

Université de Montréal

On Recurrent and Deep Neural Networks

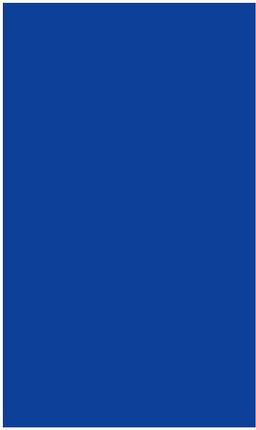
par Razvan Pascanu

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

Mai, 2014

© Razvan Pascanu, 2014.



To:

*Elena for always being by my side and
Sophie for joining us in this crazy adventure*

ACKNOWLEDGMENTS

Looking back to the last five–six years of my life, I was lucky enough to meet a lot of wonderful people who helped me go through the many ups and downs that come with such an endeavour as a PhD. Even if words might not do it justice, I’d like to thank all of them, and apologize in advance for any omissions, which are bound to happen due to the large temporal span, my weak memory and my limited understanding of all the events that lead up to this moment.

I want to thank my advisor, Prof. Yoshua Bengio, for taking a chance on me and helping me survive this long process of self discovery. Going back more than five years, during my master studies it was Prof. Herbert Jaeger who first introduced me to neural networks and laid this path in front of me. I have to thank him for pointing me towards Yoshua when I started looking for possible places to do a PhD, and for vouching for me.

This dissertation would not have happened with out all the people I collaborated and published with. I want to thank Mike Mandel for all his support and advice, it was a pleasure to work with him, and Hugo Larochelle with whom I had the opportunity to work in my first years as a PhD student. I owe Herbert Jaeger for all the help and insights he provided when writing our paper on working memory. I want to thank Tomas Mikolov for his insightful comments and fruitful collaboration that lead to our ICML paper and Nicholas Boulanger-Lewandowski, with whom I explored a large variety of tricks for training recurrent models. Nicholas kindly offered to translate the summary of this work in French, which I would have not been able to do on my own. Thank you.

I owe Aaron Courville and Guillaume Desjardins close to a fourth of this work for pushing me to better understand natural gradient. Thank you both for an exciting and I think important collaboration. Guillaume, thank you for your friendship.

I want to thank Guido Montufar for being so helpful and allowing me to follow my intuitions. I owe him all my theoretical work on the efficiency of deep models, to which he brought formal rigour and elegance. I want to thank KyungHyun Cho who was always willing to help and with whom, in a very short period of time, I managed to collaborate successfully on a lot of projects. His advice was always valuable. Caglar Gulcehere helped me with the dreadful task of running experiments and implementing different architectures, thank you.

Prof. Surya Ganguli, in a short visit to the lab, was able to focus my energy on an interesting optimization problem, saddle points. Yann Dauphin joined in on this task and did a lot of the ground work, resulting in our technical report presented in this thesis. Thank you both for the insightful conversations and ongoing collaboration.

Yoshua was present in almost all these projects, and his advice was always useful.

I want to thank Prof. Neal Stewart for his detailed and very insightful comments,

which improved the quality of this document considerably (even though it is still probably far from being self-sufficient and some results still remain mostly at an intuitive level). Next I want to thank Prof. Herbert Jaeger and Dr. Ilya Sutskever for their comments and suggestions on this document. I've tried to integrate them as best as I could.

I want to thank James and Kaira Bergstra for their support in our first year in Montreal. James mentored and guided me and without him I would not be here. Dumitru Erhan was my first contact with Lisa lab. Thank you Doomie for your help and advice.

Theano, while consuming much of my time, was a key component in my day to day research. I want to thank all the developers, especially Frederic Bastien, without whom our lab would not function, Pascal Lamblin, Olivier Delalleau, Arnaud Bergeron and many others who were there in the development trenches. I hope this project will continue to grow and improve.

I want to thank Prof. Aaron Courville, Prof. Pascal Vincent, Prof. Roland Memisevic and Prof. Douglas Eck who helped shape the Lisa lab and with whom I had the chance to interact during my studies.

I worked together with David Warde-Farley and Steven Pigeon on a project related to motion data compression. Thank you both for your effort and insights. I also need to acknowledge Myriam Cote, who helped us deal with the non-scientific part of this industrial collaboration.

I want to thank Ian Goodfellow for his thoughtful comments on this document, and for the insightful conversations we had in and outside of the lab. I need to acknowledge everyone else who provided feedback on my thesis: Caglar Gulcehre, KyungHyun Cho, Li Yao, Tapani Raiko and Yoshua Bengio.

I want to thank the many members of Lisa lab, with whom I had the pleasure to share most of my last 5 years. I will give just a few names that I have not had a chance to mention yet: Xavier Glorot who was always an inspiration, Li Yao, Mehdi Mirza, Jorg Bornschein, Laurent Dinh, Pierre Luc Carrier, Vincent Dumoulin, Salah Rifali, Xavier Muller, Stanislas Lauly, Tariq Daouda, Gregoire Mesnil, Guillaume Alain, Heng Luo, Amjad Almahairi, etc.

We had many visitors to the lab and with some I had very fruitful and useful interactions. I would like to thank them all for their thoughts and help and especially I want to mention: Tomas Mikolov, Stephan Gouws, Hal Daume III, Holger Schwenk, Thomas Paine.

During my short research years I met many wonderful fellow researchers with whom I hope to keep interacting in the coming years. I want to thank Justin Bayer, with whom I had many useful and insightful conversations, Oriol Vinyals, Tom Schaul, Prof. Benjamin Schrauwen, Ilya Sutskever, Richard Socher, Daan Wierstra, Alex Graves and many, many others.

I want to acknowledge Google DeepMind for sponsoring my last year of PhD, and for the opportunity they offered me afterwards. NSERC, Calcul Québec, Com-

pute Canada, the Canada Research Chairs and CIFAR provided us with research funding and computing support, thank you. I also want to thank Jay Stokes, with whom I had the pleasure to work during my internship, it was a wonderful experience.

