(\mathbf{M})$ is the largest singular value of matrix $\mathbf{M}$. Using the result from (Loyka, 2015), we can lower bound $\sigma_1(\mathbf{Q}_{t_1 t_0} + \mathbf{R}_{t_1 t_0})$ as follows:

$$\sigma_1(\mathbf{Q}_{t_1 t_0} + \mathbf{R}_{t_1 t_0}) \geq |\sigma_1(\mathbf{Q}_{t_1 t_0}) - \sigma_1(\mathbf{R}_{t_1 t_0})| \tag{3.31}$$

For long sequences we know that $\sigma_1(\mathbf{Q}_{t_1 t_0})$ will go to 0 (see equation 2.16). Hence,

$$\sigma_1(\mathbf{Q}_{t_1 t_0} + \mathbf{R}_{t_1 t_0}) \geq \sigma_1(\mathbf{R}_{t_1 t_0}) \tag{3.32}$$

The rate at which $\sigma_1(\mathbf{R}_{t_1 t_0})$ reaches zero is strictly smaller than the rate at which $\sigma_1(\mathbf{Q}_{t_1 t_0})$ reaches zero and with ideal memory access, it will not reach zero. Hence unlike vanilla RNNs, Equation 3.32 states that the upper bound of the norm of the Jacobian will not reach zero for a MANN with ideal memory access.

Theorem 1. *Consider a memory augmented neural network with T memory cells for a sequence of length T, and each hidden state of the controller is stored in*

different cells of the memory. If the prediction at time step t_1 has only a long-term dependency to t_0 and the prediction at t_1 is independent from the tokens appearing before t_0, and the memory reading mechanism is perfect, then the model will not suffer from vanishing gradients when we back-propagate from t_1 to t_0. [i]

Proof: If the input sequence has a longest-dependency on t_0 for t_1, we would only be interested in gradients propagating from t_1 to t_0 and the Jacobians from t_1 to t_0, i.e. $\frac{\partial \mathbf{h}_{t_1}}{\partial \mathbf{h}_{t_0}}$. If the controller learns a perfect reading mechanism at time step t_1 it would read memory cell where the hidden state of the RNN at time step t_0 is stored at. Thus following the Jacobians defined in the Equation 3.23, we can rewrite the Jacobians as,

$$\begin{aligned}\frac{\partial \mathbf{h}_{t_1}}{\partial \mathbf{h}_{t_0}} &= \prod_{t_0<t\leq t_1} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \\ &= \left(\prod_{t_0<t\leq t_1} \text{diag}[\text{f}'(\mathbf{z}_t)]\boldsymbol{W}\right) + \sum_{k=t_0}^{t_1-1}\Big(\prod_{k<t^*<t_1} \text{diag}[\text{f}'(\mathbf{z}_{t^*})]\boldsymbol{W}\Big)\ \text{diag}[\text{f}'(\mathbf{z}_k)]\mathbf{V}\mathbf{A}\frac{\partial \mathbf{h}_{i_k}}{\partial \mathbf{h}_{t_0}} \\ &\quad + \ \text{diag}[\text{f}'(\mathbf{z}_{t_1})]\mathbf{V}\mathbf{A}\frac{\partial \mathbf{h}_{t_0}}{\partial \mathbf{h}_{t_0}} \qquad (3.33)\end{aligned}$$

In Equation 3.33, the first two terms might vanish as $t_1 - t_0$ grows. However, the singular values of the third term do not change as $t_1 - t_0$ grows. As a result, the gradients propagated from t_1 to t_0 will not necessarily vanish through time. However, in order to obtain stable dynamics for the network, the initialization of the matrices, $\mathbf{V}$ and $\mathbf{A}$ is important. □

This analysis highlights the fact that an external memory model with optimal read/write mechanism can handle long-range dependencies much better than an RNN. However, this is applicable only when we use discrete addressing for read-/write operations. Both NTM and D-NTM still have to learn how to read and write from scratch which is a challenging optimization problem. For TARDIS, tying the read/write operations make the learning become much simpler for the model. In particular, the results of the Theorem 1 points the importance of coming up with better ways of designing attention mechanisms over the memory.

The controller of a MANN may not be able learn to use the memory efficiently. For example, some cells of the memory may remain empty or may never be read. The controller can overwrite the memory cells which have not been read. As a result the information stored in those overwritten memory cells can be lost completely. However TARDIS avoids most of these issues by the construction of the algorithm.

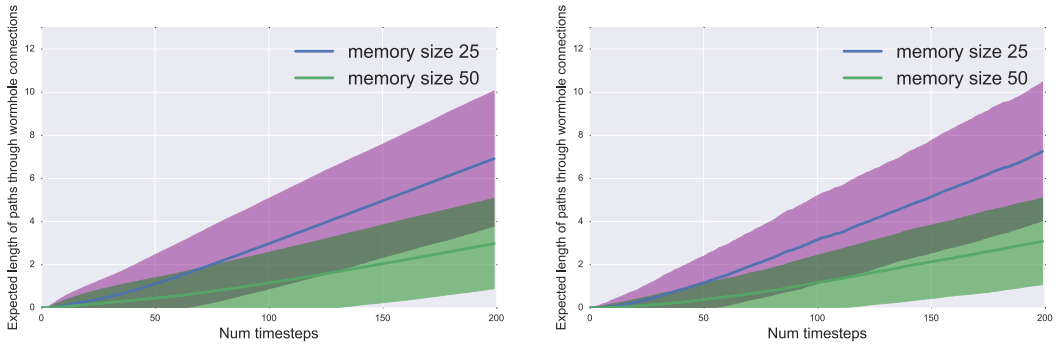
i. Let us note that, unlike the Markovian n-gram assumption, here we assume that at each time step the dependency length n can be different.

3.5 On the Length of the Paths Through the Wormhole Connections

As we have discussed in Section 3.4, the rate at which the gradients vanish for a MANN depends on the length of the paths passing along the wormhole connections. In this section we will analyse those lengths in depth for untrained models such that the model will assign uniform probability to read or write all memory cells. This will give us a better idea on how each untrained model uses the memory at the beginning of the training.

A wormhole connection can be created by reading a memory cell and writing into the same cell in TARDIS. For example, in Figure 3.2, while the actual path from $\mathbf{h}_4$ to $\mathbf{h}_0$ is of length 4, memory cell $\mathbf{a}_0$ creates a shorter path of length 2 ($\mathbf{h}_0 \rightarrow \mathbf{h}_2 \rightarrow \mathbf{h}_4$). We call the length of the actual path T and the length of the shorter path created by wormhole connections T_{mem}.

Consider a TARDIS model which has k cells in its memory. If TARDIS accesses each memory cell uniformly at random, then the probability of accessing a random cell i, $p[i] = \frac{1}{k}$. The expected length of the shorter path created by wormhole connections (T_{mem}) would be proportional to the number of reads and writes into a memory cell. For TARDIS with reader choosing a memory cell uniformly at random, this would be $T_{mem} = \sum_{i=k}^{T} p[i] = \frac{T}{k} - 1$ at the end of the sequence. We verify this result by simulating the read and write heads of TARDIS as in Figure 3.3 (a).



a) b)

Figure 3.3 – In these figures we visualized the expected path length in the memory cells for a sequence of length 200, memory size 50 with 100 simulations. a) shows the results for TARDIS and b) shows the simulation for uMANN with uniformly random read and write heads.

Now consider a MANN with separate read and write heads each accessing the memory in discrete and uniformly random fashion. Let us call it uMANN. We will compute the expected length of the shorter path created by wormhole connections

(T_{mem}) for uMANN. $\mathbf{w}_t^r$ and $\mathbf{w}_t^w$ are the read and write head weights, each sampled from a multinomial distribution with uniform probability for each memory cells respectively. Let j_t be the index of the memory cell read at timestep t. For any memory cell i, len$(\cdot)$, defined below, is a recursive function that computes the length of the path created by wormhole connections in that cell.

$$\text{len}(\mathbf{M}_t[i], i, j_t) = \begin{cases} \text{len}(\mathbf{M}_{t-1}[j_t], i, j_t) + 1 & if\ \mathbf{w}_t^w[i] = 1 \\ \text{len}(\mathbf{M}_{t-1}[i], i, j_t) & if\ \mathbf{w}_t^w[i] = 0 \end{cases} \tag{3.34}$$

It is possible to prove that $T_{mem} = \sum_t \mathbb{E}_{i,j_t}[\text{len}(\mathbf{M}_t[i], i, j_t)]$ will be $T/k - 1$ by induction for every memory cell. However, the proof assumes that when t is less than or equal to k, the length of all paths stored in the memory len$(\mathbf{M}_t[i],\ i,\ j_t)$ should be 0. We have run simulations to compute the expected path length in a memory cell of uMANN as in Figure 3.3 (b).

This analysis shows that while TARDIS with uniform read head maintains the same expected length of the shorter path created by wormhole connections as uMANN, it completely avoids the reader/writer synchronization problem.

If k is large enough, $T_{mem} << T$ should hold. In expectation, $\sigma_1(\mathbf{R}_{t_1 t_0})$ will decay proportionally to T_{mem} whereas $\sigma_1(\mathbf{Q}_{t_1 t_0})$ will decay proportional [i] to T. With ideal memory access, the rate at which $\sigma_1(\mathbf{R}_{t_1 t_0})$ reaches zero would be strictly smaller than the rate at which $\sigma_1(\mathbf{Q}_{t_1 t_0})$ reaches zero. Hence, as per Equation 3.32, the upper bound of the norm of the Jacobian will vanish at a much smaller rate. However, this result assumes that the dependencies on which the prediction relies are accessible through the memory cell which has been read by the controller.

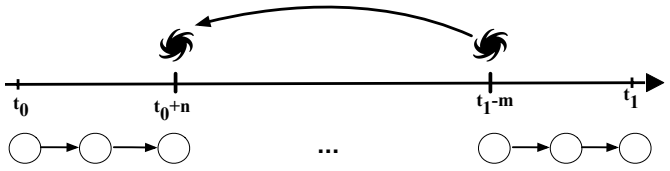


Figure 3.4 – Assuming that the prediction at t_1 depends on the t_0, a wormhole connection can shorten the path by creating a connection from $t_1 - m$ to $t_0 + n$. A wormhole connection may not directly create a connection from t_1 to t_0, but it can create shorter paths which the gradients can flow without vanishing. In this figure, we consider the case where a wormhole connection is created from $t_1 - m$ to $t_0 + n$. This connections skips all the tokens in between $t_1 - m$ and $t_0 + n$.

In the more general case, consider a MANN with $k \geq T$. The writer just fills in the memory cells in a sequential manner and the reader chooses a memory cell

i. Exponentially when the Equation 2.16 holds.

uniformly at random. Let us call this model urMANN. Let us assume that there is a dependency between two time-steps t_0 and t_1 as shown in Figure 3.4. If t_0 was taken uniformly between 0 and $t_1 - 1$, then there is a probability 0.5 that the read address invoked at time t_1 will be greater than or equal to t_0 (proof by symmetry). In that case, the expected shortest path length through that wormhole connection would be $(t_1 - t_0)/2$, but this still would not scale well. If the reader is very well trained, it could pick exactly t_0 and the path length will be 1.

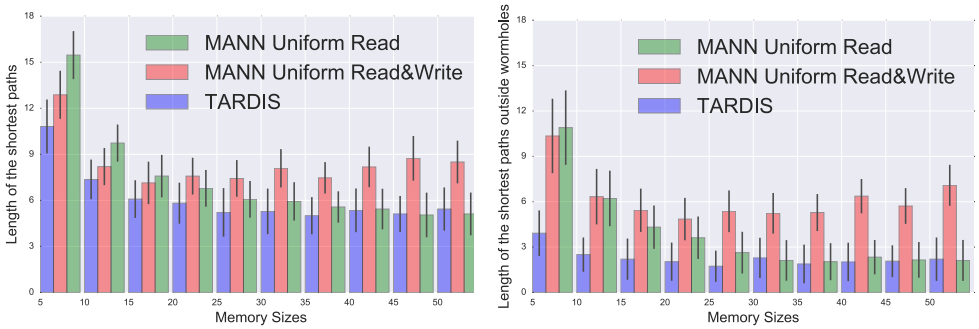
Let us consider all the paths of length less than or equal to $k + 1$ of the form in Figure 3.4. Also, let $n \leq k/2$ and $m \leq k/2$. Then, the shortest path from t_0 to t_1 now has length $n + m + 1 \leq k + 1$, using a wormhole connection that connects the state at $t_0 + n$ with the state at $t_1 - m$. There are $O(k^2)$ such paths that are realized, but we leave the distribution of the length of that shortest path as an open question. However, the probability of hitting a very short path (of length less than or equal to $k + 1$) increases exponentially with k. Let the probability of the read at $t_1 - m$ to hit the interval $(t_0,\ t_0\ +\ k/2)$ be p. Then the probability that the shorter paths over the last k reads hits that interval is $1 - (1 - p)^{k/2}$, where p is on the order of k/t_1. On the other hand, the probability of not hitting that interval approaches to 0 exponentially with k.

Figure 3.4 illustrates how wormhole connections can create shorter paths. In Figure 3.5 (b), we show that the expected length of the path travelled outside the wormhole connections obtained from the simulations decreases as the size of the memory decreases. In particular, for urMANN and TARDIS the trend is very close to exponential. As shown in Figure 3.5 (a), this also influences the total length of the paths travelled from time-step 50 to 5 as well. Writing into the memory by using weights sampled with uniform probability for all memory cells cannot use the memory as efficiently as other approaches that we compare to. In particular fixing the writing mechanism seems to be useful.

Even if the reader does not manage to learn where to read, there are many "short paths" which can considerably reduce the effect of vanishing gradients.

3.6 On Generalization over the Longer Sequences

Graves et al. (2014) have observed that LSTMs did not generalize well on sequences substantially longer than the ones seen during the training. On the other hand, a MANN such as an NTM or a D-NTM has been shown to generalize to longer sequences on a set of toy tasks.



a) b)

Figure 3.5 – We have run simulations for TARDIS, MANN with uniform read and write mechanisms (uMANN) and MANN with uniform read and write head is fixed with a heuristic (urMANN). In our simulations, we assume that there is a dependency from timestep 50 to 5. We run 200 simulations for each one of them with different memory sizes for each model. In plot a) we show the results for the expected length of the shortest path from timestep 50 to 5. In the plots, as the size of the memory gets larger for both models, the length of the shortest path decreases dramatically. In plot b), we show the expected length of the shortest path travelled outside the wormhole connections with respect to different memory sizes. TARDIS seems to use the memory more efficiently compared to other models in particular when the size of the memory is small by creating shorter paths.

We believe that the main reason for why LSTMs typically do not generalize to sequences longer than the ones that are seen during training is mainly because the hidden state of an LSTM network utilizes an unbounded history of the input sequence and as a result, its parameters are optimized using the maximum likelihood criterion to fit the sequences with lengths specific to the training examples. However, an n-gram language model or an HMM does not suffer from this issue. In comparison, an n-gram LM would use an input context with a fixed window size and an HMM has the Markov property in its latent space. As argued below, we claim that while being trained, a MANN can also learn the ability to generalize for sequences with a longer length than the ones that appear in the training set by modifying the contents of the memory and reading from it.

A regular RNN will minimize the negative log-likelihood objective function for the targets $\mathbf{y}_t$ by using the unbounded history represented by the hidden state of the RNN, and it will model the parametrized conditional distribution $p(\mathbf{y}_t|\mathbf{h}_t;\boldsymbol{\theta})$ for the prediction at time-step t, while a MANN would learn $p(\mathbf{y}_t|\mathbf{h}_t,\mathbf{r}_t;\boldsymbol{\theta})$. If we assume that $\mathbf{r}_t$ represents all the dependencies that $\mathbf{y}_t$ depends on in the input sequence, we will have $p(\mathbf{y}_t|\mathbf{h}_t,\mathbf{r}_t;\boldsymbol{\theta}) \approx p(\mathbf{y}_t|\mathbf{r}_t,\mathbf{x}_t;\boldsymbol{\theta})$ where $\mathbf{r}_t$ represents the dependencies in a limited context window that only contains paths shorter than the sequences seen during training. Due to this property, we claim that MANNs such as NTM, D-NTM or TARDIS can generalize to longer sequences more easily. In our experiments on

PennTreebank, we show that a TARDIS language model trained to maximize the log-likelihood for $p(\mathbf{y}_t|\mathbf{h}_t, \mathbf{r}_t; \boldsymbol{\theta})$ yields very close results while making predictions even while using $p(\mathbf{y}_t|\mathbf{r}_t, \mathbf{x}_t; \boldsymbol{\theta})$ instead of $p(\mathbf{y}_t|\mathbf{h}_t, \mathbf{r}_t; \boldsymbol{\theta})$ during testing. The fact that the best results on bAbI dataset obtained in (Gulcehre et al., 2016) is with a feedforward controller and similarly in (Graves et al., 2014) a feedforward controller was used to solve some of the toy tasks also confirms our hypothesis. As a result, what has been written into the memory and what has been read becomes very important to be able to generalize to the longer sequences.

3.7 Experiments

3.7.1 Character-level Language Modeling on PTB

As a preliminary study on the performance of our model we consider character-level language modelling. We have evaluated our models on the Penn TreeBank (PTB) corpus (Marcus et al., 1993) based on the train, valid and test sets used in (Mikolov et al., 2012). On this task, we are using layer-normalization (Ba et al., 2016) and recurrent dropout (Semeniuta et al., 2016) as those are also used by the SOTA results on this task. Using layer-normalization and recurrent dropout improves the performance significantly and reduces the effects of overfitting. We train our models with Adam (Kingma and Ba, 2014) over sequences of length 150. We show our results in Table 3.1.

In addition to the regular char-LM experiments, in order to confirm our hypothesis regarding the ability of MANNs to generalize to longer sequences, we have trained a language model which learns $p(\mathbf{y}_t|\mathbf{h}_t, \mathbf{r}_t; \boldsymbol{\theta})$ by using a softmax layer as described in Equation 3.11 and tested the performance using $p(\mathbf{y}_t|\mathbf{r}_t, \mathbf{x}_t; \boldsymbol{\theta})$. As in Table 3.1, the model's performance by using $p(\mathbf{y}_t|\mathbf{h}_t, \mathbf{r}_t; \boldsymbol{\theta})$ is 1.26 BPC, however by using $p(\mathbf{y}_t|\mathbf{r}_t, \mathbf{x}_t; \boldsymbol{\theta})$ it becomes 1.28 BPC. This gap is small enough to confirm our assumption that $p(\mathbf{y}_t|\mathbf{h}_t, \mathbf{r}_t; \boldsymbol{\theta}) \approx p(\mathbf{y}_t|\mathbf{r}_t, \mathbf{x}_t; \boldsymbol{\theta})$.

3.7.2 Sequential Stroke Multi-digit MNIST task

In this subsection, we introduce a new pen-stroke based sequential multi-digit MNIST prediction task as a benchmark for long term dependency modelling. We also benchmark the performance of LSTM and TARDIS on this challenging task.

Model	BPC
CW-RNN (Koutnik et al., 2014)	1.46
HF-MRNN (Sutskever et al., 2011)	1.41
ME n-gram (Mikolov et al., 2012)	1.37
BatchNorm LSTM (Cooijmans et al., 2016)	1.32
Zoneout RNN (Krueger et al., 2016)	1.27
LayerNorm LSTM (Ha et al., 2016)	1.27
LayerNorm HyperNetworks (Ha et al., 2016)	**1.23**
LayerNorm HM-LSTM & Step Fn. & Slope Annealing(Chung et al., 2016)	1.24
Our LSTM + Layer Norm + Dropout	1.28
TARDIS + REINFORCE + R	1.28
TARDIS + REINFORCE + Auxiliary Cost	1.28
TARDIS + REINFORCE + Auxiliary Cost + R	1.26
TARDIS + Gumbel Softmax + ST + R	**1.25**

Table 3.1 – Character-level language modelling results on Penn TreeBank Dataset. TARDIS with Gumbel Softmax and straight-through (ST) estimator performs better than REINFORCE and it performs competitively compared to the SOTA on this task. "+ R" notifies the use of RESET gates α and β.

Task and Dataset

Recently (de Jong, 2016) introduced an MNIST pen stroke classification task and also provided a dataset which consisted of pen stroke sequences representing the skeleton of the digits in the MNIST dataset. Each MNIST digit image $\mathcal{I}$ is represented as a sequence of quadruples $\{dx_i, dy_i, eos_i, eod_i\}_{i=1}^{T}$, where T is the number of pen strokes to define the digit, (dx_i, dy_i) denotes the pen offset from the previous to the current stroke (can be 1, -1 or 0), eos_i is a binary valued feature to denote end of stroke and eod_i is another binary valued feature to denote end of the digit. In the original dataset, the first quadruple contains absolute values (x, y) instead of offsets (dx, dy). Without loss of generality, we set the starting position (x, y) to $(0, 0)$ in our experiments. Each digit is represented by 40 strokes in average and the task is to predict the digit at the end of the stroke sequence.

While this dataset was proposed for incremental sequence learning in (de Jong, 2016), we consider the multi-digit version of this dataset to benchmark models that can handle long term dependencies. Specifically, given a sequence of pen-stroke sequences, the task is to predict the sequence of digits corresponding to each pen-stroke sequences in the given order. This is a challenging task since it requires the model to learn to predict the digit based on the pen-stroke sequence, count the number of digits and remember them and generate them in the same order after seeing all the strokes. In our experiments we consider 3 versions of this task

with 5, 10, and 15 digit sequences respectively. We generated 200,000 training data points by randomly sampling digits from the training set of the MNIST dataset. Similarly we generated 20,000 validation and test data points by randomly sampling digits from the validation set and test set of the MNIST dataset respectively. Average length of the stroke sequences in each of these tasks are 199, 399, and 599 respectively.

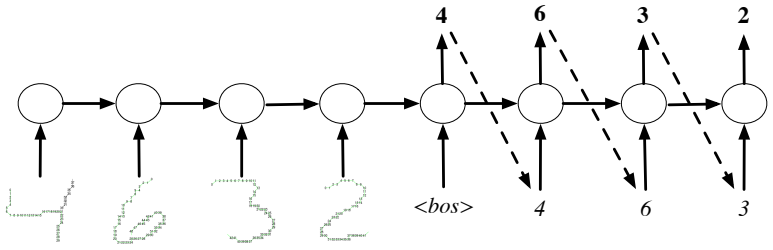


Figure 3.6 – An illustration of the sequential MNIST strokes task with multiple digits. The network is first provided with the sequence of strokes information for each MNIST digits (location information) as input, during the prediction the network tries to predict the MNIST digits that it has just seen. When the model tries to predict, the predictions from the previous time steps are fed back into the network. For the first time step, the model receives a special <*bos*> token which is fed into the model in the first time step when the prediction starts.

Results

We benchmark the performance of LSTM and TARDIS in this new task. Both models receive the sequence of pen strokes and at the end of the sequence are expected to generate the sequence of digits followed by a particular <*bos*> token. The tasks is illustrated in Figure 3.6. We evaluate the models based on per-digit error rate. We also compare the performance of TARDIS with REINFORCE with that of TARDIS with gumbel softmax. All the models were trained for the same number of updates with early stopping based on the per-digit error rate in the validation set. Results for all 3 versions of the task are reported in Table-3.2. From the table, we can see that TARDIS performs better than the LSTM in all three versions of the task. Also TARDIS with gumbel-softmax performs slightly better than TARDIS with REINFORCE, which is consistent with our other experiments.

We also compare the learning curves of all three models in Figure-3.7. From the figure we can see that TARDIS learns to solve the task faster than the LSTM by effectively utilizing the given memory slots. Also, TARDIS with gumbel softmax converges faster than TARDIS with REINFORCE.

Model	5-digits	10-digits	15-digits
LSTM	3.00%	3.54%	8.81%
TARDIS with REINFORCE	2.09%	2.56%	3.67%
TARDIS with gumbel softmax	**1.89%**	**2.23%**	**3.09%**

Table 3.2 – Per-digit based test error in sequential stroke multi-digit MNIST task with 5,10, and 15 digits.

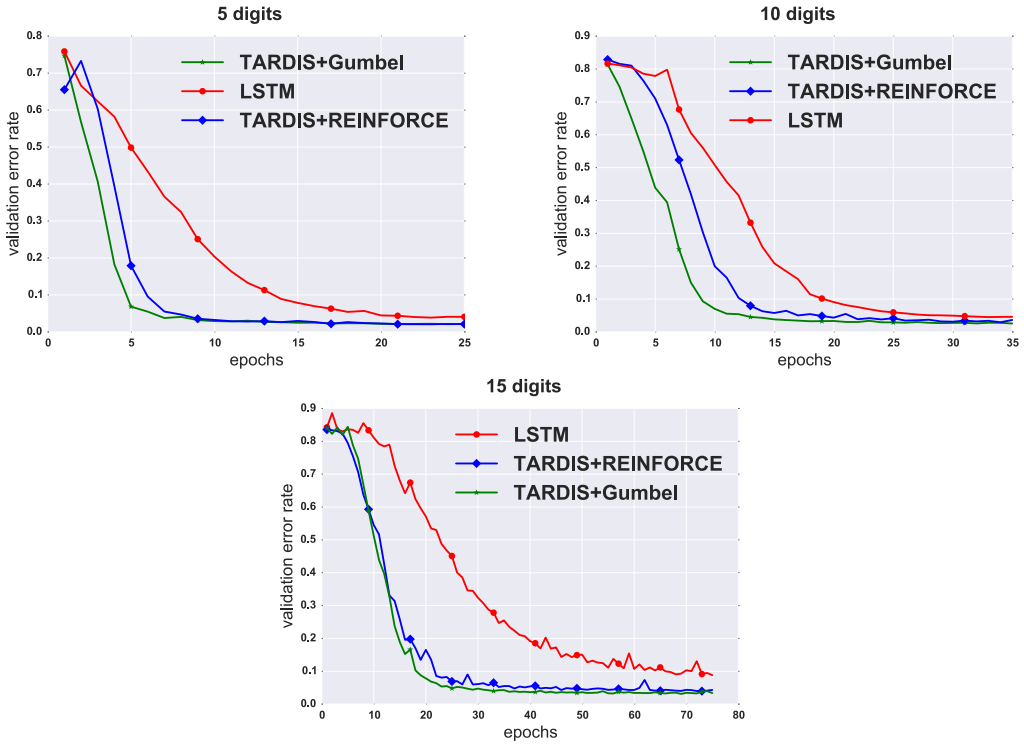


Figure 3.7 – Learning curves for LSTM and TARDIS for sequential stroke multi-digit MNIST task with 5, 10, and 15 digits respectively.

3.7.3 NTM Tasks

Graves et al. (2014) proposed associative recall and the copy tasks to evaluate a model's ability to learn simple algorithms and generalize to sequences longer than the ones seen during the training. We trained a TARDIS model with 4 features for the address and 32 features for the memory content part of the model. We used a model with hidden states of size 120. Our model uses a memory of size 16. We train our model with Adam and used a learning rate of 3e-3. We show the results of our model in Table 3.3. TARDIS was able to solve both tasks, both with Gumbel-softmax and REINFORCE.

	Copy Task	Associative Recall
D-NTM cont. (Gulcehre et al., 2016)	Success	Success
D-NTM discrete (Gulcehre et al., 2016)	Success	Failure
NTM (Graves et al., 2014)	Success	Success
TARDIS + Gumbel Softmax + ST	Success	Success
TARDIS REINFORCE + Auxiliary Cost	Success	Success

Table 3.3 – In this table, we consider a model to be successful on copy or associative recall if its validation cost (binary cross-entropy) is lower than 0.02 over the sequences of maximum length seen during the training. We set the threshold to 0.02 to determine whether a model is successful on a task as in (Gulcehre et al., 2016).

3.7.4 Stanford Natural Language Inference

Bowman et al. (2015) proposed a new task to test a machine learning algorithm's ability to infer whether two given sentences entail, contradict or are neutral (semantic independence) from each other. However, this task can be considered a long-term dependency task if the premise and the hypothesis are presented to the model in sequential order, as also explored by Rocktäschel et al. (2015). Because the model should learn the dependency relationship between the hypothesis and the premise. Our model first reads the premise, then the hypothesis. At the end of the hypothesis, the model predicts whether the premise and the hypothesis contradicts or entails. The model proposed by Rocktäschel et al. (2015), applies attention over its previous hidden states over the premise when it reads the hypothesis. In that sense their model can still be considered to have some task-specific architectural design choice. TARDIS and our baseline LSTM models do not include any task-specific architectural design choices. In Table 3.4, we compare the results of different models. Our model performs better than the other models we compare against in this experiment. However recently it has been shown that with architectural tweaks, it is possible to design a model specifically to solve this task and achieve 88.2% test accuracy (Chen et al., 2016).

3.8 Conclusion

In this chapter, we proposed a simple and efficient memory augmented neural network model which can perform well both on algorithmic tasks and more realistic tasks. Unlike previous approaches, we show better performance on real-world NLP tasks, such as language modelling and SNLI. We have also proposed a new task to measure the performance of the models dealing with long-term dependencies.

Model	Test Accuracy
Word by Word Attention (Rocktäschel et al., 2015)	83.5
Word by Word Attention two-way (Rocktäschel et al., 2015)	83.2
LSTM + LayerNorm + Dropout	81.7
TARDIS + REINFORCE + Auxiliary Cost	82.4
TARDIS + Gumbel Softmax + ST	**84.3**
NTM	80.2
D-NTM	80.9
LSTM (Bowman et al., 2015)	82.3
NTM + Layer-Norm + Dropout	81.8
D-NTM + Layer-Norm + Dropout	82.3

Table 3.4 – Comparisons of different baselines on SNLI Task.

We provide a detailed analysis on the effects of using external memory for the gradients and justify the reason why MANNs generalize better on sequences longer than the ones seen in the training set. We have also shown that the gradients will vanish at a much slower rate (if they vanish) when an external memory is being used. Our theoretical results should encourage further studies in the direction of developing better attention mechanisms that can create *wormhole* connections efficiently.

4 Non-saturating Recurrent Units

In this chapter, we introduce our second solution to mitigating vanishing gradients, called Non-saturating Recurrent Units (NRUs) (Chandar et al., 2019).

4.1 Introduction

Successful architectures, like the LSTM (Hochreiter and Schmidhuber, 1997) and GRU (Cho et al., 2014), alleviate vanishing gradients by allowing information to skip transition operators (and their activation functions) via an additive path that serves as memory. Both gates and transition operators use bounded activation functions (sigmoid, tanh). These help keep representations stable but attenuate gradients in two ways: they are contractive and they saturate at the bounds. Activation functions whose gradients contract as they saturate are typically referred to as saturating nonlinearities; in that sense, rectified linear units (ReLU) (Nair and Hinton, 2010) are non-saturating and thus help reduce vanishing gradients in deep feed-forward networks (Krizhevsky et al., 2012; Xu et al., 2015).

Saturating activation functions still dominate in the literature for recurrent neural networks, in part because of the success of LSTM on sequential problems (Sutskever et al., 2014; Vinyals and Le, 2015; Merity et al., 2017; Mnih et al., 2016). As these functions saturate, gradients contract. In addition, while saturated gates may push more information through memory and improve gradient flow across a long sequence, the gradient to the gating units themselves vanishes. We expect that non-saturating activation functions and gates could improve the modelling of long-term dependencies by avoiding these issues. We propose a non-saturating recurrent unit (NRU) that forgoes both saturating activation functions and saturating gates. We present a new architecture with the rectified linear unit (ReLU) and demonstrate that it is well equipped to process data with long distance dependencies, without sacrificing performance on other tasks.

Saturating gating functions introduce a trade-off where distant gradient propagation may occur at the cost of vanishing updates to the gating mechanism. We seek to avoid this trade-off between distant gradient propagation but not updateable gates (saturated gates) and short gradient propagation but updateable gates

(non-saturated gates). To that end, we propose the Non-saturating Recurrent Unit — a gated recurrent unit with no saturating activation functions.

4.2 Non-saturating Recurrent Units

At any time-step t, the NRU takes some input $\mathbf{x}_t$ and updates its hidden state as follows:

$$\mathbf{h}_t = f(\boldsymbol{W}_h \mathbf{h}_{t-1} + \boldsymbol{W}_i \mathbf{x}_t + \boldsymbol{W}_c \mathbf{m}_{t-1}) \tag{4.1}$$

where $\mathbf{h}_{t-1} \in \mathbb{R}^h$ is the previous hidden state, $\mathbf{m}_{t-1} \in \mathbb{R}^m$ is the previous memory cell, and f is a ReLU non-linearity. The memory cell in NRU is a flat vector similar to the cell vector in the LSTM. However, in NRU, the hidden state and the memory cell need not be of the same size. We allow the memory cell to be larger than the hidden state to hold more information.

At every time step t, memory cells are updated as follows:

$$\mathbf{m}_t = \mathbf{m}_{t-1} + \sum_{i=1}^{k_1} \alpha_i \mathbf{v}_i^w - \sum_{i=1}^{k_2} \beta_i \mathbf{v}_i^e \tag{4.2}$$

where

- $\mathbf{m}_t \in \mathbb{R}^m$ is the memory vector at time t.
- k_1 is the number of writing heads and k_2 is the number of erasing heads. We often set $k_1 = k_2 = k$.
- $\mathbf{v}_i^w \in \mathbb{R}^m$ is a normalized vector which defines where to write in the memory. Similarly $\mathbf{v}_i^e \in \mathbb{R}^m$ is a normalized vector which defines where to erase in the memory.
- α_i is a scalar value which represents the content which is written in the memory along the direction of $\mathbf{v}_i^w$. Similarly, β_i is a scalar value which represents the content which is removed from the memory along the direction of $\mathbf{v}_i^e$.

Intuitively, the network first computes a low-dimensional projection of the current hidden state by generating a k-dimensional α vector. Then it writes each unit of this k-dimensional vector along k different basis directions ($\mathbf{v}_i^w$s). These k basis directions specify *where to write* as continuous-valued vectors and hence there is no need for saturating non-linearities or discrete addressing schemes. The network also similarly erases some content from the memory (by using β and $\mathbf{v}_i^e$s).

Scalars α_i and β_i are computed as follows:

$$\alpha_i = f_\alpha(\mathbf{x}_t, \mathbf{h}_t, \mathbf{m}_{t-1}) \tag{4.3}$$
$$\beta_i = f_\beta(\mathbf{x}_t, \mathbf{h}_t, \mathbf{m}_{t-1}) \tag{4.4}$$

where f_α and f_β are linear functions followed by an optional ReLU non-linearity. Vectors $\mathbf{v}_i^w$ and $\mathbf{v}_i^e$ are computed as follows:

$$\mathbf{v}_i^w = f_w(\mathbf{x}_t, \mathbf{h}_t, \mathbf{m}_{t-1}) \tag{4.5}$$
$$\mathbf{v}_i^e = f_e(\mathbf{x}_t, \mathbf{h}_t, \mathbf{m}_{t-1}) \tag{4.6}$$

where f_w and f_e are linear functions followed by an optional ReLU non-linearity followed by an explicit normalization. We used L_5 normalization in all of our experiments since it performed slightly better than L_2 normalization in our initial experiments.

Each $\mathbf{v}$ vector is m-dimensional and there are $2k$ such vectors that need to be generated. This requires a lot of parameters which could create problems both in terms of computational cost and in terms of overfitting. We reduce this large number of parameters by using the following outer product trick to factorize the $\mathbf{v}$ vectors:

$$\mathbf{p}_i^w = f_{w_1}(\mathbf{x}_t, \mathbf{h}_t, \mathbf{m}_{t-1}) \in \mathbb{R}^{\sqrt{m}} \tag{4.7}$$
$$\mathbf{q}_i^w = f_{w_2}(\mathbf{x}_t, \mathbf{h}_t, \mathbf{m}_{t-1}) \in \mathbb{R}^{\sqrt{m}} \tag{4.8}$$
$$\mathbf{v}_i^w = g(\text{vec}(\mathbf{p}_i^w \mathbf{q}_i^{wT})) \tag{4.9}$$

where f_{w_1} and f_{w_2} are linear functions, vec() vectorizes a given matrix, g is an optional ReLU non-linearity followed by an explicit normalization. Thus, instead of producing an m-dimensional vector for the direction, we only need to produce a $2\sqrt{m}$-dimensional vector which requires substantially fewer parameters and scales more reasonably. We can further reduce the number of parameters by generating a $2\sqrt{km}$-dimensional vector instead of $2k\sqrt{m}$-dimensional vector and then use the outer product trick to generate the required km-dimensional vector. We do this trick in our implementation of NRU.

If there is no final ReLU activation function while computing α, β, and $\mathbf{v}$, then there is no distinction between write and erase heads. For example, either β_i or $\mathbf{v}_i^e$ could become negative, changing the erasing operation into one that adds to the memory. Having an explicit ReLU activation in all these terms forces the architecture to use writing heads to add information to the memory and erasing heads to erase information from the memory. However, we treat this enforcement as optional and in some experiments, we let the network decide how to use the heads by removing these ReLU activations.