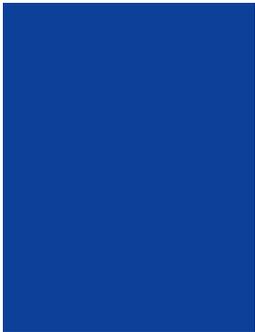
The hardest part for a researcher, I think, is to not lose track of the big picture. It is so easy to get lost in your formulas and experiments and forget about the world around you. I have to thank my friends and family who kept me grounded. Thank you, our old friends who have not forgotten us regardless of the ocean and time difference between us: Dan and Ludmila who were always there for us, Cosmin and Corina, Mishu and Corina, Adrian, Miha, Silviu, Mia, Alex and many others. Without friends in Montreal, this city would have been a very lonely and sad place. I would like to thank those who made both me and Elena feel at home: Julie, Jeff, Konrad, Agatha, Mike, Robert. Thank you all, it is hard to measure how important a friendship is.

The hardships of parenthood, with out the support of having family nearby, can not be underestimated. Neither Elena nor me would have been able to survive it with out the support and friendship of the Power Moms: Lisa, Sarah, Theresa, Katie, Maja, Sunny, Jenny, Wendy, Cathy and their little ones. Elena, Sophie and me, we are all very grateful for having the chance of getting to know you and I know we will stay in touch for many years to follow.

I want to thank my parents and family for their encouragements to pursue my dreams and freedom they gave me in my early years.

All our friends and family helped me to stay grounded and live in the present, but no one did that better than Sophie. Thank you for coming into my life and making me remember every day all the small things that count and with out which life would be meaningless. I hope I can do the same for you some day.

But most of all, the person I owe everything, that helped get up when I was down and that was there through thick and thin, all my gratitude has to go to my loving wife, Elena. I can not stress enough how demanding this whole experience was for her. Thank you for being there and for all the sacrifices you have made.



Résumé

L'apprentissage profond est un domaine de recherche en forte croissance en apprentissage automatique qui est parvenu à des résultats impressionnants dans différentes tâches allant de la classification d'images à la parole, en passant par la modélisation du langage. Les réseaux de neurones récurrents, une sous-classe d'architecture profonde, s'avèrent particulièrement prometteurs. Les réseaux récurrents peuvent capter la structure temporelle dans les données. Ils ont potentiellement la capacité d'apprendre des corrélations entre des événements éloignés dans le temps et d'emmagasiner indéfiniment des informations dans leur mémoire interne.

Dans ce travail, nous tentons d'abord de comprendre pourquoi la profondeur est utile. Similairement à d'autres travaux de la littérature, nos résultats démontrent que les modèles profonds peuvent être plus efficaces pour représenter certaines familles de fonctions comparativement aux modèles peu profonds. Contrairement à ces travaux, nous effectuons notre analyse théorique sur des réseaux profonds acycliques munis de fonctions d'activation linéaires par parties, puisque ce type de modèle est actuellement l'état de l'art dans différentes tâches de classification.

La deuxième partie de cette thèse porte sur le processus d'apprentissage. Nous analysons quelques techniques d'optimisation proposées récemment, telles l'optimisation Hessian free, la descente de gradient naturel et la descente des sous-espaces de Krylov. Nous proposons le cadre théorique des méthodes à région de confiance généralisées et nous montrons que plusieurs de ces algorithmes développés récemment peuvent être vus dans cette perspective. Nous argumentons que certains membres de cette famille d'approches peuvent être mieux adaptés que d'autres à l'optimisation non convexe.

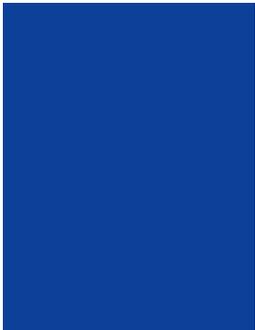
La dernière partie de ce document se concentre sur les réseaux de neurones récurrents. Nous étudions d'abord le concept de mémoire et tentons de répondre aux questions suivantes: Les réseaux récurrents peuvent-ils démontrer une mémoire sans limite? Ce comportement peut-il être appris? Nous montrons que cela est possible si des indices sont fournis durant l'apprentissage.

Ensuite, nous explorons deux problèmes spécifiques à l'entraînement des réseaux récurrents, à savoir la dissipation et l'explosion du gradient. Notre analyse se termine par une solution au problème d'explosion du gradient qui implique de borner la norme du gradient. Nous proposons également un terme de régularisation conçu spécifiquement pour réduire le problème de dissipation du gradient. Sur un ensemble de données synthétique, nous montrons empiriquement que ces mécanismes peuvent permettre aux réseaux récurrents d'apprendre de façon autonome à mémoriser des informations pour une période de temps indéfinie.

Finalement, nous explorons la notion de profondeur dans les réseaux de neurones

récurrents. Comparativement aux réseaux acycliques, la définition de profondeur dans les réseaux récurrents est souvent ambiguë. Nous proposons différentes façons d'ajouter de la profondeur dans les réseaux récurrents et nous évaluons empiriquement ces propositions.

Mots-clés: apprentissage profond, mémoire, gradient naturel, réseaux de neurones, optimisation, réseaux de neurones récurrents, méthodes du second ordre, apprentissage supervisé.



Summary

Deep Learning is a quickly growing area of research in machine learning, providing impressive results on different tasks ranging from image classification to speech and language modelling. In particular, a subclass of deep models, recurrent neural networks, promise even more. Recurrent models can capture the temporal structure in the data. They can learn correlations between events that might be far apart in time and, potentially, store information for unbounded amounts of time in their innate memory.

In this work we first focus on understanding why depth is useful. Similar to other published work, our results prove that deep models can be more efficient at expressing certain families of functions compared to shallow models. Different from other work, we carry out our theoretical analysis on deep feedforward networks with piecewise linear activation functions, the kind of models that have obtained state of the art results on different classification tasks.

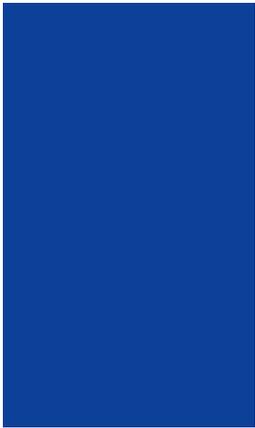
The second part of the thesis looks at the learning process. We analyse a few recently proposed optimization techniques, including Hessian Free Optimization, natural gradient descent and Krylov Subspace Descent. We propose the framework of generalized trust region methods and show that many of these recently proposed algorithms can be viewed from this perspective. We argue that certain members of this family of approaches might be better suited for non-convex optimization than others.

The last part of the document focuses on recurrent neural networks. We start by looking at the concept of memory. The questions we attempt to answer are: Can recurrent models exhibit unbounded memory? Can this behaviour be learnt? We show this to be true if hints are provided during learning.

We explore, afterwards, two specific difficulties of training recurrent models, namely the vanishing gradients and exploding gradients problem. Our analysis concludes with a heuristic solution for the exploding gradients that involves clipping the norm of the gradients. We also propose a specific regularization term meant to address the vanishing gradients problem. On a toy dataset, employing these mechanisms, we provide anecdotal evidence that the recurrent model might be able to learn, with out hints, to exhibit some sort of unbounded memory.

Finally we explore the concept of depth for recurrent neural networks. Compared to feedforward models, for recurrent models the meaning of depth can be ambiguous. We provide several ways in which a recurrent model can be made deep and empirically evaluate these proposals.

Keywords: deep learning, memory, natural gradient, neural network, optimization, recurrent neural networks, second order methods, supervised learning.



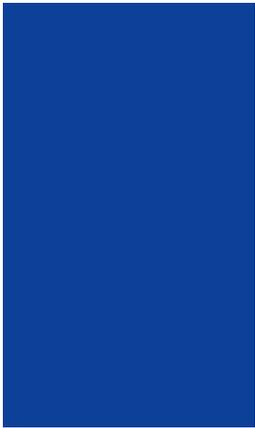
Contents

DEDICATION	ii
ACKNOWLEDGMENTS	iii
Résumé	vi
Summary	viii
Contents	ix
List of Figures	xiii
List of Tables	xv
List of Abbreviations	xvi
1 Introduction	1
1.1 Annotated Overview of Chapters	4
2 Background	9
2.1 Introduction to Machine Learning	9
2.1.1 Regularization	10
2.1.2 Supervised learning	12
2.1.3 Unsupervised Learning	15
2.2 Neural Networks	15
2.2.1 Convolutional neural networks	18
2.2.2 Universality of single hidden layer networks	19
2.2.3 Backpropagation algorithm	21
2.2.4 Pretraining	21
2.3 Recurrent Neural Networks	22
2.3.1 Formal description of Recurrent Neural Networks	26
2.3.2 Backpropagation Through Time	28
2.3.3 Other algorithms for evaluating the gradients and for training recurrent models	31
2.3.4 Difficulties of training recurrent networks	32

2.3.5	Long-Short Term Memory Networks	33
2.3.6	Reservoir Computing	35
2.3.7	Research on recurrent networks	36
2.4	Optimization	37
2.4.1	Gradient Descent and Stochastic Gradient Descent	37
2.4.2	Using curvature	39
2.4.3	Using the structure of the parameter manifold	40
3	The importance of depth for neural networks	44
3.1	Preliminaries	46
3.2	Single hidden layer feedforward model	51
3.3	Deep networks	53
3.3.1	Understanding units in higher layers	56
3.3.2	A special class of deep models	63
3.3.3	Folding the space	67
3.3.4	Identification of Inputs as Space Foldings	70
3.3.5	Symmetries in \mathbb{R}^U	71
3.3.6	Extending the special class of deep models	74
3.3.7	Formal Result	75
3.4	Asymptotic behaviour	78
3.5	Other piecewise linear models	81
3.6	Conclusion and Outlook	82
4	Learning and non-convex optimization	84
4.1	Literature review	85
4.1.1	Second Order methods	85
4.1.2	Conjugate Gradient	92
4.2	Generalized trust region methods	95
4.3	Analysis of certain optimization techniques for deep learning	96
4.3.1	Natural gradient descent	96
4.3.2	Hessian-Free Optimization	101
4.3.3	Natural gradient descent (TONGA)	106
4.3.4	Natural Conjugate Gradient	109
4.3.5	Krylov Subspace Descent	110
4.3.6	Improved natural conjugate gradient	112
4.3.7	Adding second order information – using the curvature of the error	113
4.3.8	Mixing constraints	115
4.4	Saddle-points	116
4.4.1	Understanding saddle points	116
4.4.2	Optimization algorithm and saddle points	121
4.5	Addressing the saddle point problem	126

4.5.1	Constraining the change in the gradient – squared Newton method	126
4.5.2	Limit the influence of second order terms – Saddle-Free Newton Method	128
4.5.3	Benchmark	132
4.6	Properties of natural gradient descent	144
4.6.1	Using unlabelled data	145
4.6.2	Robustness to reorderings of the train set	148
4.7	Summary and Outlook	150
5	Training recurrent neural models and memory	155
5.1	Motivation	156
5.2	Brief introduction to Dynamical Systems for Recurrent Networks	158
5.2.1	Phase portraits and bifurcation diagrams	159
5.3	Working memory	162
5.4	Echo state networks with working memory	166
5.4.1	Model	166
5.4.2	Experiments	169
5.4.3	Analysing the model	174
5.5	Learning: The exploding and vanishing gradients problem	177
5.5.1	The mechanics	179
5.5.2	Linear model	180
5.5.3	Nonlinear model	181
5.5.4	The geometrical interpretation	183
5.5.5	Drawing similarities with Dynamical Systems	186
5.6	Existing solutions for the vanishing and exploding gradients problem	189
5.7	Clipping the gradient norm	191
5.8	Preserving norm by regularization	192
5.9	Learning long term correlations	195
5.9.1	Pathological synthetic problems	195
5.9.2	Overview: Language modelling and polyphonic music prediction	198
5.9.3	Details of the experimental setup and results	200
5.10	Learning to memorize: discussion and summary	207
6	Depth for recurrent models	211
6.1	Motivation	211
6.2	Another Perspective: Neural Operators	213
6.3	Deep Operators	214
6.3.1	Formal descriptions of deep RNNs	218
6.4	Experiments	220
6.4.1	Model Descriptions	221
6.4.2	Training	222