4.2.1 Discussion

In vanilla RNNs, the hidden state acts as the memory of the network. So there is always a contradiction between stability (which requires small enough $\boldsymbol{W}_h$ to avoid explosions) and long-term storage (which requires high enough $\boldsymbol{W}_h$ to avoid vanishing gradients). This contradiction is exposed in a ReLU RNN which avoids the saturation from gates at the cost of network stability. LSTM and GRU reduce this problem by introducing memory in the form of information skipped forward across transition operators, with gates determining which information is skipped. However, the gradient on the gating units themselves vanishes when the unit activations saturate. Since distant gradient propagation across memory is aided by gates that are locked to ON (or OFF), this introduces an unfortunate trade-off where either the gate mechanism receives updates or gradients are skipped across many transition operators.

NRU also maintains a separate memory vector like the LSTM and GRU. Unlike the LSTM and GRU, NRU does not have saturating gates. Even if the paths purely through hidden states h vanish (with small enough $\boldsymbol{W}_h$), the actual long term memories are stored in memory vector $\mathbf{m}$ which has additive updates and hence no vanishing gradient.

Although the explicit normalization in vectors $\mathbf{v}_i^w$ and $\mathbf{v}_i^e$ may cause gradient saturation for large values, we focus on the nonlinearities because of their outsized effect on vanishing gradients. In comparison, the effect of normalization is less pronounced than that of explicitly saturating nonlinearities like tanh or sigmoid which are also contractive based on our experiments. Also, normalization has been employed to avoid the saturation regimes of these nonlinearities (Ioffe and Szegedy, 2015; Cooijmans et al., 2016). Furthermore, we observe that foregoing these nonlinearities allows the NRU to converge much faster than other models.

4.3 Experiments

In this section, we compare the performance of NRU networks with several other RNN architectures in three synthetic tasks (5 including variants) and two real tasks. Specifically, we consider the following extensive list of recurrent architectures: RNN with orthogonal initialization (RNN-orth), RNN with identity initialization (RNN-id), LSTM (Hochreiter and Schmidhuber, 1997), LSTM with chrono initialization (LSTM-chrono) (Tallec and Ollivier, 2018), GRU (Cho et al., 2014), JANET (van der Westhuizen and Lasenby, 2018), SRU (Oliva et al., 2017), EURNN (Jing et al., 2017b), GORU (Jing et al., 2017a). We used the FFT-like algorithm for the orthogonal transition operators in EURNN and GORU as it tends

to be much faster than the tunable algorithm, both proposed by Jing et al. (2017b). While these two algorithms have a low theoretical computational complexity, they are difficult to parallelize.

We mention common design choices for all the experiments here: RNN-orth and RNN-id were highly unstable in all our experiments and we found that adding layer normalization helps. Hence all our RNN-orth and RNN-id experiments use layer normalization. We did not see any significant benefit in using layer normalization for other architectures and hence it is turned off by default for other architectures. NRU with linear writing and erasing heads performed better than with ReLU based heads in all our experiments except the language model experiment. Hence we used linear writing and erasing heads unless otherwise mentioned. We used the Adam optimizer (Kingma and Ba, 2014) with a default learning rate of 0.001 in all our experiments. We clipped the gradients by norm value of 1 for all models except GORU and EURNN since their transition operators do not expand norm. We used a batch size of 10 for most tasks, unless otherwise stated.

Table-4.1 summarizes the results of this section. Out of 10 different architectures that we considered, NRU is the only model that performs among the top 2 models across all 7 tasks. The code for the NRU Cell is available at https://github.com/apsarath/NRU.

Model	in Top-1	in Top-2
RNN-orth	1	1
RNN-id	1	1
LSTM	1	1
LSTM-chrono	1	1
GRU	1	2
JANET	1	3
SRU	1	1
EURNN	2	3
GORU	0	1
NRU	**4**	**7**

Table 4.1 – Number of tasks where the models are in top-1 and top-2. Maximum of 7 tasks. Note that there are ties between models for some tasks so the column for top-1 performance would not sum up to 7.

4.3.1 Copying Memory Task

The copying memory task was introduced in Hochreiter and Schmidhuber (1997) as a synthetic task to test the network's ability to remember information over many time-steps. The task is defined as follows. Consider n different symbols. In the

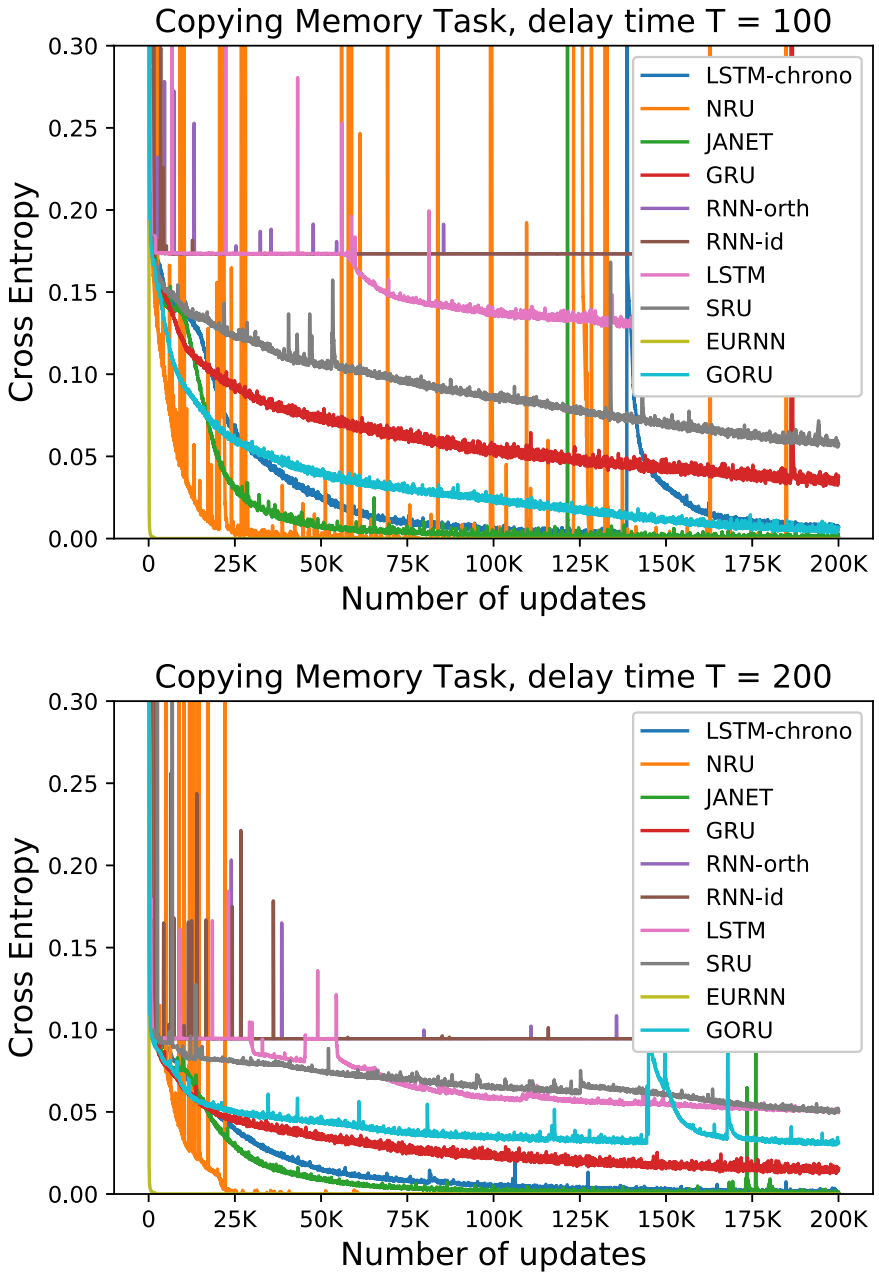


Figure 4.1 – Copying memory task for T = 100 (in top) and T = 200 (in bottom). Cross-entropy for random baseline : 0.17 and 0.09 for T=100 and T=200 respectively.

first k time-steps, the network receives an initial sequence of k random symbols from the set of n symbols sampled with replacement. Then the network receives a "blank" symbol for $T-1$ steps followed by a "start recall" marker. The network should learn to predict a "blank" symbol, followed by the initial sequence after the marker. Following Arjovsky et al. (2016), we set $n = 8$ and $k = 10$. The copying task is a pathologically difficult long-term dependency task where the output at the end of the sequence depends on the beginning of the sequence. We can vary the dependency length by varying the length of the sequence T. A memoryless model is expected to achieve a sub-optimal solution which predicts a constant sequence after the marker, for any input (referred to as the "baseline" performance). The cross-entropy for such a model would be $\frac{klogn}{T+2n}$ (Arjovsky et al., 2016).

We trained all models with approximately the same number of parameters ($\sim$23.5k), in an online fashion where every mini-batch is dynamically generated. We consider $T = 100$ and $T = 200$. In Figure 4.1 we plot the cross-entropy loss for all the models. Unsurprisingly, EURNN solves the task in a few hundred updates as it is close to an optimal architecture tuned for this task (Henaff et al., 2016). Conversely, RNN-orth and RNN-id get stuck at baseline performance. NRU converges faster than all other models, followed by JANET which requires two to three times more updates to solve the task.

We observed the change in the memory vector across the time steps in Figure-4.2. We can see that the network has learnt to add information into the memory in the beginning of the sequence and then it does not access the memory until it sees the marker. Then it makes changes in the memory in the last 10 time steps to copy the sequence. Note that NRU has to make changes in the memory even while copying the sequence since there is no location based addressing of the memory.

We performed additional experiments following the observation by Henaff et al. (2016) that gated models like the LSTM outperform a simple non-gated orthogonal network (similar to the EURNN) when the time lag T is varied in the copying task. This variable length task highlights the non-generality of the solution learned by models like EURNN. Only models with a dynamic gating mechanism can solve this problem. In Figure-4.3, we plot the cross-entropy loss for all the models. The proposed NRU model is the fastest to solve this task, followed by JANET, while the EURNN performs poorly, as expected according to Henaff et al. (2016). These two experiments highlight the fact that NRU can store information longer than other recurrent architectures. Unlike EURNN which behaves like a fixed clock mechanism, NRU learns a gating function to lock the information in the memory as long as needed. This is similar to the behaviour of other gated architectures. However, NRU beats other gated architectures mainly due to better gradient flow which results in faster learning. Figure-4.4 shows that NRU converges significantly faster than its strongest competitors (JANET and LSTM-chrono) in all the 4 tasks.

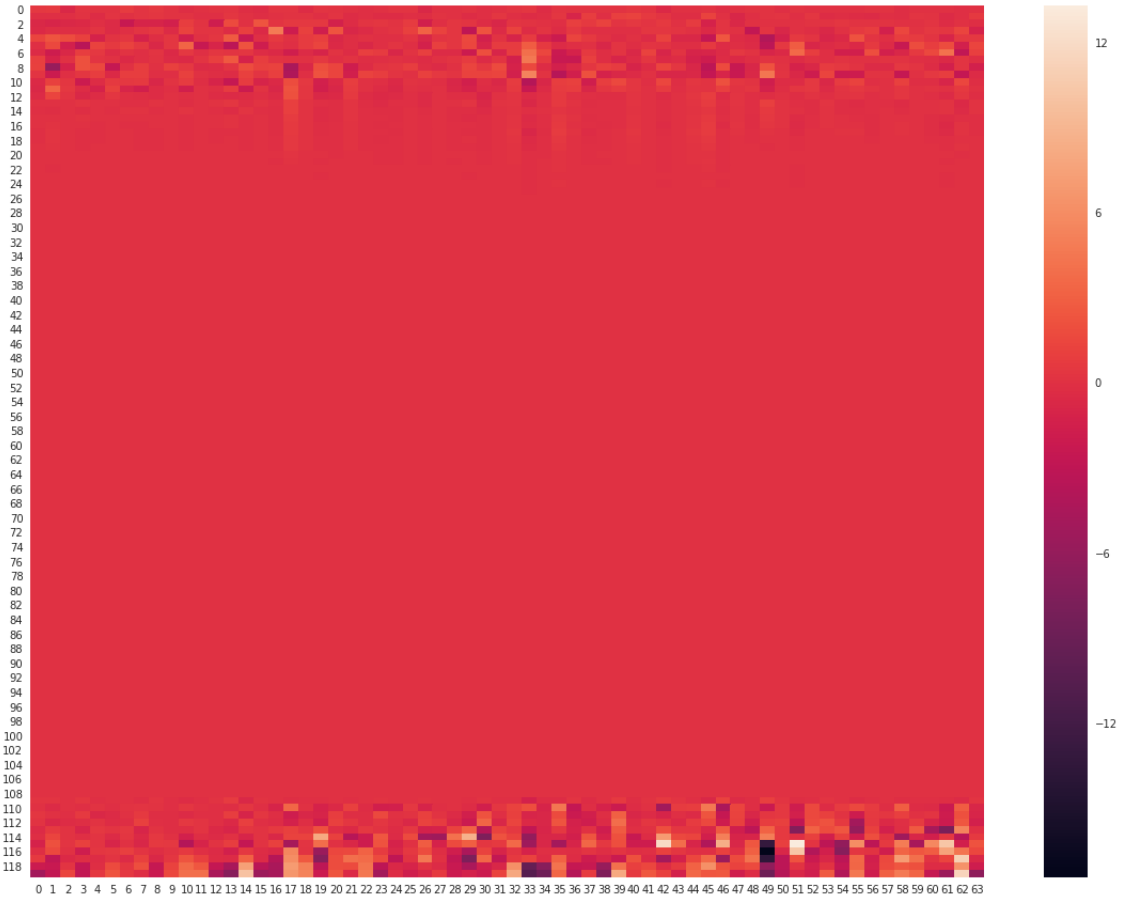


Figure 4.2 – Change in the content of the NRU memory vector for the copying memory task with T=100. We see that the network has learnt to use the memory in the first 10 time steps to store the sequence. Then it does not access the memory until it sees the marker. Then it starts accessing the memory to generate the sequence.

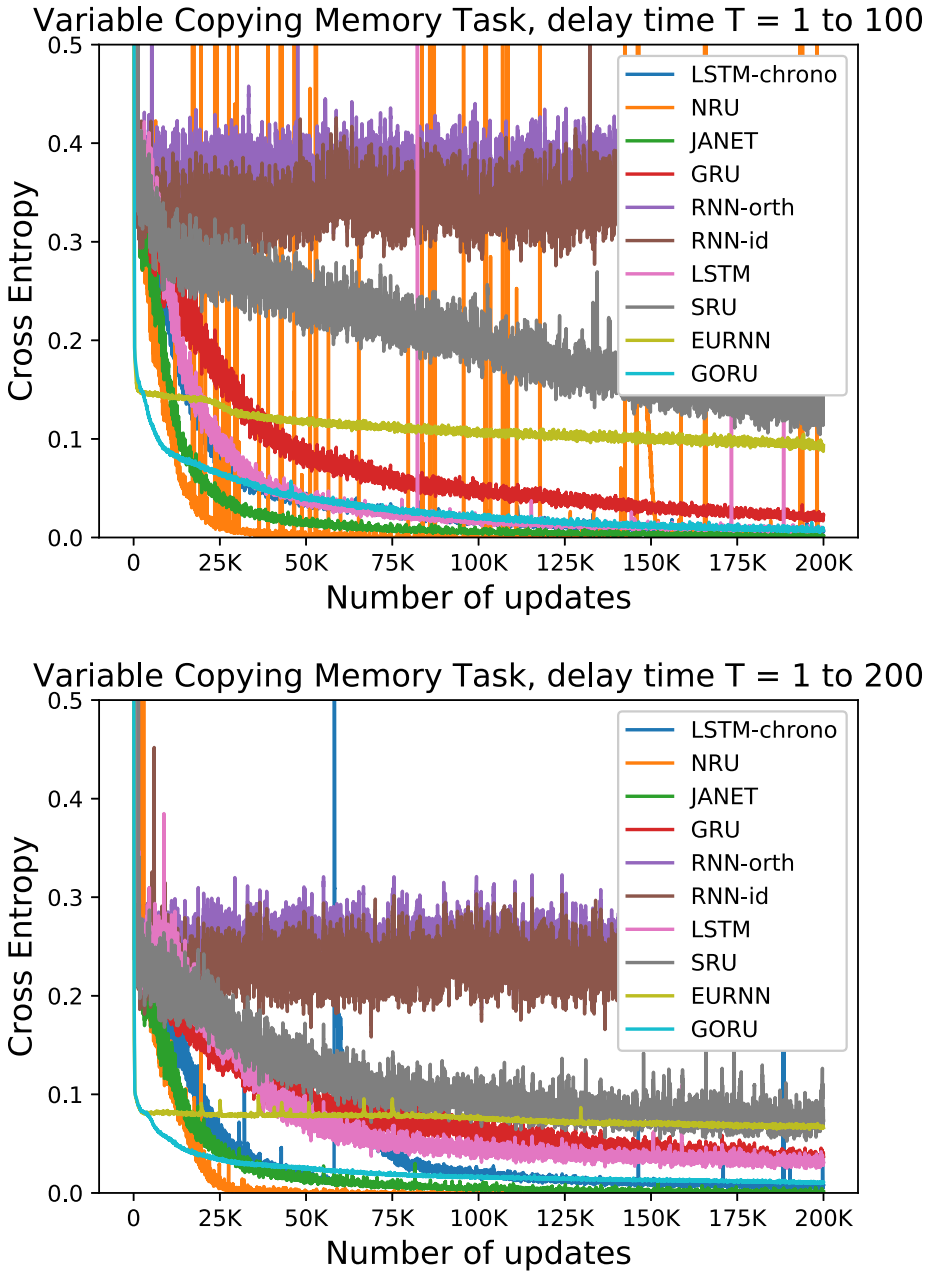


Figure 4.3 – Variable Copying memory task for T = 100 (in left) and T = 200 (in right).