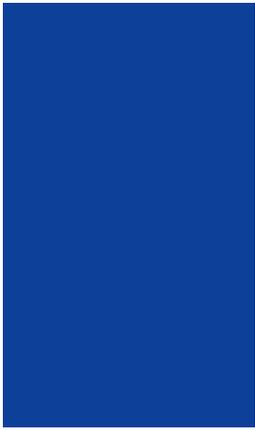
6.4.3	Result and Analysis	223
6.5	Conclusion and Outlook	226
7	A final remark	228
	References	230



List of Figures

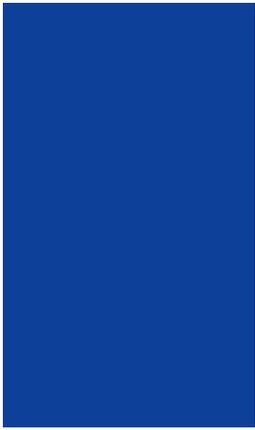
2.1	Multilayer perceptron	16
2.2	Convolutional Neural Network	19
2.3	Information flow in a recurrent model	22
2.4	Unfolding a recurrent neural network in time	28
2.5	Flow of gradient signal	29
2.6	Long Short-Term Memory cell	34
3.1	Partition of space by a single layer rectifier MLP	47
3.2	Number of linear segments:shallow vs deep	49
3.3	Sweep hyperplane method	52
3.4	Depiction of piecewise linear boundary in two dimension	54
3.5	Multiple intersection points of two piecewise linear functions	55
3.6	Visualization for an MLP trained on TFD – layer 1 and 2	59
3.7	Visualization for an MLP trained on TFD – layer 3	60
3.8	Visualization for an MLP trained on TFD – output layer	61
3.9	Points correspondence for the abs activation function	64
3.10	2 layer MLP with a intermediary linear layer	65
3.11	Illustration of a two layer MLP whose units mimic the absolute value function	66
3.12	Function that can be represented efficiently by a deep model	66
3.13	Identification of regions across layers of a deep model	68
3.14	Space folding of 2–D Euclidean space along the two axes.	70
3.15	Non-trivial space folding of 2–D space	71
3.16	Visualization of inputs examples that are mapped to same activation	73
3.17	Folding of the real line into equal-length segments	75
4.1	Newton Method	86
4.2	Trust Region Method	90
4.3	Natural Gradient Descent as a trust region	97
4.4	Different saddle point structures	117
4.5	Qualitative evaluation of different optimization algorithms on small scale task	134
4.6	Training error versus the index and distribution of eigenvalues for different critical points	135
4.7	Benchmark of various optimization algorithms (I)	139

4.8	Benchmark of various optimization algorithms (II)	140
4.9	Benchmark between various optimization algorithms	143
4.10	Using unlabelled data for natural gradient descent	147
4.11	Natural gradient descent is robust to the reordering of the data	149
5.1	Text book examples of attractors	161
5.2	Bifurcation map for the logistic map	163
5.3	Diagram of the WM model	165
5.4	Illustration of synthetic data for WM model	169
5.5	Visualization of the memory states	176
5.6	Gradient norm during learning	184
5.7	Cliff like structure in the error surface of a RNN	185
5.8	Bifurcation diagram of a single hidden unit RNN	187
5.9	Diagram depicting an RNN as a successive application of a series of maps	188
5.10	Diagram decoupling the input contribution and the fix hidden to hidden transition for a RNN	189
5.11	Experimental results on solving the temporal order problem	197
6.1	Recurrent neural network unfolded in time	212
6.2	Operator-based framework for RNNs	213
6.3	Transitions in an RNN	215
6.4	Illustrations of different strategies of making an RNN deep	218



List of Tables

4.1	Best performing hyper-parameters for stochastic gradient descent. . .	135
5.1	Number of erroneous WM states	172
5.2	Trigger characters for false positives, WM model	173
5.3	Average learned output weights for ESN, WM model	173
5.4	Results for polyphonic music prediction by standard RNNs	199
5.5	Addition task	203
5.6	Multiplication task	203
5.7	3-bit temporal order task	204
5.8	Random permutations task	204
5.9	Noiseless memorization problem	205
6.1	Hyper-parameters for deep RNN experiments	221
6.2	Results for deep RNNs on polyphonic music experiments	223
6.3	Results for deep RNNs on language modeling	225



List of Abbreviations

ANN	Artificial Neural Network
BFGS	Broyden-Fletcher-Goldfarb-Shanno algorithm
BPTT	Backpropagation Through Time
CG	Conjugate Gradient
DBM	Deep Boltzmann Machine
ESN	Echo State Networks
ESP	Echo State Property
FIM	Fisher Information matrix
GD	Gradient Descent
HF	Hessian-Free Optimization
KL-divergence	Kullback-Leibler divergence
KSD	Krylov Subspace Descent
LSTM	Long-Short Term Memory network
MLP	Multi-Layer Perceptron
MSGD	mini-batch Stochastic Gradient Descent
NCG	Nonlinear Conjugate Gradient
NatCG	Natural Conjugate Gradient
PCA	Principal Component Analysis
RBM	Restricted Boltzmann Machine
RC	Reservoir Computing
RNN	Recurrent Neural Network
RTRL	Real Time Recurrent Learning
SGD	Stochastic Gradient Descent
TDNN	Time-Delay Neural Network
t-SNE	t-Distributed Stochastic Neighbour Embedding
WM	Working Memory

1

Introduction

“The solution takes the form of a new associationism, or better, since it differs deeply and widely from that older British associationism, of a new connectionism.”

– Edward Thorndike, *The Fundamentals of Learning*, 1932¹

Connectionist approaches to learning is an old idea that dates back at least to the beginning of the last century². Loosely inspired by the structure of the brain, the term itself is somewhat vague, having slightly different particular meanings in different fields of research. We will focus on the “engineering” approach to connectionism which follows in spirit the claim made by [Reeke and Edelman \(1989\)](#), namely that it has more in common with statistical physics and engineering rather than biology. By “engineering” connectionism we mostly refer here to artificial neural networks, where a set of interconnected computational cells are used to learn some target behaviour. The power of this model is hidden in the topology, the strength of the weights connecting the different cells, rather than the single cell itself. Also, our focus is in understanding how and what kind of behaviours these models can learn, rather than if this is how the brain works.

The connectionist movement went in and out of the spotlight during its complicated history. Currently, in the field of machine learning, it gained a lot of popularity under the name of *deep learning*. The seminal work of [Hinton et al. \(2006\)](#); [Bengio et al. \(2007\)](#); [Ranzato et al. \(2007\)](#) showed that, by employing a layer-wise pretraining followed by a global finetuning stage to train a deep multi-layer neural network, the trained model can outperform its shallow counterparts. From this perspective *deep learning* is just a reformulation of connectionism, where special emphasis is put on the depth of the models and their ability to extract useful features from the data. The term *deep learning* is also used to indicate this specific

1. Quote taken from [Medler \(1998\)](#)

2. For a history of the movement, check [Medler \(1998\)](#)

resurgence of neural networks research (which had become somewhat unpopular previous to the above mentioned work of Hinton, Bengio and LeCun) rather than research on neural networks with several layers which kept being carried out even during this period of unpopularity (Schmidhuber, 2014).

A wealth of empirical evidence followed this resurgence, where state of the art results have been obtained on image classification (Krizhevsky et al., 2012; Zeiler and Fergus, 2013; Goodfellow et al., 2013, 2014), speech (Dahl et al., 2010; Hinton et al., 2012; Graves et al., 2013), language modelling (Pascanu, Gulcehre, Cho, and Bengio (2014), Mikolov et al. (2011)), online handwritten recognition (Graves et al., 2009), chaotic systems prediction (Jaeger and Haas, 2004)¹ and drug discovery² to name just a few. Some of these methods are being employed successfully for different vision and NLP tasks by industrial giants like Microsoft, Google, IBM, Yahoo³. Facebook has opened a research center with an emphasis on deep learning⁴.

In spite of all these successes, more work is needed to understand, automate and extend these techniques. Especially since, without properly understanding these models, we could easily end up moving back towards a dark age period for connectionism. This would be induced by over-estimating or over-promising what these models can deliver based on just a few observations. Little is known about how efficient these models can model certain families of functions, or what tasks are fundamentally impossible to model.

We consider our work as an attempt to increase our understanding of some of these open core questions and we hope that these results can help in some small amount to increase even more, if that is possible, the momentum of this renaissance of neural networks. This work is motivated by the belief that artificial intelligence could be obtained via a connectionist approach. This belief is based, for example, on the relationship between artificial neural networks and the brain (which structurally at least seems to use a similar strategy for processing information). Another argument for the ability of neural networks to exhibit intelligence comes

1. This work precedes the work of Hinton et al., but it does rely on recurrent neural models and hence can be seen as an application of *deep learning*

2. Merck Molecular Activity Challenge. <https://www.kaggle.com/c/MerckActivity>

3. Source: <http://gigaom.com/2013/11/01/the-gigaom-guide-to-deep-learning-whos-doing-it-and-why-it-matters/>

4. Source: <http://techcrunch.com/2013/12/09/facebook-artificial-intelligence-lab-lecun/>

from their universal approximator properties, in particular the fact that recurrent models can approximate arbitrarily well any Turing machine. If this observation is coupled with the efficiency of these models to approximate complex behaviour it becomes an argument for their role in constructing an intelligent agent.

The thesis does not cover all projects I took part of while doing my doctoral studies. An incomplete list of work to which I contributed but which is not covered here is as follows:

- In [Gulcehre, Cho, Pascanu, and Bengio \(2014\)](#) we propose a novel activation function inspired by the recently proposed *maxout* activation function ([Goodfellow et al., 2013](#)). We argued that the proposed L_p units can represent more efficiently boundaries of non-stationary curvature by employing a different learnt exponent p for different units.
- In [Desjardins, Pascanu, Courville, and Bengio \(2013\)](#) we provide an efficient implementation of natural gradient for Deep Boltzmann Machines (DBM).
- In [Bengio, Boulanger-Lewandowski, and Pascanu \(2013\)](#) we investigate the effects of a few different alterations proposed for recurrent neural models. Among other things we explore using rectifier units, leaky-integration units (where a low pass filter of different sampled cut-off is set on each hidden unit), a different formulation of Nesterov momentum, etc.
- In [Mandel, Pascanu, Eck, Bengio, Aeillo, Schifanella, and Menczer \(2011\)](#); [Larochelle, Mandel, Pascanu, and Bengio \(2012\)](#) we investigated discriminative restricted Boltzmann machines for multitask learning.
- [Bergstra et al. \(2010\)](#); [Bastien et al. \(2012\)](#) describe Theano, a linear algebra library developed in our group that employs symbolic computations. For this library I had major contributions, among other minor tasks, to the modules related to:
 - building loops (crucial for implementing recurrent models),
 - applying the chain rule in reverse order (the R-operator),
 - doing lazy evaluation.
- During my internship at Microsoft I investigated the use of recurrent neural networks and echo state networks for malware detection.