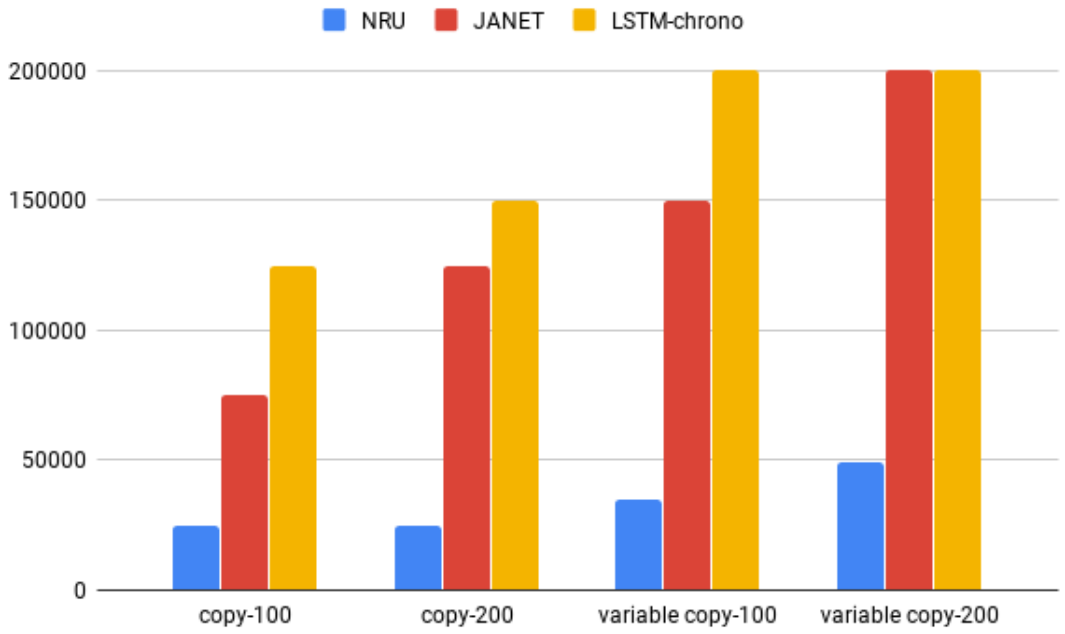


Figure 4.4 – Comparison of top-3 models w.r.t the number of the steps to converge for different tasks. NRU converges significantly faster than JANET and LSTM-chrono.

4.3.2 Denoising Task

The denoising task was introduced by Jing et al. (2017a) to test both the memorization and forgetting ability of RNN architectures. This is similar to the copying memory task. However, the data points are located randomly in a long noisy sequence. Consider again an alphabet of n different symbols from which k random symbols are sampled with replacement. These symbols are then separated by random lengths of strings composed of a "noise" symbol, in a sequence of total length T. The network is tasked with predicting the k symbols upon encountering a "start recall" marker, after the length T input sequence. Again we set $n = 8$ and $k = 10$. This task requires both an ability to learn long term dependencies and also an ability to filter out unwanted information (considered noise).

The experimental procedure is exactly the same as described for the copying memory task. In Figure 4.5 we plot the cross-entropy loss for all the models for $T = 100$. In this task, all the models converge to the solution except EURNN. While NRU learns faster in the beginning, all the algorithms converge after approximately the same number of updates.

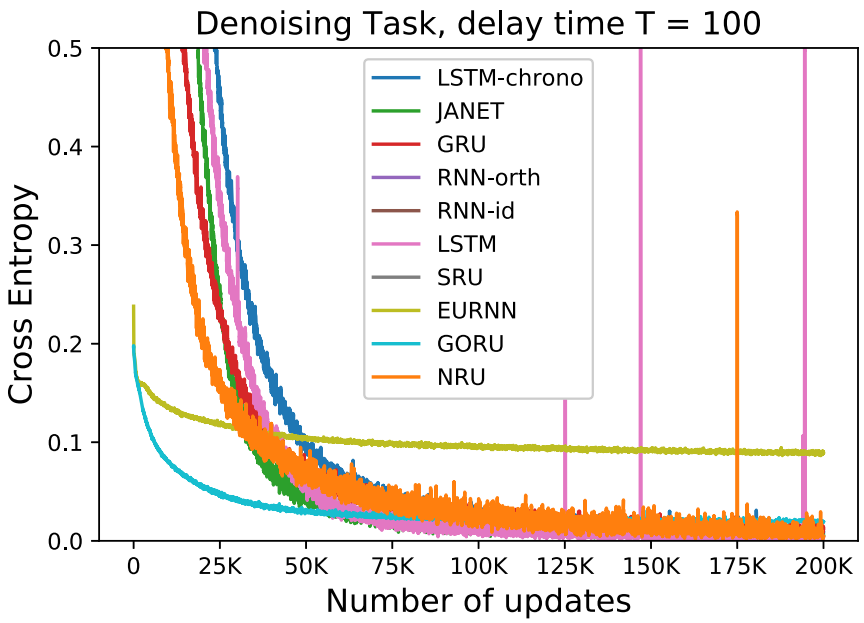


Figure 4.5 – Denoising task for T = 100.

4.3.3 Character Level Language Modelling

Going beyond synthetic data, we consider character level language modelling with the Penn Treebank Corpus (PTB) (Marcus et al., 1993). In this task, the network is fed one character per time step and the goal is to predict the next character. Again we made sure that all the networks have approximately the same capacity (∼2.15M parameters). We use a batch size of 128 and perform truncated backpropagation through time, every 150 time steps. We evaluate according to bits per character (BPC) and accuracy.

All models were trained for 20 epochs and evaluated on the test set after selecting for each the model state which yields the lowest BPC on the validation set. The test set BPC and accuracy are reported in Table-4.2. We did not add drop-out or batch normalization to any model. From the table, we can see that GRU is the best performing model, followed closely by NRU. We note that language modelling does not require very long term dependencies. This is supported by the fact that changing the additive memory updates in NRU to multiplicative updates does not hurt performance (it actually improves it). This is further supported by our observation that setting T_{max} to 50 was better than setting T_{max} to 150 in chrono-initialization for LSTM and JANET. All the best performing NRU models used ReLU activations in the writing and erasing heads.

Model	BPC	Accuracy
LSTM	1.48	67.98
LSTM-chrono	1.51	67.41
GRU	**1.45**	**69.07**
JANET	1.48	68.50
GORU	1.53	67.60
EURNN	1.77	63.00
NRU	1.47	68.48

Table 4.2 – Bits Per Character (BPC) and Accuracy in test set for character level language modelling in PTB.

4.3.4 Permuted Sequential MNIST

The Permuted Sequential MNIST task was introduced by Le et al. (2015) as a benchmark task to measure the performance of RNNs in modelling complex long term dependencies. In a sequential MNIST task, all the 784 pixels of an MNIST digit are fed to the network one pixel at a time and the network must classify the digit in the 785th time step. Permuted sequential MNIST (psMNIST) makes the problem harder by applying a fixed permutation to the pixel sequence, thus introducing longer term dependencies between pixels in a more complex order.

For this task, we used a batch size of 100. All the networks have approximately the same number of parameters (∼165k). This corresponds to 200 to 400 hidden units for most of the architectures. Since the FFT-style algorithm used with EURNN requires few parameters, we used a large 1024 unit hidden state still achieving fewer parameters (∼17k parameters) at maximum memory usage. We report validation and test accuracy in Table-4.3 and plot the validation curves in Figure-4.6. On this task, NRU performs better than all the other architectures, followed by the EURNN. The good performance of the EURNN could be attributed to its large hidden state, since it is trivial to store all 784 input values in 1024 hidden units. This model excels at preserving information, so performance is bounded by the classification performance of the output layer. While NRU remains stable, RNN-orth and RNN-id are not stable as seen from the learning curve. Surprisingly, LSTM-chrono is not performing better than regular LSTM.

4.3.5 Model Analysis

Stability: While we use gradient clipping to limit exploding gradients in NRU, we observed gradient spikes in some of our experiments. We suspect that the network recovers due to consistent gradient flow. To verify that the additive nature

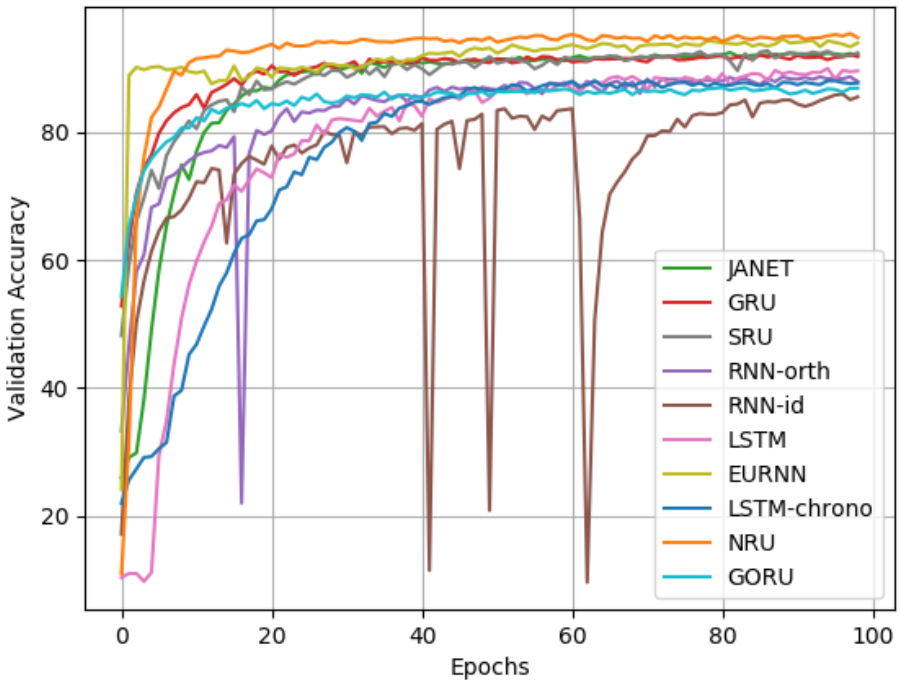


Figure 4.6 – Validation curve for psMNIST task.

Model	valid	test
RNN-orth	88.70	89.26
RNN-id	85.98	86.13
LSTM	90.01	89.86
LSTM-chrono	88.10	88.43
GRU	92.16	92.39
JANET	92.50	91.94
SRU	92.79	92.49
EURNN	94.60	94.50
GORU	86.90	87.00
NRU	**95.46**	**95.38**

Table 4.3 – Validation and test set accuracy for psMNIST task.

of the memory does not cause memory to explode for longer time steps, we performed a sanity check experiment where we trained the model on the copy task (with T=100, 200, and 500) with random labels and observed that training did not diverge. We observed some instabilities with all the models when training with longer time steps ($T = 2000$). With NRU, we observed that the model converged faster and was more stable with a higher memory size. For instance, our model converged almost twice as early when we increased the memory size from 64 to 144.

Forgetting Ability: We performed another sanity check experiment to gauge the forgetting ability of our model. We trained it on the copy task but reset the memory only once every k= 2, 5, and 10 examples and observed that NRU learned to reset the memory in the beginning of every example.

Gradient Flow: To give an overview of the gradient flow across different time steps during training, we compared the total gradient norms in the copying memory task for the top 3 models: NRU, JANET and LSTM-Chrono (Figure 4.7). We see that the NRU's gradient norm is considerably higher during the initial stages of the training while the other model's gradient norms rise after about 25k steps. After 25k steps, the drop in the NRU gradient norm coincides with the model's convergence, as expected. We expect that this ease of gradient flow in NRU serves as an additional evidence that NRU can model long-term dependencies better than other architectures.

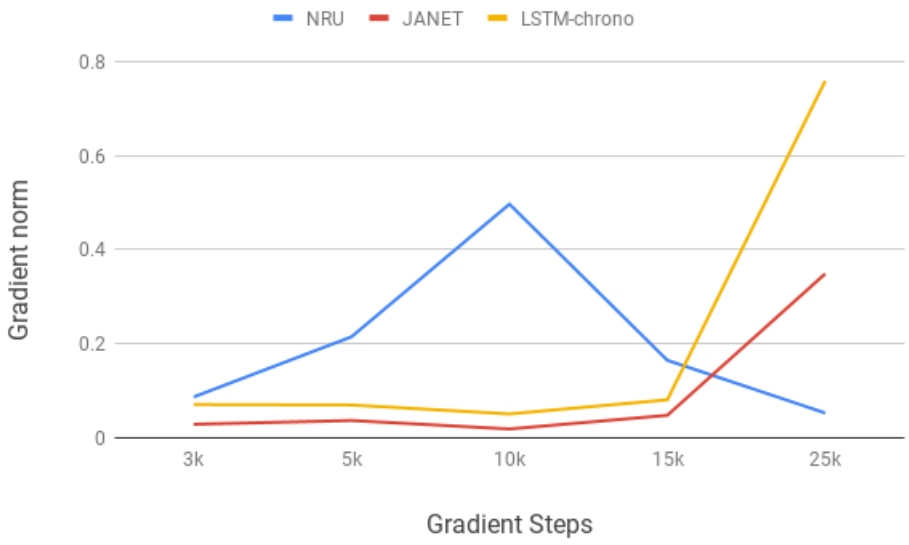


Figure 4.7 – Gradient norm comparison with JANET and LSTM-chrono across the training steps. We observe significantly higher gradient norms for NRU during the initial stages compared to JANET or LSTM-chrono. As expected, NRU's gradient norms decline after about 25k steps since the model has converged.

Hyper-parameter Sensitivity Analysis

NRU has three major hyper-parameters: memory size (m), number of heads (k) and hidden state size (h). To understand the effect of these design choices, we varied each hyper-paramteter by fixing other hyper-parameters for the psMNIST task. The baseline model was $m = 256, k = 4, h = 200$. We varied $k \in \{1, 4, 9, 16\}$, $m \in \{100, 256, 400\}$, and $h \in \{50, 100, 200, 300\}$. We trained all the models for a fixed number of 40 epochs. Validation curves are plotted in Figure-4.8. It appears that increasing k is helpful, but yields diminishing returns above 4 or 9. For a hidden state size of 200, a larger memory size (like 400) makes the learning difficult in the beginning, though all the models converge to a similar solution. This may be due to the increased number of parameters. On the other hand, for a memory size of 256, small hidden states appear detrimental.

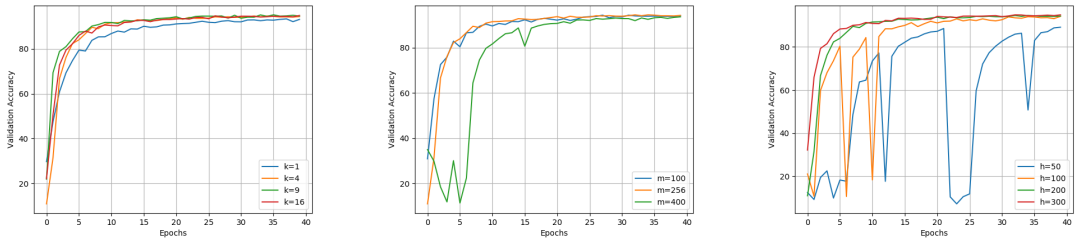


Figure 4.8 – Effect of varying the number of heads (left), memory size (middle), and hidden state size (right) in psMNIST task.

4.4 Conclusion

In this chapter, we introduce Non-saturating Recurrent Units (NRUs) for modelling long term dependencies in sequential problems. The gating mechanism in NRU is additive (like in LSTM and GRU) and non-saturating (unlike in LSTM and GRU). This results in better gradient flow for longer durations. We present empirical evidence in support of non-saturating gates in the NRU with (1) improved performance on long term dependency tasks, (2) higher gradient norms, and (3) faster convergence when compared to baseline models. NRU was the best performing general purpose model in all of the long-term dependency tasks that we considered and is competitive with other gated architectures in short-term dependency tasks. This work opens the door to other potential non-saturating gated recurrent network architectures.

We would also like to apply NRU in several real world sequential problems in natural language processing and reinforcement learning where the LSTM is the default choice for recurrent computation.

5 On Training Recurrent Neural Networks for Lifelong Learning

In the last two chapters, we explored the problem of learning long-term dependencies that arises when training recurrent neural networks with very long sequences. The focus was on single-task setting. In this chapter, we explore the challenges that arise while training RNNs in a multi-task lifelong learning setting. The results reported in this chapter appears in (Sodhani et al., 2019).

5.1 Introduction

Lifelong machine learning considers systems that can learn many tasks (from one or more domains) over a lifetime (Thrun, 1998; Silver et al., 2013). This has several names and manifestations in the literature: incremental learning (Solomonoff, 1989), continual learning (Ring, 1997), explanation-based learning (Thrun, 1996, 2012), never-ending learning (Carlson et al., 2010), etc. The underlying idea motivating these efforts is the following: lifelong learning systems would be more effective at learning and retaining knowledge across different tasks. By exploiting similarity across tasks, they would be able to obtain better priors for the task at hand. Lifelong learning techniques are very important for training intelligent autonomous agents that would need to operate and make decisions over extended periods of time. These characteristics are especially important in the industrial setups where the deployed machine learning models are being updated frequently with new incoming data whose distribution need not match the data on which the model was originally trained.

The lifelong learning paradigm is not just restricted to the multi-task setting with clear task boundaries. In real life, the system may have no control over what task it receives at any given time step. In such situations, there is no clear task boundary. Lifelong learning is also relevant when the system is learning just a single task but the data distribution changes over time.

Lifelong learning is an extremely challenging task for machine learning models because of two primary reasons:

1. **Catastrophic Forgetting**: As the model is trained on a new task (or a new data distribution), it is likely to forget the knowledge it acquired from the previous tasks (or data distributions). This phenomenon is also known as *catastrophic interference* (McCloskey and Cohen, 1989).
2. **Capacity Saturation**: Any parametric model, however large, can only have a fixed amount of representational capacity, to begin with. Given that we want the model to retain knowledge as it progresses through multiple tasks, the model would eventually run out of capacity to store the knowledge acquired in the successive tasks. The only way for it to continue learning, while retaining previous knowledge, is to increase its capacity on the fly.

Catastrophic forgetting and capacity saturation are related issues. In fact, capacity saturation can lead to catastrophic forgetting. But it is not the only cause for catastrophic forgetting. As the model is trained on one data distribution for a long time, it can forget its "learning" from the previous data distributions irrespective of how much effective capacity it has. While an under-capacity model could be more susceptible to catastrophic forgetting, having sufficient capacity (by say using very large models) does not protect against catastrophic forgetting (as we demonstrate in section 5.5). Interestingly, a model that is immune to catastrophic forgetting could be more susceptible to capacity saturation (as it uses more of its capacity to retain the previously acquired knowledge). We demonstrate this effect as well in section 5.5. It is important to think about both catastrophic forgetting and capacity saturation together, as solving just one problem does not take care of the other problem. Further, the role of capacity saturation and capacity expansion in lifelong learning is an under-explored topic.

Motivated by these challenges, we compile a list of desirable properties that a model should fulfill to be deemed suitable for lifelong learning settings:

1. **Knowledge Retention** - As the model learns to solve new tasks, it should not forget how to solve the previous tasks.
2. **Knowledge Transfer** - The model should be able to reuse the knowledge acquired during previous tasks to solve the current task. If the tasks are related, this knowledge transfer would lead to faster learning and better generalization over the lifetime of the model.
3. **Parameter Efficiency** - The number of parameters in the model should ideally be bounded, or grow at-most sub-linearly as new tasks are added.
4. **Model Expansion** - The model should be able to increase its capacity on the fly by "expanding" itself.

The model expansion characteristic comes with additional constraints: In a true lifelong learning setting, the model would experience a continual stream of training

data that cannot be stored. Hence, any model would, at best, have access to only a small sample of the historical data. In such a setting, we cannot rely on past examples to train the expanded model from scratch and a zero-shot knowledge transfer is desired. Considering the parameter efficiency and the model expansion qualities together implies that we would also want the computational and memory costs of the model to increase only sublinearily as the model trains on new tasks.

We propose to unify the Gradient Episodic Memory (GEM) model (Lopez-Paz and Ranzato, 2017) and the *Net2Net* framework (Chen et al., 2015) to develop a model suitable for lifelong learning. The GEM model provides a mechanism to alleviate catastrophic forgetting, while allowing for improvement in the previous tasks by beneficial backward transfer of knowledge. *Net2Net* is a technique for transferring knowledge from a smaller, trained neural network to another larger, untrained neural network. We discuss both these models in detail in the Related Work (section 5.2).

One reason hindering research in lifelong learning is the absence of standardized training and evaluation benchmarks. For instance, the vision community benefited immensely from the availability of the ImageNet dataset (Deng et al., 2009) and we believe that availability of a standardized benchmark would help to propel and streamline research in the domain of lifelong learning. Creating a good benchmark set up to study different aspects of lifelong learning is extremely challenging. Lomonaco and Maltoni (2017) proposed a new benchmark for Continuous Object Recognition (CORe50) in the context of computer vision. Lopez-Paz and Ranzato (2017) considered different variants of MNIST and CIFAR-100 datasets for lifelong supervised learning. These benchmarks help study-specific challenges like catastrophic forgetting by abstracting out the other challenges, but they are quite far from a real-life setting. Another limitation of the existing benchmarks is that they are largely focused on non-sequential tasks and there has been no such benchmark available for lifelong learning in the context of sequential supervised learning. Sequential supervised learning, like reinforcement learning, is a sequential task and hence more challenging than one step supervised learning tasks. However, unlike reinforcement learning, the setup is still supervised and hence makes it easier to focus on the challenges in lifelong learning in isolation from the challenges in reinforcement learning.

In this work, we propose a curriculum-based, simple and intuitive benchmark for evaluating lifelong learning models in the context of sequential supervised learning. We consider a single task setting where the model starts with the first data distribution (the simplest data distribution) and subsequently progresses to the more difficult data distributions. We can consider each data distribution as a task by itself. Each task has well-defined criteria of completion and the model can start training on a task only after learning over all the previous tasks in the curriculum. Each time the model finishes a task, it is evaluated on all the tasks in the

curriculum (including the tasks that it has not been trained on so far) so as to compare the performance of the model in terms of both catastrophic forgetting (for the previously seen tasks) and generalization (to unseen tasks).

If the model fails to learn a task (as per pre-defined criteria of success), we expand the model and let it train on the current task again. The expanded model is again evaluated on all the tasks just like the regular, unexpanded model. Performing this evaluation step enables us to analyze and understand how the model expansion step affects the model's capabilities in terms of generalization and catastrophic forgetting. We describe the benchmark and the different tasks in detail in the Tasks and Setup section(section 5.3).

The main contributions of this work are as follows:

1. We tackle the two main challenges of lifelong learning by unifying Gradient Episodic Memory (a lifelong learning technique to alleviate catastrophic forgetting) with *Net2Net* (a capacity expansion technique).
2. We propose a simple benchmark of tasks for training and evaluating models for learning sequential problems in the lifelong learning setting.
3. We show that both GEM and *Net2Net* which are originally proposed for feed-forward architectures are indeed useful for recurrent neural networks as well.
4. We evaluate the proposed unified model on the proposed benchmark and show that the unified model is better suited to the lifelong learning setting as compared to the two constituent models.

5.2 Related Work

We review the prominent works dealing with catastrophic forgetting, capacity saturation and model expansion, as these are the important aspects of lifelong learning.

5.2.1 Catastrophic Forgetting

Much of the work in the domain of catastrophic forgetting can be broadly classified into two approaches:

1. **Model Regularization**: A common and useful strategy is to freeze parts of the model as it trains on successive tasks. This can be seen as locking in the knowledge about how to solve different tasks in different parts of the model so

that training on the subsequent tasks cannot interfere with this knowledge. Sometimes, the weights are not completely frozen and are regularized to not change *too much* as the model trains across different tasks. This approach is adopted by elastic weight consolidation (EWC) (Kirkpatrick et al., 2016). As the model trains through the sequence of tasks, learning is slowed down for weights which are important to the previous tasks. Liu et al. (2018) extended this model by reparameterizing the network to approximately diagonalize the Fisher information matrix of the network parameters. This reparameterization leads to a factorized rotation of the parameter space and makes the diagonal Fisher Information Matrix assumption (of the EWC model) more applicable. Chaudhry et al. (2018) presented RWalk, a generalization of EWC and Path Integral (Zenke et al., 2017) with a theoretically grounded KL-divergence based perspective along with several new metrics. One downside of such approaches is the loss in the effective trainable capacity of the model as more and more model parameters are regularized over time. This may seem counter-intuitive given the desirable properties that we want the lifelong learning systems to have (section 5.1), but the different objectives can be in tension.

2. **Rehearsing using previous examples**: When learning on a given task, the model is also shown examples from the previous tasks. This *rehearsal* setup (Silver and Mercer, 2002) can help in two ways - if the tasks are related, training on multiple tasks helps in transferring knowledge across the tasks. If the tasks are unrelated, the setup still helps to protect against catastrophic forgetting. Rebuffi et al. (2017) proposed the *iCaRL* model which focuses on the class-incremental learning setting where as the number of classes (in the classification system) increases, the model is shown examples from the previous tasks. Generally, this strategy requires persisting some training examples per task. In practice, the cost of persisting some data samples (in terms of memory requirements) is much smaller than the memory requirements of the model. Though, in the rehearsal setup, the computational cost of training the model increases with each new task as the model has to *rehearse* on the previous tasks as well.

Mensink et al. (2012) proposed the *Nearest Mean Classifier* (NCM) model in the context of large scale, multi-class image classification. The idea is to use distance-based classifiers where a training example is assigned to the class which is "nearest" to it. The setup allows adding new classes and new training examples to existing classes at a near-zero cost. Thus the system can be updated on the fly as more training data becomes available. Further, the model could periodically be trained on the complete dataset (collected thus far). Li and Hoiem (2016) proposed the *Learning without Forgetting* (LwF) approach in the context of computer vision tasks. The idea is to divide the model into different components. Some of these

components are shared between different tasks and some of the components are task-specific. When a new task is introduced, first the existing network is used to make predictions for the data corresponding to the new task. These predictions are used as the "ground-truth" labels to compute a regularization loss that ensures that training on the new task does not affect the model's performance on the previous task. Then a new task-specific component is added to the network and the network is trained to minimize the sum of loss on the current task and the regularisation loss. The "addition" of new components per task makes the LwF model parameter inefficient.

Li and Hoiem (2016) proposed to use the distillation principle (Hinton et al., 2015) to incrementally train a single network for learning multiple tasks by using data only from the current task. Lee et al. (2017) proposed incremental moment matching (IMM) which incrementally matches the moment of the posterior distribution of the neural network which is trained on the first and the second task, respectively. While this approach seems to give strong results, it is evaluated only on datasets with very few tasks. Serrà et al. (2018) proposed to use hard attention targets (HAT) to learn pathways in a given base network using the ID of the given task. The pathways are used to obtain the task-specific networks. The limitation of this approach is that it requires knowledge about the current task ID.

The recently proposed Gradient Episodic Memory approach (Lopez-Paz and Ranzato, 2017) outperforms many of these models while enabling positive transfer on the previous tasks. It uses an episodic memory which stores a subset of the observed examples from each task. When training on a given task, an additional constraint is added such that the loss on the data corresponding to the previous tasks does not increase though it may or may not decrease. One limitation of the model is the need to compute gradients corresponding to the previous task at each learning iteration. Given that GEM needs to store only a few examples per task (in our experiments, we stored just one batch of examples), the storage cost is negligible. Given the strong performance and low memory cost, we use GEM as the first component of our unified model.

5.2.2 Capacity Saturation and Model Expansion

The problem of capacity saturation and model expansion has been extensively studied from different perspectives. Some works explored model expansion as a mean of transferring knowledge from a small network to a large network, to ease the training of deep neural networks (Gutstein et al., 2008; Furlanello et al., 2018). Analogously, the idea of distilling knowledge from a larger network to a smaller network has been explored in (Hinton et al., 2015; Romero et al., 2014). The majority of these approaches focus on training the new network on a single supervised

task where the data distribution does not change much and the previous examples can be reused several times. This is not possible in a true online lifelong learning setting, where the model experiences a continual stream of training data and has no access to previously seen examples again.

Chen et al. (2015) proposed using function-preserving transformations to expand a small, trained network (referred to as the teacher network) into a large, untrained network (referred to as the student network). Their primary motivation was to accelerate the training of large neural networks by first training small neural networks (which are easier and faster to train) and then transferring their knowledge to larger neural networks. The paper evaluated the technique in the context of single task supervised learning and mentioned continual learning as one of the motivations. Given that *Net2Net* enables the zero-shot transfer of knowledge to the expanded network, we use this idea of function preserving transformations to achieve zero-shot knowledge transfer in the proposed unified model.

Rusu et al. (2016) proposed the idea of Progressive Networks that explicitly supports the transfer of features across a sequence of tasks. The progressive network starts with a single *column* or model (neural network) and new *columns* are added as more tasks are encountered. Each time the network learned a task, the newly added *column* (corresponding to the task) is "frozen" to ensure that "knowledge" cannot be lost. Each new *column* uses the layer-wise output from all the previous columns to explicitly enable transfer learning. As a new *column* is added per task, the number of *columns* (and hence the number of network parameters) increases quadratically with the number of tasks. Further, when a new *column* is added, only a fraction of the new capacity is actually utilized, thus each new column is increasingly underutilized. Another limitation is that during training, the model explicitly needs to know when a new task starts so that a new *column* can be added to the network. Similarly, during inference, the network needs to know the task to which the current data point belongs to so that it knows which *column* to use. Aljundi et al. (2016) build upon this idea and use a Network of Experts where each expert model is trained for one task. During inference, a set of gating autoencoders are used to select the expert model to query. This gating mechanism helps to reduce the dependence on knowing the task label for the test data points.

Mallya et al. (2018) proposed the piggyback approach to train the model on a base task and then learn different bit masks (for parameters in the base network) for different tasks. One advantage as compared to Progressive Networks is that only 1 bit is added per parameter of the base model (as compared to 1 new parameter per parameter of the base model). The shortcoming of the approach, however, is that knowledge can be transferred only from the base task to the subsequent tasks and not between different subsequent tasks.

Table 5.1 compares the different lifelong learning models in terms of the desirable

Table 5.1 – Comparison of different models in terms of the desirable properties they fulfill.

Property / Model	Knowledge Retention	Knowledge Transfer	Parameter Efficiency	Model Expansion
EWC	✓		✓	
IMM	✓	✓	✓	
iCaRL	✓		✓	
NCM	✓		✓	
LwF	✓			
GEM	✓	✓	✓	
Net2Net			✓	✓
Progressive Nets	✓	✓		✓
Network of Experts	✓	✓		✓
Piggyback	✓			✓
HAT	✓	✓	✓	

properties they fulfill. The table makes it very easy to determine which combination of models could be feasible. If we choose a parameter-inefficient model, then the unified model will be parameter inefficient which is clearly undesirable. Further, we want at least one of the component models to have the expansion property so that the capacity can be increased on the fly. This narrows down the choice of the first model to *Net2Net*. Since this model lacks both knowledge retention and knowledge transfer, we could pick either IMM, GEM or HAT as the second component. IMM is evaluated for very few tasks while HAT requires the task IDs to be known beforehand. In contrast, GEM is reported to work well for a large number of tasks (Lopez-Paz and Ranzato, 2017). Given these considerations, we choose GEM as the second component. Now, the unified model has all the four properties.

5.3 Tasks and Benchmark

In this section, we describe the tasks, training, and the evaluation setup that we proposed for benchmarking lifelong learning models in the context of sequential supervised learning. In a true lifelong learning setting, the training distribution can change arbitrarily and no explicit demarcation exists between the data distribution corresponding to the different tasks. This makes it extremely hard to study how model properties like catastrophic forgetting and generalization capability evolve with the training. We sidestep these challenges by using a curriculum-based, simple and intuitive setup where we can have full control over the training data distributions. This setup gives us explicit control over when the model experiences different

data distributions and in what order. Specifically, we train the models in the curriculum style setup (Bengio et al., 2009) where the tasks are ordered by difficulty. We discuss the rationale behind using the curriculum approach in section 5.3.5. We consider the following three tasks as part of the benchmark.

5.3.1 Copy Task

The copy task is an algorithmic task introduced by Graves et al. (2014) to test whether the training network can learn to store and recall a long sequence of random vectors. Specifically, the network is presented with a sequence of randomly initialized, seven-bit vectors. Each such vector is followed by an eighth bit which serves as a delimiter flag. This flag is zero at all time-steps except for the end of the sequence. The network is trained to generate the entire sequence except the delimiter flag. The different levels are defined by considering input sequences of different lengths. We start with input sequences of length 5 and increase the sequence length in steps of 3 and go till the maximum sequence length of 62 (20 levels). We can consider arbitrarily large sequences but we restrict ourselves to maximum sequence length of 62 as none of the considered models were able to learn all these sequences. We report the bit-wise accuracy metric.

5.3.2 Associative Recall Task

The associative recall task is another algorithmic task introduced by Graves et al. (2014). In this task, the network is shown a list of items where each item is a sequence of randomly initialized 8-bit binary vectors, bounded on the left and the right by the delimiter symbols. First, the network is shown a sequence of items and then it is shown one of the items (from the sequence). The model is required to output the item that appears next from the ingested sequence. We set the length of each item to be 3. The levels are defined in terms of the number of items in the sequence. The first level considers sequences with 5 items and the number of items is increased in steps of 3 per level, going till 20 levels where there are 62 items per sequence. We report the bit-wise accuracy metric.

5.3.3 Sequential Stroke MNIST Task

The Sequential Stroke MNIST (SSMNIT) task was introduced by Gülçehre et al. (2017) with an emphasis on testing the long-term dependency modeling capabilities of RNNs. In this task, each MNIST digit image I is represented as a sequence of quadruples $\{dx_i, , dy_i, eos_i, eod_i\}_{i=1}^{T}$. Here, T is the number of pen strokes needed

to define the digit, (dx_i, dy_i) denotes the pen offset from the previous to the current stroke (can be 1, -1 or 0), eos_i is a binary-valued feature to denote end of stroke and eod_i is another binary-valued feature to denote end of the digit. The average number of strokes per digit is 40. Given a sequence of pen-stroke sequences, the task is to predict the sequence of digits corresponding to each pen-stroke sequences in the given order. This is an extremely challenging task as the model is first required to predict the digits based on the pen-stroke sequence, count the number of digits, and then generate the digits in the same order as the input after having processed the entire sequence of pen-strokes. The levels are defined in terms of the number of digits that make up the sequence. Given that this task is more challenging than the other two tasks, we use a sequence of length 1 (i.e. single digit sequences) for the first level and increase the sequence length in steps of 1. Just like before, we consider 20 levels and report the per-digit accuracy as the metric.

5.3.4 Benchmark

So far, we have considered the setup with three tasks and have defined multiple levels within each task. Alternatively, we could think of each task as a "task distribution" and each level (within the task) as a task (within a "task distribution"). From now on, we employ the *task-distribution / task* notation to keep the discussion consistent with the literature in lifelong learning where multiple tasks are considered. Thus we have 3 task distributions (Copy, Associative Recall, and SSMNIST) and multiple tasks (in increasing order of difficulty) per task distribution. To be closely aligned with the *true* lifelong learning setup, we train all the models in an online manner where the network sees a stream of training data. Further, none of the examples are seen more than once. A common setup in online learning is to train the model with one example at a time. Instead, we train the model using mini-batches of 10 examples at a time to exploit the computational benefits of using mini-batches. However, we ensure that every mini-batch is generated randomly and that none of the examples are repeated so that a separate validation or test dataset is not needed. For each task (within a task distribution), we report the *current task accuracy* as an indicator of the model's performance on the current task. If the running-average of the *current task accuracy*, averaged over the last k batches, is greater-than or equal-to $c\%$, the model is said to have learned the task and we can start training the model on the next task. If the model fails to learn the current task, we stop the training procedure and report the number of tasks completed. Every model is trained for m mini-batches before it is evaluated to check if it has learned the task. Since we consider models with different capacity, some models could learn the task faster thus experiencing fewer examples. This setup ensures that each model is trained on the same number of examples. This training procedure is repeated for all the task distributions. Benchmark parameters k, m,

and c can be set to any reasonable value, as long as they are kept constant for all tasks in a given task distribution. Specifically, we set $k = 100$, and $m = 10000$ for all the tasks, as well as $c = 80$ for Copy and $c = 75$ for Associative Recall and SSMNIST.

In the lifelong learning setting, it is very important for the model to retain knowledge from the previous tasks while generalizing to the new tasks. Hence, each time the model learns a task, we evaluate it on all the previous tasks (that it has been trained on so far) and report the model's performance (in terms of accuracy) for each of the previous task. Additionally, we also report the average of all these previous task accuracies and denote it as the *per-task-previous-accuracy*. When the model fails to learn a task and its training is stopped, we report both the individual *per-task-previous-accuracy* metrics and the average of these metrics, which is denoted as the *previous-task-accuracy*. While the *per-task-previous-accuracy* metric can be used as a crude approximation to quantify the effect of catastrophic forgetting, we highlight that the metric, on its own, is an insufficient metric. Consider a model which learns to solve just 1 task and terminates training after the 2^{nd} task. When evaluated for backward transfer, it would be evaluated only on the 1^{st} task. Now consider a model which just finished training on the 10^{th} task. When evaluated for backward transfer, it would be evaluated on the first 9 tasks. *per-task-previous-accuracy* metric favors models which stop training early and hence the series of *per-task-previous-accuracy* metrics is a more relevant measure.

Another interesting aspect of lifelong learning is the generalization to unseen tasks. Analogous to the *per-task-previous-accuracy* and *previous-task-accuracy*, we consider the *per-task-future-accuracy* and *future-task-accuracy*. There is no success criteria associated with this evaluation phase and the metrics are interpreted as a proxy of a model's ability to generalize to future tasks. In our benchmark, the tasks are closely related, which makes it reasonable to test generalization to new tasks. Note that the benchmark tasks can have levels beyond 20 as well. We limited our evaluation to 20 levels as none of the models could complete all the levels.