1.1 Annotated Overview of Chapters

The content of this thesis overlaps with seven different papers that I published while doing my studies, and, some of the content of the thesis has been borrowed directly from these works. As most research carried out in this field, these publications are the result of close collaborations with different colleagues. For any of these works it can be hard to discern exactly to whom does each idea or element of the paper rightfully belong. In this section I will try to do exactly this, and somehow extract my original (main) contributions for each work. By doing so, I hope to highlight, to some degree, my ownership of the content presented in this thesis, which is mostly based on these elements that I considered mine. At the same time, in this section, I will try to provide an overview of the whole document.

Chapter 2 provides a brief introduction to machine learning focused on concepts used in the successive chapters. It covers notions such as *learning*, *supervised training*, *neural networks*, etc. The chapter is meant as a refresher and should be sufficient to build some intuitions on the results presented in all subsequent chapters. Note that, each chapter also comes with its own introduction, where more specific notions to the subject treated in that chapter are introduced.

Chapter 3 contains work done jointly with Dr. Guido F. Montufar, Dr. KyungHyun Cho and Prof. Yoshua Bengio. Most of the content of this chapter is taken from two publications that explore the importance of depth for neural networks.

The first paper, *On the number of response regions for deep feedforward networks with piecewise linear activations* (Pascanu, Montufar, and Bengio, 2014), was accepted at the International Conference on Learning Representations (ICLR) 2014 and was work carried out by me together with Guido Montufar and Yoshua Bengio. In this work, I provided the main construction of the paper (Section 4) and the special case in Section 5 and I was instrumental in defining the question investigated by proposing to look at the problem from a computational geometry perspective. The provided proofs have been, however, written together with Guido Montufar, and he had a big role in improving these proofs. I also helped writing the illustrative proof of Section 3, though the proof itself is a well known result. Most of the text had been written jointly, where each author had a few passes improving different sections of the paper. I argued intuitively the content of Lemma 2 and Lemma 3, though the final provided proofs belong to Guido Montufar. The other

results presented in the paper belong to my co-authors.

The second paper, *On the Number of Linear Regions of Deep Neural Networks* (Montufar, Pascanu, Cho, and Bengio, 2014) is submitted to the Conference on Neural Information Processing Systems (NIPS) 2014 and is work done jointly with Guido Montufar, KyungHyun Cho and Yoshua Bengio. My main contribution was the new geometrical construction used to prove that deep rectifier models can be more efficient than shallow ones. I regard this as one of the main contributions of the paper. I helped, to some extent, in writing the formal description of the number of regions as defined by how intermediate layers identify input regions between them. I provided the illustrative toy experiment represented in Figure 1 and proposed the methodology by which we visualized the different responses of a unit in a intermediary layer of the model. The folding metaphor and the concept of identifying regions were proposed by Guido Montufar as an explanation of the proof in Section 3. Both concepts were further developed by all authors of the paper. The stability to perturbation analysis and the exploration of maxout networks also belongs to Guido Montufar. As for the previous paper, the write-up was the joint effort of all co-authors. I regard both me, KyungHyun Cho and Guido Montufar as having an important role in writing the text, with the role of my co-authors somewhat bigger than mine.

The content of the Chapter 4 explores the connection between different recently proposed optimization algorithms for deep learning, providing a unifying framework. Based on the new understanding of these algorithms we provide several possible variations and discuss one specific problem in non-convex optimization, namely the saddle point problem. We first provide a more in-depth introduction of the topic in Section 4.1.

Parts of Chapter 4 are based on the paper *Revisiting Natural Gradient for Deep Networks* (Pascanu and Bengio, 2014) that was accepted at the International Conference of Learning Representations (ICLR) 2014. This is joint work with Prof. Yoshua Bengio. The derivation of natural gradient descent as a different constrained optimization at each step was the result of a tight collaboration with Dr. Guillaume Desjardins and Prof. Aaron Courville, work that was published in Desjardins, Pascanu, Courville, and Bengio (2013). This derivation however was also employed in our paper in Section 2, and it was modified for neural networks in Section 2.1. The derivation of Section 2.1 is joint work with Yoshua Bengio.

The relationship between natural gradient descent and Hessian-Free Optimization, Krylov Subspace Descent and TONGA (Section 4, 5, 6) were derived by me. Implementation of the algorithm and experiments were also carried out by me. Section 7, 8 and 9 are contributions that I proposed to the paper. However some of the fine details in all sections of this paper are the result of discussions with Yoshua Bengio, and, specifically, the experiment carried out in Section 8 is inspired by such a discussion. The original draft of the text mostly belongs to me, but both me and Yoshua Bengio had several passes improving (sometimes considerably) the write up of several sections. Guillaume Desjardins also provided useful comments for improving the write-up of this paper.

Section 4.4 and 4.5 and all references to the saddle point problem or Saddle-Free Newton method are based on the technical report *On the saddle point problem for non-convex optimization* (Pascanu, Dauphin, Ganguli, and Bengio, 2014). This represents joint work with Yann N. Dauphin, Prof. Surya Ganguli and Prof. Yoshua Bengio. My contributions to this technical report are as follows. I proposed the detailed description of how different optimization techniques behave near saddle-points, Section 2 of the technical report. I proposed the generalized trust region method framework and played a crucial role in defining a proper theoretical justification for our solution, the Saddle-Free Newton algorithm. This algorithm is loosely based on the Square Newton method, an algorithm that I proposed as well but which was excluded from the technical report for brevity. I contributed significantly to the specific proof provided in this technical report for the Saddle-Free Newton, though all co-authors contributed to it with suggestions. Most of the technical report has been written by me with edits from my co-authors. Surya Ganguli helped to greatly improve the literature review that motivates the need to address saddle points, especially the statistical physics literature. All the experimental results provided in the technical report had been carried out by Yann Dauphin. Surya Ganguli and Yoshua Bengio helped define some of the experiments carried out in this technical report.