In the context of lifelong learning systems, the model needs to expand its capacity once it has saturated, to make sure it can keep learning from the incoming data. We simulate this scenario in our benchmark setting as follows. If the model fails to complete a given task, we use some capacity expansion technique and expand the original model into a larger model. Specifically, since we are considering RNNs, we expand the size of the hidden state matrix. The expanded model is then allowed to train on the current task for 20000 iterations. From there, the expanded model is evaluated (and trained on subsequent tasks) just like a regular model. If the expanded model fails on any task, the training is terminated. Note that this termination criterion is a part of our evaluation protocol. In practice, we can expand the model as many times as we want. In the ablation studies, we consider a case where the model is expanded twice.

5.3.5 Rationale for using curriculum style setup

For all three task distributions, it can be argued that as the sequence length increases, the tasks become more challenging, since the model needs to store/retrieve a much longer sequence. Hence, for each task distribution, we define a curriculum of tasks by controlling the length of the input sequences. We note that our experimental setup is different from a real-life setting in two ways: First, in real-life, we may not know beforehand which data point belongs to which data (or task) distribution. Second, in real life, we have no control over the difficulty or complexity of the incoming data points. For the benchmark, we assume perfect knowledge of which data points belong to which task and we assume full control over the data distribution. This trade-off has several advantages:

1. As the tasks are arranged in increasing order of difficulty, it becomes much easier to quantify the change in the model's performance as the evaluation data distribution becomes different from the training data distribution.
2. It enables us to extrapolate the capacity of the model with respect to the unseen tasks. If the model is unable to solve the n^{th} task, it is unlikely to solve any of the subsequent tasks as they are harder than the current task. Thus, we can use the number of tasks solved (while keeping other factors like optimizer fixed) as an ordinal indicator of the model's capacity.
3. As the data distribution becomes harder, the model is forced to use more and more of its capacity to learn the task.
4. In general, given n tasks, there are $n!$ ways of ordering the task and the model should be evaluated on all these combinations as the order of training tasks could affect the model's performance. Having the notion of the curriculum gives us a natural way to order the tasks.

To highlight the fact that curriculum-based training is not trivial, we show the performance of an LSTM in the SSMNIST task in figure 5.1. We can see that training on different tasks makes the model highly susceptible to over-fitting to any given task and less likely to generalize across tasks.

Capacity saturation can happen because of two reasons in our proposed benchmark:

1. The model is operating in a lifelong learning setting and when the model learns a new task, it also needs to spend some capacity to retain knowledge about the previous tasks.
2. As the sequence length increases, the new tasks require more capacity to be learned.

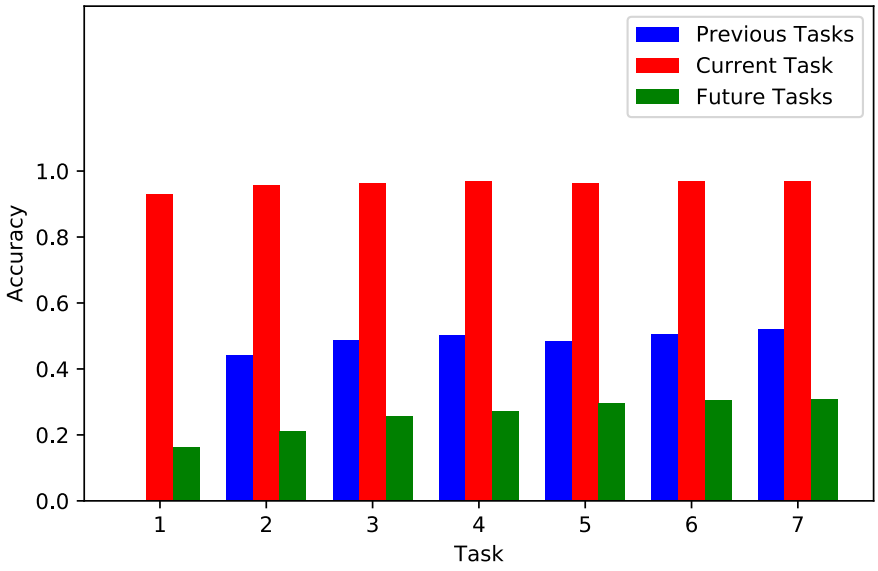


Figure 5.1 – Per-level accuracy on previous tasks, current task, and future tasks for a 128 dimensional LSTM trained in the SSMNIST task distribution by using the curriculum. The model heavily overfits to the sequence length.

Given these factors, it is expected that as the model learns new tasks, its capacity would be strained, thus necessitating solutions that enable the model to increase its capacity on the fly.

5.4 Model

In this section, we first describe how the rehearsal setup is used in the GEM model and how the function preserving transformations can be used in the *Net2Net* model. Next, we describe how we extend the *Net2Net* model for RNNs. Then, we describe how the proposed model leverages both these mechanisms in a unified lifelong learning framework.

5.4.1 Gradient Episodic Memory (GEM)

In this section, we provide a brief overview of the Gradient Episodic Memory (Lopez-Paz and Ranzato, 2017) and how it is used for alleviating catastrophic forgetting, while ensuring positive transfer on the previous tasks.

The basic idea is to store some input examples corresponding to each task (that the model has been trained on so far) in a memory buffer B. In practice, the buffer would have a fixed size, say B_{size}. If we know T, the number of tasks that the model would encounter, we could reserve B_{size}/T number of slots for each task. Alternatively, we could start with the first task, use all the slots for storing the examples from the first task. Then, as we progress through tasks, we keep reducing the number of memory slots per task. While selecting the examples to store in the buffer, we just save the last few examples from each task. Specifically, we store only 1 minibatch of examples (10 examples) per task and find that even this small amount of data is sufficient.

As the model is training on the l^{th} task, care is taken to ensure that the current gradient updates do not increase the loss on the examples already saved in the memory. This is achieved as follows. Given that the model is training on the l^{th} task, we first compute the parameter gradient with respect to the data for the current task, which we denote as the *current task gradient* or as g_l. Then a parameter gradient is computed corresponding to each of the previous tasks and is denoted as the *previous task gradient*. If the current gradient g_l increases the loss on any of the previous tasks, it is projected to the closest gradient $\tilde{g}_l$ (where closeness is measured in terms of L_2 norm) such that the condition is no more violated. Whether the *current task gradient* increases the loss on any of the previous tasks can be checked by computing the dot product between *current task gradient* and the *previous task gradient* (corresponding to the given previous task). The projected gradient update $\tilde{g}_l$ can be obtained by solving the following set of equations

$$\begin{aligned} \text{minimize}_{\tilde{g}_l} \quad & \frac{1}{2} \quad \|g_l - \tilde{g}_l\|_2^2 \\ \text{subject to} \quad & \langle \tilde{g}_l, g_k \rangle \geq 0 \text{ for all } k < l. \end{aligned} \tag{5.1}$$

To solve (5.1) efficiently, the authors use the primal of a Quadratic Program (QP) with inequality constraints:

$$\begin{aligned} \text{minimize}_z \quad & \frac{1}{2} z^\top C z + p^\top z \\ \text{subject to} \quad & Az \geq b, \end{aligned} \tag{5.2}$$

where $C \in \mathbb{R}^{p \times p}$, $p \in \mathbb{R}^p$, $A \in \mathbb{R}^{(t-1) \times p}$, and $b \in \mathbb{R}^{t-1}$. The dual problem of (5.2) is:

$$\begin{aligned} \text{minimize}_{u,v} \quad & \frac{1}{2} u^\top C u - b^\top v \\ \text{subject to} \quad & A^\top v - Cu = p, \\ & v \geq 0. \end{aligned} \tag{5.3}$$

If $(u^\star, v^\star)$ is a solution to (5.3), then there is a solution $z^\star$ to (5.2) satisfying

$Cz^\star = Cu^\star$ (Dorn, 1960).

The primal GEM QP (5.1) can be rewritten as:

$$\begin{aligned} \text{minimize}_z \quad & \frac{1}{2} z^\top z - g^\top z + \frac{1}{2} g^\top g \\ \text{subject to} \quad & Gz \geq 0, \end{aligned}$$

where $G = -(g_1, \ldots, g_{t-1})$, and the constant term $g^\top g$ is discarded. This new equation is a QP on p variables (where p is the number of parameters of the neural network). Since the network could have a lot of parameters, it is not feasible to solve this equation and the dual of the GEM QP is considered:

$$\begin{aligned} \text{minimize}_v \quad & \frac{1}{2} v^\top G G^\top v + g^\top G^\top v \\ \text{subject to} \quad & v \geq 0, \end{aligned} \tag{5.4}$$

since $u = G^\top v + g$ and the term $g^\top g$ is constant. This is a QP on $t-1 \ll p$ variables (where t is the number of observed tasks so far). Solution for the dual problem (5.4), $v^\star$, can be used to recover the projected gradient update as $\tilde{g} = G^\top v^\star + g$. The authors recommend adding a small constant $\gamma \geq 0$ to $v^\star$ as it helps to bias the gradient projection to updates that favoured beneficial backwards transfer.

We refer to this projection step as computing the *GEM gradient* and the resulting update as the *GEM update*. Since the projected gradient is only constrained to not increase the loss on the previous examples, a beneficial backward transfer is possible.

There are several downsides of using the GEM model. First, the projection of *current task gradient* regularizes the model, thereby decreasing its effective capacity. This effect can be seen in figure 5.2 where for all the three task distributions, the green curve (Large LSTM model which does not use the GEM update) consistently outperforms the red curve (LSTM model which uses the GEM update) both in terms of *current task accuracy* and in terms of numbers of tasks completed. We counter this limitation by using functional transformations to enable capacity expansion. Another downside is the cost - both in terms of computation and memory - of storing and rehearsing over the previous examples. We found that for all our experiments, storing just 10 examples per task is sufficient to get benefit from the GEM model. Hence the memory footprint of storing the training examples is very small and almost negligible as compared to the memory cost of persisting different copies of the model. The computational overhead of computing the GEM *gradient* could be reduced to some extent by controlling the frequency at which the model rehearses on the previous examples and future work could look at a more systematic approach to eliminate or reduce this computational cost.

5.4.2 Net2Net

Training a lifelong learning system on a continual stream of data can be seen as training a model with an infinite amount of data. As the model experiences more and more data points, the size of its effective training dataset increases and eventually the network would have to expand its capacity to continue training. *Net2Net* (Chen et al., 2015) proposed a very simple technique, based on function preserving transformations, to achieve zero-shot knowledge transfer when expanding a small, trained network (referred to as the teacher network) into a large, untrained network (referred to as the student network). Given a teacher network represented by the function $y = f(x, \theta)$ (where θ refers to the network parameters), a new set of parameters ϕ are chosen such that $\forall x, f(x, \phi) = g(x, \theta)$. The paper considered two variants of this approach - *Net2WiderNet* which increases the width of an existing network and *Net2DeeperNet* which increases the depth of the existing network. The main benefit of using function-preserving transformations is that the student network immediately performs as well as the original network without having to go through a period of low performance.

We use the *Net2WiderNet* for expanding the capacity of the model. The formulation of *Net2WiderNet* is as follows.

Assume that we start with a fully connected network where we want to widen layers i and $i + 1$. The weight matrix associated with layer i is $\boldsymbol{W}^{(i)} \in \mathbb{R}^{m \times n}$ and that associated with layer $i+1$ is $\boldsymbol{W}^{(i+1)} \in \mathbb{R}^{n \times p}$. Layer i may use any element-wise non-linearity. When we widen layer i, the weight matrix $\boldsymbol{W}^{(i)}$ expands into $\boldsymbol{U}^{(i)}$ to have q output units where $q > n$. Similarly, when we widen layer $i + 1$, the weight matrix $\boldsymbol{W}^{(i+1)}$ expands into $\boldsymbol{U}^{(i+1)}$ to have q input units.

A random mapping function $g : \{1, 2, \cdots, q\} \rightarrow \{1, 2, \cdots, n\}$, is defined as:

$$g(j) = \begin{cases} j & j \leq n \\ \text{random sample from } \{1, 2, \cdots n\} & j > n \end{cases}$$

For expanding $\boldsymbol{W}^{(i)}$, the columns of $\boldsymbol{U}^{(i)}$ are randomly chosen from $\boldsymbol{W}^{(i)}$ using g as shown:

$$\boldsymbol{U}^{(i)}_{k,j} = \boldsymbol{W}^{(i)}_{k,g(j)}$$

Notice that the first n columns of $\boldsymbol{W}^{(i)}$ are copied directly into $\boldsymbol{U}^{(i)}$.

The rows of $\boldsymbol{U}^{(i+1)}$ are randomly chosen from $\boldsymbol{W}^{(i+1)}$ using g as shown:

$$\boldsymbol{U}^{(i+1)}_{j,h} = \frac{1}{|\{x|g(x) = g(j)\}|} \boldsymbol{W}^{(i+1)}_{g(j),h}$$

Similar to the previous case, the first n rows of $\boldsymbol{W}^{(i+1)}$ are copied directly into $\boldsymbol{U}^{(i+1)}$.

The replication factor, (given by $\frac{1}{|\{x|g(x)=g(j)\}|}$), is introduced to make sure that the output of the two models is exactly the same. This procedure can be easily extended to multiple layers. Similarly, the procedure can be used for expanding convolutional networks (where layers will have more convolution channels) as convolution is multiplication by a doubly block circulant matrix).

Once the training network has been expanded, the newly created larger network can continue training on the incoming data. In theory, there is no restriction on how many times the *Net2Net* transformation is applied, though we limit to using the transformation only once for most of our experiments.

While Chen et al. (2015) mention lifelong learning as one of their motivations, they only focused on transfer learning from smaller network to a larger network for the single-task setup. Secondly, they considered the *Net2Net* transformation in the context of feed-forward and convolutional models. Our work is the first attempt to use *Net2Net* style function transformations for model expansion in the context of lifelong learning or even for sequential models.

5.4.3 Extending Net2Net for RNNs

In this section, we discuss the applicability of the *Net2Net* formulation for the RNNs in the context of lifelong learning.

The *Net2WiderNet* transformation makes two recommendations about the training of the student network. The first is that the learning rate for the student network may be reduced by an order of 10. This argument seems useful in the original setup in which *Net2Net* is proposed: training the student model over the same data on which the teacher model was trained. In the context of lifelong learning, the model does not see the same data again and the data distribution changes with the task. Hence the argument about lowering the learning rate does not apply. Our preliminary experiments showed that reducing the learning rate degrades the performance of the model. Hence we decided not to reduce the learning rate after the expansion.

The second and more important recommendation is that a small amount of random noise should be added to the student network to break the symmetry. In our initial experiments, we found that adding noise is a requirement and the model without noise performs extremely poorly. This is in contrast to the feed-forward setting where the model works quite well even without using noise.

In the case of RNNs, when we apply the *Net2WiderNet* transformation, the

condition number of the hidden-to-hidden matrices increases drastically and it becomes ill-conditioned. Recall that the condition number is defined as the ratio of the largest singular value of the matrix to its smallest singular value. The ideal condition number would be 1 (as is the case of orthogonal matrices) and ill-conditioned networks are harder to train. Without adding noise, the condition number becomes infinity after expansion. This is due to the presence of correlated rows in the matrix. One way to get around this problem is to add a small amount of noise which helps to precondition the weight matrices and hence reduce their condition number. The issue with adding random noise is that it breaks down the equality condition and hence comes with a trade-off - A higher amount of random noise reduces the condition number more (make it better conditioned) but pushes the output of the newly instantiated student network away from the predictions of the old teacher network.

To that end, we propose a simple extension to the noise addition procedure which ensures that the output of the student and the teacher networks remain the same while taking care of the preconditioning aspect. Let us say that we had the weight matrix $\mathbf{W}_{m \times n}$ which we expanded into $\mathbf{U}_{m \times p}$ using the *Net2WiderNet* transformation (where $p > n$). $\mathbf{U}$ would have some columns of $\mathbf{W}$ replicated. Let us say that the i^{th} column was replicated j times. Then, we would generate a noise matrix of small random values of size $m \times j$. The columns from this noise matrix would be added to columns that were replicated from i^{th} column of the input matrix $\mathbf{W}$. The noise matrix is generated such that for any row in the matrix, the sum of elements in that row of the noise matrix is 0. It can be shown mathematically that this transformation gives the exact same output as the case of no noise. We have to employ this procedure to make sure that the random noise we add sums up to 0. Since the given noise is random, it eliminates the correlation between rows and columns of the expanded weight matrix. Since the noise sums up to 0, it does ensure that the output of the student network is the same as that of the teacher network.

How do we generate a matrix of random values where the sum of values along each row is 0? We describe a technique to generate a vector of random numbers such that the values sum up to 1 and then we can use the technique multiple times to sample multiple rows to form the matrix. Let us say we want to generate a vector of random values of length k such that the values sum to 1. We first sample $k - 1$ random points in the range (0, 1). Note that all these $k - 1$ values will be smaller than 1 and larger than 0. We added the numbers 0 and 1 to this sequence and sort the sequence in the ascending order. This gives us a sorted sequence of $k + 1$ points where each point lies in the range [0, 1] with the *first value* being 0 and the *last value* being 1. We take pairwise difference of values between the adjacent points *i.e. (second value - first value), (third value - second value)* and so on. Summing up this sequence of values would give us *(last value - first value)* as

all the other terms would cancel out. Since the *first value* is 0 and the *last value* is 1, the sum of the sequence of resulting k points is 1. From this sequence of numbers, we can subtract $1/k$ to each of them and the resulting sequence would exactly sum up to 0. These steps are also described in Algorithm 2. Additionally, we scale the noise so that it is in the same range as the magnitude of the weights of the teacher network. Scaling the noise does not change the sum of the noise elements as both the positive and the negative elements get scaled by the same amount and still cancel each other. We use this strategy while using the expansion step.

Algorithm 2 Generating a random-valued vector of length k where the values sum to 0

1: **procedure** GENERATOR(k)
2: Sample $k-1$ random points in the range (0, 1).
3: Add values 0 and 1 to the sequence of sampled values.
4: Sort the sequence and create a new sequence by subtracting the pairwise values from the sorted sequence.
5: The resulting sequence of k values will sum to 1 (described in the text).
6: From each of the values, subtract $1/k$ to ensure that the resulting sequence of random values sums up to 0.

5.4.4 Unified Model

We now describe how we combine the catastrophic forgetting solution (GEM) and the capacity expansion solution (functional transformations) to come up with a more suitable model for lifelong learning. Given a task distribution, we randomly initialize a model, reset the episodic memory to be empty and start training the model on the first task (simplest task). Once a task is learned, the model starts training on the subsequent, more difficult tasks. When we are training the model on the l^{th} task, the episodic memory already has some examples corresponding to the first $l-1$ tasks. The *current task gradient* is projected with respect to the *previous task gradient*s to ensure that it does not increase the loss associated with any of the examples in the episodic memory. The projected *GEM Gradient* is used to update the weights of the model (*GEM Update*). The model is trained on the current task for a fixed number of iterations. The last m training examples from the current task are stored in the episodic memory for use in the subsequent tasks. In general, the m examples can be selected with some more sophisticated strategy though Lopez-Paz and Ranzato (2017) reports, and we validate, that using just the last m samples works well in practice.

If the model completes learning the current task (i.e. achieves a threshold amount of accuracy after training), the model can start training on the next task. If the model fails to learn the current task, and has not been expanded so far, the model is expanded to a larger model and is allowed to train further on the current task. Once the expanded model is trained, it is re-evaluated to check if it has learned the task. If it has, the model progresses to the next task, otherwise, the training procedure is terminated. Irrespective of how much is the *current task accuracy*, the model is evaluated on all the tasks - to measure its *previous task accuracy* and *future task accuracy*.

5.4.5 Analysis of the computational and memory cost of the proposed model

As noted in section 5.1, an important desideratum in lifelong learning models is that the computational and the memory costs of the model should ideally grow sublinearly as the model is trained on new tasks. In the context of our proposed model, the computational and memory costs can change in the following ways:

1. The *Net2Net* component expands the model. In this case, the expanded model would take more resources (both in terms of parameters and time) than the earlier model. We note that the expansion step does not happen for every new task and is performed only when the model's capacity saturates. This is in contrast to approaches like Rusu et al. (2016) where a new copy of the network is added every time a new task is introduced thus increasing both the parameters and the compute time linearly with the number of tasks. In our case, the frequency of expansion is sublinear in the number of tasks.
2. The GEM model stores some examples (in a buffer) from the previous tasks and performs gradient computation with respect to those examples, along with the gradient computation for the current examples. As noted in section 5.4.1, we could keep the buffer size to be fixed and replace some examples from the previous tasks as new examples are observed while making sure that all the tasks are represented through examples in the buffer. In practice, we found that storing a few examples per task is sufficient to get benefit from the GEM model (as also observed by Lopez-Paz and Ranzato (2017)), making the memory footprint negligible. As noted earlier, future work could look at some systematic ways of selecting the examples from the buffer thus reducing the computational overhead.

One beneficial side effect of using *Net2Net* expansion is the zero-shot knowledge transfer that further amortizes the cost of training a newly initialized larger model

- either from a smaller, pre-trained model or from dataset corresponding to the tasks encountered so far.

5.5 Experiments

5.5.1 Models

For each task distribution, we consider a standard recurrent (LSTM) model operating in the lifelong learning setting. We consider the different aspects of training a lifelong learning system and describe how the model variants can account for these aspects. We start with an LSTM model with hidden state size of 128 and refer to this model as the *small-Lstm* model. This model has sufficient capacity to learn the first few tasks. We start training the (*small-Lstm*) model as described in section 5.3.4. To avoid catastrophic forgetting, we could additionally use the *GEM update* when training the model. The resulting model is referred to as the *small-Lstm-Gem* model. After learning some tasks, the model would have used up all its capacity (since it is retaining the knowledge of the previous tasks as well). In this case, we could expand the model's capacity using the *Net2Net* transformation and the model with this capability is referred to as the *small-Lstm-Gem-Net2Net* model. This is the model we propose. Alternatively, we could have started the training with a larger model (*large-Lstm* model) and could have used the GEM update (*large-Lstm-Gem* model) to counter forgetting. The strategy of always starting training with a large network would not work in practice because in the lifelong learning setting we do not know what network would be sufficiently large to learn all the tasks beforehand. If we start with a very large model, we would need a lot more computational resources to train the model and the model would be very prone to over-fitting. Our proposed model (*small-Lstm-Gem-Net2Net*) gets around this problem by increasing the capacity on the fly as and when needed. For the *large-Lstm* model family, we set the size of the hidden layer to be 256. Our empirical analysis shows that it is possible to expand the models to a size much larger than their current size without interfering with the *GEM update*.

For the performance on the current task, *large-Lstm* model can be treated as the gold standard since this model has the largest capacity among all the models considered. Unlike the models which use the *GEM Update*, this model does not have to "use" some of its capacity for retaining the knowledge of the previous tasks. For the performance on the previous tasks (catastrophic forgetting), we consider the *large-Lstm-Gem* model as the gold standard as this model has the largest capacity among all the models and is specifically designed to counter catastrophic forgetting.

While we do not have a gold standard for the *Future Task Accuracy*, both *large-Lstm-Gem* and *large-Lstm* are reasonable models to compare with. Overall, we have three different gold standards for three setups (and metrics) and we compare our proposed model to these different gold standards (each specialized for a specific use-case).

5.5.2 Hyper Parameters

All the models are implemented using PyTorch 0.4.1 (Paszke et al., 2017). Adam optimizer (Kingma and Ba, 2014) is used with a learning rate of 0.001. We used one layer LSTM models with hidden dimensions of size 128 and 256. *Net2Net* is used to expand LSTM models of size 128 to 256. For the GEM model, we keep one minibatch (10 examples) of data per task for obtaining the projected gradients. We follow the guidelines and hyperparameter configurations as specified in the respective papers for both *GEM* and *Net2Net* models.

5.5.3 Results

Figure 5.2 shows the trend of the *current task accuracy* for the different models on the three task distributions. In these plots, a higher curve corresponds to the model that has higher accuracy on the current task and models which learn more tasks are spread out more along the x-axis. We compare the performance of the proposed model *small-Lstm-Gem-Net2Net* with the gold standard *large-Lstm* model. We additionally compare with *large-Lstm-Gem* model as both this model and the proposed model are constrained to use some of their capacity on the previous tasks. Hence it provides a more realistic estimate of the strength of the proposed model. It also allows us to study the effect of the *GEM Update* on the model's effective capacity (in terms of the number of tasks cleared). The blue dotted line corresponds to the expansion step when the model is not able to learn the current task and had to expand. This shows that using the capacity expansion technique from *Net2Net* enables learning on newer tasks. We highlight that before expansion, the proposed model *small-Lstm-Gem-Net2Net* had a much smaller capacity (128 hidden dims) as compared to the other two models which started with a much larger capacity (256 hidden dims). This explains why the larger models have much better performance in the initial stages. Post expansion, the proposed model overtakes the GEM based model in all the cases (in terms of the number of tasks solved). We can observe that in all the cases, the *large-Lstm* model outperforms the *large-Lstm-Gem* model, which suggests that using the *Gem Update* comes at the cost of reducing the capacity for the current task. Using capacity expansion techniques with *GEM* enables the model to account for this

loss of capacity.

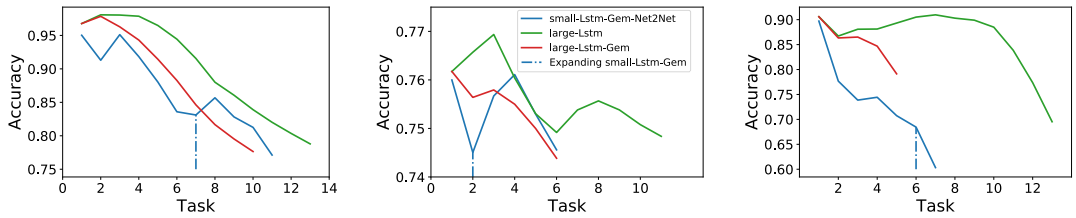


Figure 5.2 – *Current Task Accuracy* for the different models on the three "task distributions" (Copy, Associative Recall, and SSMNIST respectively). On the x-axis, we plot the index of the task on which the model is training currently and on the y-axis, we plot the accuracy of the model on that task. Higher curves have higher *current task accuracy* and curves extending more have completed more tasks. For all the three "task distributions", our proposed *small-Lstm-Gem-Net2Net* model clears either more levels or same number of levels as the *large-Lstm-Gem* model. Before the blue dotted line, the proposed model is of much smaller capacity (hidden size of 128) as compare to other two models which have a larger hidden size (256). Hence the larger models have better accuracy initially. Capacity expansion technique allows our proposed model to clear more tasks than it would have cleared otherwise.

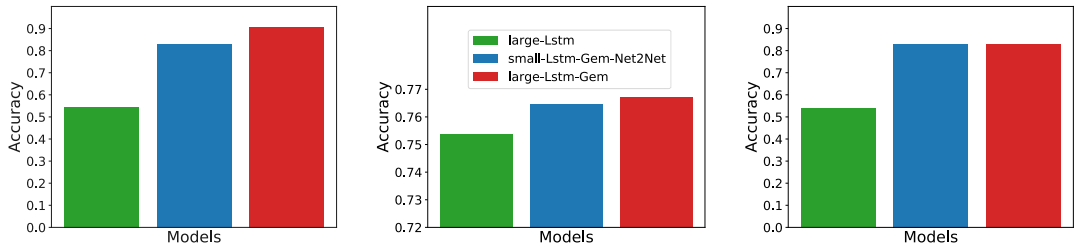


Figure 5.3 – *Previous Task Accuracy* for the different models on the three task distributions (Copy, Associative Recall, and SSMNIST respectively). Different bars represent different models and on the y-axis, we plot the average previous task accuracy (averaged for all the tasks that the model learned). Higher bars have better accuracy on the previously seen tasks and are more robust to catastrophic forgetting. For all the three task distributions, the proposed models are very close in performance to the *large-Lstm-Gem* models and much better than the *large-Lstm* models.

Figure 5.3 shows the trend of the *previous task accuracy* for the different models. A higher bar corresponds to better accuracy on the previous tasks (more resilience to catastrophic forgetting). We compare the performance of the proposed model *small-Lstm-Gem-Net2Net* with the gold standard model *large-Lstm-Gem*. We additionally compare with the *large-Lstm* model to demonstrate that *GEM Update* is essential to have a good performance on the previous tasks. The most important observation is the relative performance of the proposed *small-Lstm-Gem-Net2Net* model and the *large-Lstm-Gem* model. The *small-Lstm-Gem-Net2Net* model started as a smaller model, consistently learned more tasks than

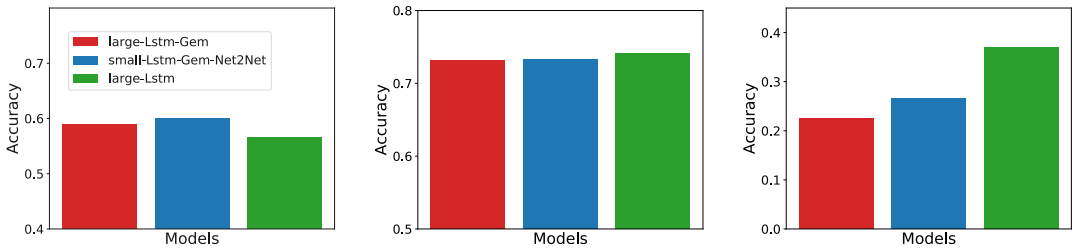
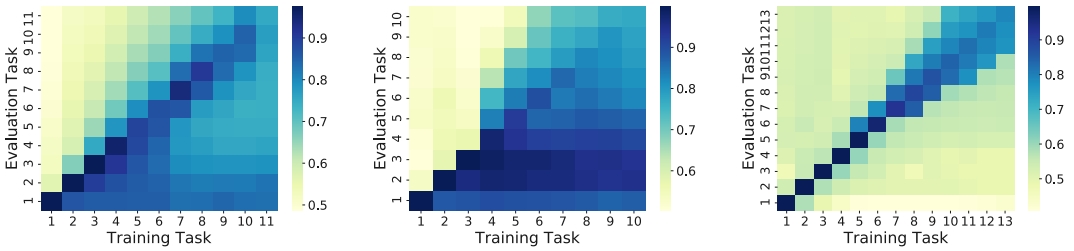


Figure 5.4 – *Future Task Accuracy* for the different models on the three task distributions (Copy, Associative Recall, and SSMNIST respectively). Different bars represent different models and on the y-axis, we plot the average future task accuracy (averaged for all the tasks that the model learned). Higher bars have better accuracy on the previously unseen tasks and are more beneficial for achieving knowledge transfer to future tasks. Even though the proposed model does not have any component for specifically generalizing to the future tasks, we expect the proposed model to generalize at least as well as the *large-Lstm-Gem* model and comparable to *large-Lstm*. Interestingly, our model outperforms the *large-Lstm* model for Copy task and is always better than (or as good as) the *large-Lstm-Gem* model.

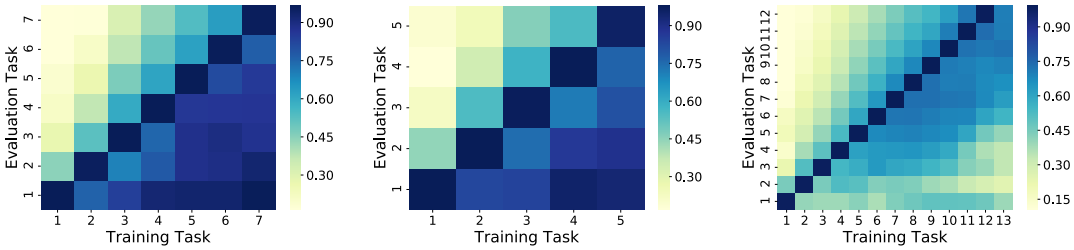
large-Lstm-Gem model and is still almost as good as *large-Lstm-Gem* model in terms of *Previous Task Accuracy*. This shows that the proposed model is very robust to catastrophic forgetting while being very good at learning the current task. We also observe that for all the three "task distributions", the models using the *GEM update* are more resilient to catastrophic forgetting as compared to the models without the *GEM Update*.

Figure 5.4 shows the trend of the *future task accuracy* for different models. A higher bar corresponds to better accuracy on the future (unseen) tasks. Since we do not have any gold standard for this setup, we consider both *large-Lstm-Gem* and *large-Lstm* models as they both are reasonable models to compare with. The general trend is that our proposed model is quite close to the reference models for 2 out of 3 tasks. Note that both the larger models started training with a much larger capacity and further, the *large-Lstm* model is not constrained by the *GEM Update* and hence the maximum amount of effective capacity. This could be one reason why the model can outperform our proposed model for one of the tasks.

We also consider the heatmap plots where we plot the accuracy of different models (for different "task distributions") as they are trained and evaluated on different tasks. As pointed out in section 5.3.4, the aggregated metrics (*current task accuracy*, *previous task accuracy*, etc) are not sufficient to compare the performance of different models and fine-grained analysis is useful for having a holistic view. We observe that for the *large-Lstm* model, the large values are concentrated along the diagonal while for the *small-Lstm-Gem-Net2Net* and the *large-Lstm-Gem* models, the high values are concentrated in the lower diagonal region indicating that the two models are quite resilient to catastrophic forgetting. Additionally, note that while



(a) Copy Task



(b) SSMNIST Task

Figure 5.5 – Accuracy of the different models (*small-Lstm-Gem-Net2Net*, *large-Lstm-Gem* and *large-Lstm* respectively) as they are trained and evaluated on different tasks for the Copy and the SSMNIST task distributions. On the x-axis, we show the task on which the model is trained and on the y-axis, we show the accuracy corresponding to the different tasks on which the model is evaluated. We observe that for the *large-Lstm* model, the high accuracy values are concentrated along the diagonal which indicates that the model does not perform well on the previous task. In the case of both *small-Lstm-Gem-Net2Net* and *large-Lstm-Gem* models, the high values are in the lower diagonal region indicating that the two models are quite resilient to catastrophic forgetting.

the *large-Lstm-Gem* model appears to be more resilient to catastrophic forgetting, the *small-Lstm-Gem-Net2Net* model consistently clears more tasks. Note that even though we are evaluating the models for all the tasks in the benchmark, we are restricting the heatmap to only show evaluation results for the highest task index that the model could solve. This results in square-shaped heatmaps which are easier to analyze.

It is important to note that we are using a single proposed model (*small-Lstm-Gem-Net2Net*) and comparing it with gold-standard models in 3 different contexts - performance on the current task, performance on the backward tasks and performance on the future tasks. Our model can provide strong performance on all three tasks by countering catastrophic forgetting and by using capacity expansion.

5.6 Conclusion

In this chapter, we study the problem of capacity saturation and catastrophic forgetting in lifelong learning in the context of sequential supervised learning. We propose to unify Gradient Episodic Memory (a catastrophic forgetting alleviation approach) and Net2Net (a capacity expansion approach) to develop a model that is more suitable for lifelong learning. We also propose a curriculum based evaluation benchmark where the models are trained on a task with increasing levels of difficulty. This enables us to sidestep some of the challenges that arise when studying lifelong learning. We conduct experiments on the proposed benchmark tasks and show that the proposed model is better suited for the lifelong learning setting as compared to the two individual models. As future work, we would want to address the computational overhead associated with the *GEM Update* step. One potential solution called *Averaged GEM* has been recently proposed by Chaudhry et al. (2019).

6 Discussions and Future Work

In this final chapter, we summarize the contributions of this thesis and discuss some possible future research directions based on this thesis's work.

6.1 Summary of Contributions

This thesis focused on the challenges in training Recurrent Neural Networks (RNNs). Specifically, we attempted to tackle the following three fundamental challenges that arise when training RNNs in single-task and multi-task settings:

— **Vanishing gradients:** When training RNNs with very long sequences, gradients vanish for two reasons: recurrent matrix multiplication (due to the linear chain dependencies on previous hidden states) and saturating activation functions. In Chapter 3, we proposed a recurrent architecture called TARDIS which is basically an LSTM with dynamic skip connections (*aka* wormhole connections) to the previous hidden states. These dynamic skip connections create tree-structured dependencies to previous hidden states rather than linear chain dependencies created in a vanilla LSTM. This helps in better gradient flow in very long sequences. Chapter 4 proposes another recurrent architecture called Non-saturating Recurrent Unit (NRU). NRUs replace recurrent multiplicative updates with recurrent additive updates and does not have any saturating activation function. Thus NRUs have significantly better gradient flow when compared to other state-of-the-art recurrent architectures.