Chapter 5 contains material published in (Pascanu and Jaeger, 2011; Pascanu, Mikolov, and Bengio, 2013).

The first paper, *A Neurodynamical Model of Working Memory*, is work done together with Prof. Herbert Jaeger and was published in the Neural Networks journal in 2011. In this work we ask the question of whether recurrent models (in

this specific case echo state networks) can exhibit memory able to store information for an unbounded amount of time. We provide a specific model that, using hints during training, can learn to exhibit such behaviour. My contribution to this work consists in the specific model used to obtain this unbounded memory. I also implemented and ran the experiments presented in the paper. The task itself was jointly proposed by me and Herbert Jaeger (where Herbert Jaeger had an important role in defining the question we attempted to answer). I helped in the write-up of the paper. Specifically, I mainly focused on Section 2 and 3 of this work. The introduction and all other sections (4 and 5) belong to Herbert Jaeger (especially the mathematical formalism of Section 4).

The second paper, *On the difficulty of training Recurrent Neural Networks* (Pascanu, Mikolov, and Bengio, 2013), was published at the International Conference on Machine Learning (ICML) 2013. It presents work done together with Dr. Tomas Mikolov and Prof. Yoshua Bengio. In this paper we attempt to improve our understanding of two particular difficulties for training recurrent models and provide heuristic solutions to them. My contribution to this work involved implementing and carrying out all the experiments presented. The analytical description of the vanishing and exploding gradients problem was carried out by me, though it is heavily based on previous work published by Yoshua Bengio. The dynamical system perspective I regard as the result of both I and Yoshua Bengio, where both of us had equal contribution to this view. I believe to have proposed the geometrical interpretation of Section 2.2. The clipping strategy belongs to Tomas Mikolov, and the regularization term used to address the vanishing gradients problem is again joint work between me and Yoshua Bengio (based on Yoshua Bengio's intuition). I contributed heavily to the write-up of the paper, though all co-authors provided modification to the text improving, sometimes considerably, the clarity and flow of the text.

How to Construct Deep Recurrent Neural Networks (Pascanu, Gulcehre, Cho, and Bengio, 2014) is the basis of Chapter 6. This represents work done with Caglar Gulcehre, Dr. KyungHyun Cho and Prof. Dr. Yoshua Bengio. This paper was published at the International Conference of Learning Representations (ICLR) 2014. The main result of the paper, the different architectural changes that induce depth in the model, I regard as my contribution. The operator view of recurrent models resulted from a discussion between Yoshua Bengio, KyungHyun Cho and

myself. I believe to have contributed to this view, though the main idea belongs to Yoshua Bengio. The implementation of these models was carried out by me with some help from Caglar Gulcehre and KyungHyun Cho. The experiments were jointly carried out by me and Caglar Gulcehre. KyungHyun Cho had an important role in the write-up of the paper, though all co-authors, including me, contributed considerably to the text.

Finally, Chapter 7 provides a unifying discussion for this thesis.

2

Background

In this section we provide a brief description of the main concepts of machine learning necessary for understanding this thesis. We do not regard this section as a complete overview of the field. Specifically there are many crucial concepts of machine learning that are not closely related to the content of the subsequent chapters which, for brevity, we do not cover. We also note that some of the definitions provided might be somewhat restrictive (and would not directly cover the entire spectrum of cases) in order to improve clarity.

2.1 Introduction to Machine Learning

Machine learning focuses on finding a suitable model f from a family \mathcal{F} of models that approximates some desired behaviour. One can rely on either a parametric or non-parametric model to learn the target mapping. A well known non-parametric classifier is, for example, the Nearest Neighbour algorithm (Bishop, 2006), which relies on the labels of the known examples *near* the current one in order to predict its label. In what follows we will ignore such families of models and focus mainly on the parametric ones.

A parametric family of models is the set of functions $\mathcal{F} = \{f_\theta | \theta \in \Theta\}$, where $f_\theta(\mathbf{x}) = f_{\mathcal{F}}(\theta, \mathbf{x})$ for some function $f_{\mathcal{F}} : \Theta \times \mathbb{D} \rightarrow \mathbb{T}$, where $\theta \in \Theta$ and $\mathbf{x} \in \mathbb{D}$. The function $f_{\mathcal{F}}$ contains the prior knowledge that we use to construct our family of models, while θ will be learnt and it identifies a specific member of this family. The structure of $f_{\mathcal{F}}$ and, by extension, that of \mathcal{F} dictates the kind of solutions we can obtain, limiting the type of behaviours the model can actually learn.

Learning is the process of finding the most suitable member f^* of a family of models \mathcal{F} for solving some task T , which for a parametric family of models is identical to finding the optimal parameter value $\theta^* \in \Theta$. This usually takes the

form of an iterative optimization process that minimizes some discrepancy measure (also called loss or objective) between the behaviour of the model and the desired behaviour. If π is some distribution over \mathbb{D} and \mathcal{L} is some *suitable task dependent* loss, we can formalize this statement as follows:

$$f^* \leftarrow \arg \min_{f \in \mathcal{F}} \mathbf{E}_{\mathbf{x} \sim \pi} [\mathcal{L}(\mathbf{x}, f)] \quad (2.1)$$

$\mathbf{E}_{\mathbf{x} \sim \pi} [\mathcal{L}(\mathbf{x}, f)]$ is called the expected loss or generalization error and $\mathbf{E}_{\mathbf{x} \sim \pi}$ stands for the expectation over \mathbf{x} sampled from the distribution π . In practice, however, we do not have access to this empirical distribution π . We have to rely on a set of examples $\mathcal{D} = \{\mathbf{x}^{(i)} \sim \pi | 0 < i \leq N\}$ from π , which is called the training set or training dataset. One approach to overcome this limitation is called the *Empirical Risk Minimization* procedure which is described by the following equation:

$$f_{ERM}^* \leftarrow \arg \min_{f \in \mathcal{F}} \frac{1}{|\mathcal{D}|} \sum_{\mathbf{x} \in \mathcal{D}} \mathcal{L}(\mathbf{x}, f) \stackrel{\text{def}}{=} \arg \min_{f \in \mathcal{F}} \mathcal{R}_{EMP}(\mathcal{D}, f) \quad (2.2)$$

We use \mathcal{R}_{EMP} to denote empirical risk or the error on the dataset \mathcal{D} . For probabilistic models this procedure is equivalent to *maximum likelihood learning*.