— **Capacity saturation and Catastrophic forgetting:** When training RNNs in the lifelong learning setting, where it would see a series of tasks, the performance on the previous tasks might degrade while learning the current task. This is known as *catastrophic forgetting*. Catastrophic forgetting could happen for two reasons: (i) while trying to find the optimal parameter configuration for the current task, the model could move away from the optimal parameter configuration for the previous tasks, (ii) Since we are using a finite capacity parametric model, the network would unlearn some previous task to learn the current task if there is not enough capacity to learn

more tasks. This second issue is known as *Capacity saturation*. While much of the existing work studies catastrophic forgetting in isolation, Chapter 5 stresses that we should study both capacity saturation and catastrophic forgetting together. We specifically propose a new way to train RNNs which incorporates solutions for solving both capacity saturation (Net2Net) and catastrophic forgetting (GEM).

6.2 Future Directions

6.2.1 Better Recurrent Architectures

The search for a fully expressive recurrent neural network with no vanishing and exploding problem is not yet over. This thesis proposes few ways to improve the current recurrent architectures and more research is needed to design an ideal recurrent neural network. The work proposed in this thesis sets up the groundwork by introducing architectural changes to improve RNNs and should be followed up by investigating how to improve the learning algorithm itself.

6.2.2 Exploding gradients

NRUs mitigate the problem of vanishing gradients by using non-saturating activations but they can still suffer from exploding gradients. Unlike vanishing gradients, exploding gradients can be controlled by gradient clipping. Our initial exploration of gradient clipping with NRUs shows that it is not a principled solution to have a stable training dynamics and more research is needed to devise better ways of handling exploding gradients. While most of the literature is focused on vanishing gradients, we hypothesise that it is easier to work in the exploding gradients regime than the vanishing gradients regime. We conjecture that once the gradient vanishes, there is no hope in learning those long-term dependencies. NRUs and ReLU RNNs could be good starting points for research on exploding gradients.

6.2.3 Do we really need recurrent architectures?

Recently introduced Transformer architecture (Vaswani et al., 2017) is shown to be superior to the recurrent architectures on several NLP tasks (Liu et al., 2019a,b). Transformer is essentially a feed-forward neural network with self-attention mechanism. This raises the critical question: do we really need *recurrent* architectures?

In one of our recent work (Sankar et al., 2019), we show that we need both attention and recurrence to model sequential problems. One potential research direction is to combine the benefits on Transformers and RNNs by designing a hybrid sequence modeling architecture. Some of the recent efforts in this direction includes Transformer-XL (Dai et al., 2019) and Universal Transformers (Dehghani et al., 2019). Another important research initiative would be to design very deep RNNs to compete with the benefits that Transformers gain through their depth. While training very deep LSTMs would have vanishing gradient issues, deep NRUs should reduce the vanishing gradient issue and would be the ideal architecture for this task.

6.2.4 Recurrent Neural Networks for Reinforcement Learning

Learning long-term credit assignment is one of the fundamental problems in Reinforcement Learning (RL) (Sutton and Barto, 1998). RNNs are primarily used to handle the partial observability in RL (Heess et al., 2015) though RNNs can be seen as the architectural solution for handling long-term credit assignment. However, dependencies in a typical RL task could be in the order of thousands of time steps. Hence we need recurrent architectures which can model really long-term dependencies. NRUs can be considered as the first step towards such powerful recurrent architectures. Our initial experiments show that NRUs can model dependencies of length of order 2000 in the copy task. One immediate step would be to apply NRUs on standard RL tasks with very long dependencies.

6.2.5 Reinforcement Learning for Recurrent Neural Networks

While the previous subsection suggested using RNNs to model long-term credit assignment problem in RL, we can also transfer the knowledge from the RL literature on the long-term credit assignment problem to learning long-term dependencies in RNNs. One potential research direction is to look at eligibility traces (Sutton and Barto, 1998) in RL and derive equivalent algorithms for training RNNs. In chapter 3, we used REINFORCE to compute approximate gradients for non-differentiable components of TARDIS architecture. A possible extension is to design better ways of computing approximate gradients using RL.

6.3 Conclusion

In this thesis, we discussed some of the fundamental challenges in training Recurrent Neural Networks. This includes the problems of vanishing gradients, catastrophic forgetting, and capacity saturation. The thesis has made some progress in each of these problems by proposing new architectures and training algorithms. As we highlighted in the introduction, any improvement to training RNNs should have a significant impact in several real-world sequential problems. We strongly believe that the proposed architectures and algorithms would show such impact.

Bibliography

R. Aljundi, P. Chakravarty, and T. Tuytelaars. Expert Gate: Lifelong Learning with a Network of Experts. *ArXiv e-prints*, November 2016.

Martín Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 1120–1128, 2016.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio. End-to-end attention-based large vocabulary speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2016, Shanghai, China, March 20-25, 2016*, pages 4945–4949, 2016. doi: 10.1109/ICASSP.2016.7472618. URL https://doi.org/10.1109/ICASSP.2016.7472618.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166, 1994.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Samuel R Bowman, Gabor Angeli, Christopher Potts, and Christopher D Manning. A large annotated corpus for learning natural language inference. *arXiv preprint arXiv:1508.05326*, 2015.

Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R Hruschka Jr, and Tom M Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, volume 5, page 3. Atlanta, 2010.

Sarath Chandar, Chinnadhurai Sankar, Eugene Vorontsov, Samira Ebrahimi Kahou, and Yoshua Bengio. Towards non-saturating recurrent units for modelling long-term dependencies. In *The Thirty-Third AAAI Conference on Artificial*

Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019., pages 3280–3287, 2019. doi: 10.1609/aaai.v33i01.33013280. URL https://doi.org/10.1609/aaai.v33i01.33013280.

A. Chaudhry, P. K. Dokania, T. Ajanthan, and P. H. S. Torr. Riemannian Walk for Incremental Learning: Understanding Forgetting and Intransigence. *arXiv e-prints*, January 2018.

Arslan Chaudhry, Marc'Aurelio Ranzato, Marcus Rohrbach, and Mohamed Elhoseiny. Efficient lifelong learning with A-GEM. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019. URL https://openreview.net/forum?id=Hkf2_sC5FX.

Qian Chen, Xiaodan Zhu, Zhenhua Ling, Si Wei, and Hui Jiang. Enhancing and combining sequential and tree lstm for natural language inference. *arXiv preprint arXiv:1609.06038*, 2016.

Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.

Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733*, 2016.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1724–1734, 2014. URL http://aclweb.org/anthology/D/D14/D14-1179.pdf.

Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014. URL http://arxiv.org/abs/1412.3555.

Junyoung Chung, Sungjin Ahn, and Yoshua Bengio. Hierarchical multiscale recurrent neural networks. *arXiv preprint arXiv:1609.01704*, 2016.

Tim Cooijmans, Nicolas Ballas, César Laurent, and Aaron C. Courville. Recurrent batch normalization. *CoRR*, abs/1603.09025, 2016. URL http://arxiv.org/abs/1603.09025.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 2978–2988, 2019. URL https://www.aclweb.org/anthology/P19-1285/.

Edwin D. de Jong. Incremental sequence learning. *arXiv preprint arXiv:1611.03068*, 2016.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019. URL https://openreview.net/forum?id=HyzdRiR9Y7.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. Ieee, 2009.

William S Dorn. Duality in quadratic programming. *Quarterly of Applied Mathematics*, 18(2):155–162, 1960.

T. Furlanello, Z. C. Lipton, M. Tschannen, L. Itti, and A. Anandkumar. Born Again Neural Networks. *ArXiv e-prints*, May 2018.

Felix A. Gers, Jürgen Schmidhuber, and Fred A. Cummins. Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 2000. doi: 10.1162/089976600300015015. URL https://doi.org/10.1162/089976600300015015.

Felix A. Gers, Nicol N. Schraudolph, and Jürgen Schmidhuber. Learning precise timing with LSTM recurrent networks. *J. Mach. Learn. Res.*, 3:115–143, 2002. URL http://jmlr.org/papers/v3/gers02a.html.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio G. Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià P. Badia, Karl M. Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, advance online publication, October 2016. ISSN 0028-0836. doi: 10.1038/nature20101. URL http://dx.doi.org/10.1038/nature20101.

Caglar Gulcehre, Sarath Chandar, Kyunghyun Cho, and Yoshua Bengio. Dynamic neural turing machine with soft and hard addressing schemes. *arXiv preprint arXiv:1607.00036*, 2016.

Çaglar Gülçehre, Sarath Chandar, and Yoshua Bengio. Memory augmented neural networks with wormhole connections. *CoRR*, abs/1701.08718, 2017. URL http://arxiv.org/abs/1701.08718.

Steven Gutstein, Olac Fuentes, and Eric Freudenthal. Knowledge transfer in deep convolutional neural nets. *International Journal on Artificial Intelligence Tools*, 17(03):555–567, 2008.

David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI Fall Symposia, Arlington, Virginia, USA, November 12-14, 2015*, pages 29–37, 2015. URL http://www.aaai.org/ocs/index.php/FSS/FSS15/paper/view/11673.

Nicolas Heess, Jonathan J. Hunt, Timothy P. Lillicrap, and David Silver. Memory-based control with recurrent neural networks. *CoRR*, abs/1512.04455, 2015. URL http://arxiv.org/abs/1512.04455.

Mikael Henaff, Arthur Szlam, and Yann LeCun. Recurrent orthogonal networks and long-memory tasks. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016*, 2016.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, page 91, 1991.

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

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

Li Jing, Çaglar Gülçehre, John Peurifoy, Yichen Shen, Max Tegmark, Marin Soljacic, and Yoshua Bengio. Gated orthogonal recurrent units: On learning to forget. *CoRR*, abs/1706.02761, 2017a.

Li Jing, Yichen Shen, Tena Dubcek, John Peurifoy, Scott A. Skirlo, Yann LeCun, Max Tegmark, and Marin Soljacic. Tunable efficient unitary neural networks (EUNN) and their application to rnns. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 1733–1741, 2017b.

Rafal Józefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 2342–2350, 2015. URL http://proceedings.mlr.press/v37/jozefowicz15.html.

D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *ArXiv e-prints*, December 2014.

J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell. Overcoming catastrophic forgetting in neural networks. *ArXiv e-prints*, December 2016.

Jan Koutnik, Klaus Greff, Faustino Gomez, and Juergen Schmidhuber. A clockwork rnn. *arXiv preprint arXiv:1402.3511*, 2014.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, pages 1106–1114, 2012.

David Krueger, Tegan Maharaj, János Kramár, Mohammad Pezeshki, Nicolas Ballas, Nan Rosemary Ke, Anirudh Goyal, Yoshua Bengio, Hugo Larochelle, Aaron Courville, et al. Zoneout: Regularizing rnns by randomly preserving hidden activations. *arXiv preprint arXiv:1606.01305*, 2016.

César Laurent, Gabriel Pereyra, Philemon Brakel, Ying Zhang, and Yoshua Bengio. Batch normalized recurrent neural networks. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2016, Shanghai, China, March 20-25, 2016*, pages 2657–2661, 2016. doi: 10.1109/ICASSP.2016.7472159. URL https://doi.org/10.1109/ICASSP.2016.7472159.

Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. A simple way to initialize recurrent networks of rectified linear units. *CoRR*, abs/1504.00941, 2015.

Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nature*, 521 (7553):436–444, 2015. doi: 10.1038/nature14539. URL https://doi.org/10.1038/nature14539.

Sang-Woo Lee, Jin-Hwa Kim, Jaehyun Jun, Jung-Woo Ha, and Byoung-Tak Zhang. Overcoming catastrophic forgetting by incremental moment matching. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4652–4662. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/7051-overcoming-catastrophic-forgetting-by-incremental-moment-matching.pdf.

Zhizhong Li and Derek Hoiem. Learning without forgetting. In *European Conference on Computer Vision*, pages 614–629. Springer, 2016.

Xialei Liu, Marc Masana, Luis Herranz, Joost Van de Weijer, Antonio M Lopez, and Andrew D Bagdanov. Rotate your networks: Better weight consolidation and less catastrophic forgetting. *arXiv preprint arXiv:1802.02950*, 2018.

Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. Multi-task deep neural networks for natural language understanding. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 4487–4496, 2019a. URL https://www.aclweb.org/anthology/P19-1441/.

Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. Improving multi-task deep neural networks via knowledge distillation for natural language understanding. *CoRR*, abs/1904.09482, 2019b. URL http://arxiv.org/abs/1904.09482.

Vincenzo Lomonaco and Davide Maltoni. Core50: a new dataset and benchmark for continuous object recognition. In Sergey Levine, Vincent Vanhoucke, and Ken Goldberg, editors, *Proceedings of the 1st Annual Conference on Robot Learning*, volume 78 of *Proceedings of Machine Learning Research*, pages 17–26. PMLR, 13–15 Nov 2017. URL http://proceedings.mlr.press/v78/lomonaco17a.html.

David Lopez-Paz and Marc Aurelio Ranzato. Gradient episodic memory for continual learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 6467–6476. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/7225-gradient-episodic-memory-for-continual-learning.pdf.

Sergey Loyka. On singular value inequalities for the sum of two matrices. *arXiv preprint arXiv:1507.06630*, 2015.

Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

Arun Mallya, Dillon Davis, and Svetlana Lazebnik. Piggyback: Adapting a single network to multiple tasks by learning to mask weights. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 67–82, 2018.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330, 1993.

Michael McCloskey and Neal J Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier, 1989.

Thomas Mensink, Jakob Verbeek, Florent Perronnin, and Gabriela Csurka. Metric learning for large scale image classification: Generalizing to new classes at near-zero cost. In *Computer Vision–ECCV 2012*, pages 488–501. Springer, 2012.

Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing LSTM language models. *CoRR*, abs/1708.02182, 2017. URL http://arxiv.org/abs/1708.02182.

Tomáš Mikolov, Ilya Sutskever, Anoop Deoras, Hai-Son Le, Stefan Kombrink, and J Cernocky. Subword language modeling with neural networks. *preprint (http://www. fit. vutbr. cz/imikolov/rnnlm/char. pdf)*, 2012.

Andriy Mnih and Karol Gregor. Neural variational inference and learning in belief networks. *arXiv preprint arXiv:1402.0030*, 2014.

Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. Recurrent models of visual attention. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 2204–2212, 2014. URL http://papers.nips.cc/paper/5542-recurrent-models-of-visual-attention.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 1928–1937, 2016.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

Junier B. Oliva, Barnabás Póczos, and Jeff G. Schneider. The statistical recurrent unit. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 2671–2680, 2017.

Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*, 2013a.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *ICML (3)*, 28:1310–1318, 2013b.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

Jack W. Rae, Jonathan J. Hunt, Tim Harley, Ivo Danihelka, Andrew W. Senior, Greg Wayne, Alex Graves, and Timothy P. Lillicrap. Scaling memory-augmented neural networks with sparse reads and writes. *CoRR*, abs/1610.09027, 2016.

Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. icarl: Incremental classifier and representation learning. In *Proc. CVPR*, 2017.

Mark B Ring. Child: A first step towards continual learning. *Machine Learning*, 28(1):77–104, 1997.

Tim Rocktäschel, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, and Phil Blunsom. Reasoning about entailment with neural attention. *arXiv preprint arXiv:1509.06664*, 2015.

Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

Chinnadhurai Sankar, Sandeep Subramanian, Chris Pal, Sarath Chandar, and Yoshua Bengio. Do neural dialog systems use the conversation history effectively? an empirical study. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 32–37, 2019. URL https://www.aclweb.org/anthology/P19-1004/.

Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. One-shot learning with memory-augmented neural networks. *arXiv preprint arXiv:1605.06065*, 2016.

Stanislau Semeniuta, Aliaksei Severyn, and Erhardt Barth. Recurrent dropout without memory loss. *arXiv preprint arXiv:1603.05118*, 2016.

Iulian Vlad Serban, Alessandro Sordoni, Ryan Lowe, Laurent Charlin, Joelle Pineau, Aaron C. Courville, and Yoshua Bengio. A hierarchical latent variable encoder-decoder model for generating dialogues. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 3295–3301, 2017. URL http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14567.

J. Serrà, D. Surís, M. Miron, and A. Karatzoglou. Overcoming catastrophic forgetting with hard attention to the task. *arXiv e-prints*, January 2018.

Daniel L Silver and Robert E Mercer. The task rehearsal method of life-long learning: Overcoming impoverished data. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pages 90–101. Springer, 2002.

Daniel L Silver, Qiang Yang, and Lianghao Li. Lifelong machine learning systems: Beyond learning algorithms. In *AAAI Spring Symposium: Lifelong Machine Learning*, volume 13, page 05, 2013.

Shagun Sodhani, Sarath Chandar, and Yoshua Bengio. On training recurrent neural networks for lifelong learning. *Neural Computation*, 2019.

Ray J Solomonoff. A system for incremental learning based on algorithmic probability. In *Proceedings of the Sixth Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition*, pages 515–527, 1989.

Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. *arXiv preprint arXiv:1503.08895*, 2015.

Ilya Sutskever, James Martens, and Geoffrey E Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1017–1024, 2011.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112, 2014.

Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981.

Corentin Tallec and Yann Ollivier. Can recurrent neural networks warp time? In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018. URL https://openreview.net/forum?id=SJcKhk-Ab.

Sebastian Thrun. Explanation-based neural network learning. In *Explanation-Based Neural Network Learning*, pages 19–48. Springer, 1996.

Sebastian Thrun. Lifelong learning algorithms. In *Learning to learn*, pages 181–209. Springer, 1998.

Sebastian Thrun. *Explanation-based neural network learning: A lifelong learning approach*, volume 357. Springer Science & Business Media, 2012.

Endel Tulving. Chronesthesia: Conscious awareness of subjective time. 2002.

Jos van der Westhuizen and Joan Lasenby. The unreasonable effectiveness of the forget gate. *CoRR*, abs/1804.04849, 2018. URL http://arxiv.org/abs/1804.04849.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 5998–6008, 2017. URL http://papers.nips.cc/paper/7181-attention-is-all-you-need.

Oriol Vinyals and Quoc V. Le. A neural conversational model. *CoRR*, abs/1506.05869, 2015. URL http://arxiv.org/abs/1506.05869.

Eugene Vorontsov, Chiheb Trabelsi, Samuel Kadoury, and Chris Pal. On orthogonality and learning recurrent networks with long term dependencies. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017*, 2017.

Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *In Proceedings Of The International Conference on Representation Learning (ICLR 2015)*, 2015. In Press.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

Scott Wisdom, Thomas Powers, John R. Hershey, Jonathan Le Roux, and Les E. Atlas. Full-capacity unitary recurrent neural networks. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 4880–4888, 2016.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016. URL http://arxiv.org/abs/1609.08144.

Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

Wojciech Zaremba and Ilya Sutskever. Reinforcement learning neural turing machines. *CoRR*, abs/1505.00521, 2015.

F. Zenke, B. Poole, and S. Ganguli. Continual Learning Through Synaptic Intelligence. *arXiv e-prints*, March 2017.

Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks. *arXiv preprint arXiv:1607.03474*, 2016.