2.1.1 Regularization

Unfortunately $\mathcal{R}_{EMP}(\mathcal{D}, f)$ is a biased estimate of the expected loss for f . For example, given a sufficiently flexible family of models \mathcal{F} , one can easily end up learning “by heart” the dataset \mathcal{D} with out actually learning the correct mapping for other samples from π .

In order to estimate the generalization error (equation (2.1)) one has to use a *held-out* set of samples from π , $\mathcal{D}_{\text{test}}$, called the test set, where there is no overlap between the training and test set (i.e. their intersection is the empty set). Measuring the empirical risk on this set, $\mathcal{R}_{EMP}(\mathcal{D}_{\text{test}}, f)$ provides a measure for the generalization error. Comparing the error obtained on the test set with the one on the training set, we can see if the model is *over-fitting* or *under-fitting*.

A model is said to be *over-fitting* some training set \mathcal{D} when the model is able to obtain a very small error on the training set, though when tested on new data (which was not part of this set) the error increases considerably. While this definition is somewhat vague, if we consider also the training procedure (which iteratively

updates the parameters of the model such that the training error is minimized), if the error computed on the training set goes down as training progress, while the error computed on the test set is increasing, then the model is over-fitting. A model is *under-fitting* if both errors are still decreasing under training.

To avoid over-fitting there are usually two main approaches. One approach is to change the family of models \mathcal{F} used to one that is not sufficiently flexible to over-fit the data. By doing so, we restrict the search space of functions f . If we are to follow the example of over-fitting given above, it would be sufficient to remove those functions that are able to memorize the training dataset without knowing the distribution π .

Another standard approach is to use a regularization term (also called penalty term). That is we rely on the following equation to find the optimum parameter value θ^* and hence the optimum function f^* :

$$\theta_{ERM}^* \leftarrow \arg \min_{\theta \in \Theta} \mathcal{R}_{EMP}(\mathcal{D}, f_{\theta}) + \lambda \Omega(\theta), \text{ with } \lambda > 0 \text{ and } \forall \theta \in \Theta : \Omega(\theta) > 0 \quad (2.3)$$

The role of the additive regularization term $\Omega(\theta)$ is to restrict in some meaningful way the search space for θ , and therefore to restrict the flexibility of \mathcal{F} . The choice of Ω is model and task specific and there is no further assumption on Ω except that it is differentiable (specifically it does not have to be convex). The term λ is the weight of the regularization term and controls the importance of minimizing Ω versus minimizing \mathcal{R}_{EMP} .

λ can not be learnt as the other parameters, as it could lead to the trivial solution $\lambda = 0$. The value of λ is chosen before learning, and we regard λ as a *hyper-parameter* of our family of model \mathcal{F} . In general, models can have several hyper-parameters, and, for deep neural networks, one has typically tens of them.

The process of selecting the value of the hyper-parameter λ is called *hyper-parameter tuning* or *model selection*. It involves using yet another independent set of samples from our empirical distribution π called the validation set $\mathcal{D}_{\text{valid}}$ (which as before has an empty intersection with both the training and test set). We can find the optimum values of these hyper-parameters by minimizing the empirical risk on the validation set $\mathcal{D}_{\text{valid}}$, where the parameter θ considered for some fixed choice of hyper-parameter values is found by minimizing the regularized error on

the training set \mathcal{D} :

$$\lambda^* = \arg \min_{\lambda \in \mathbb{R}} \mathcal{R}_{EMP} \left(\mathcal{D}_{\text{valid}}, \arg \min_{\theta \in \Theta} \mathcal{R}_{EMP}(\mathcal{D}, f_{\theta}) + \lambda \Omega(\theta) \right) \quad (2.4)$$

Equation (2.4) is written only in terms of a single hyper-parameter $\lambda \in \mathbb{R}$. Usually one relies on a *grid-search* where values for the different hyper-parameters are proposed based on splitting the space into intervals using a grid (and consider one value for each cell of the grid). Alternatively one can use *manual tuning* where the researcher uses their intuition to propose new hyper-parameter values to evaluate based on previous observations. The best values for these hyper-parameters are given in practice by the best performing values among the different proposals. Recently this process of tuning hyper-parameters has received more attention (specifically due to the high number of hyper-parameters for a deep model) and approaches such as *random search* (Bergstra and Bengio, 2012) and *black-box optimization* methods (Snoek et al., 2012; Bergstra et al., 2011) have been proposed.

One particular regularization technique frequently used in practice is *early-stopping*. It relies on monitoring regularly, during the iterative optimization algorithm used to train the model, the error obtained on the validation set and stopping the algorithm as soon as the validation error starts increasing.

2.1.2 Supervised learning

The learning procedure might differ according to the task at hand. One can usually split the different approaches into the family of *Supervised learning* methods and *Unsupervised learning* methods.

For supervised learning the samples $\mathbf{x} \in \mathbb{D}$ take the form of a pair

$$\mathbf{x} = (\mathbf{u}, \mathbf{t}) \in \mathbb{U} \times \mathbb{T}. \quad (2.5)$$

\mathbb{U} and \mathbb{T} are some sets of possible values for the pair (\mathbf{u}, \mathbf{t}) . In such situations we are after learning the true conditional $\pi(\mathbf{t}|\mathbf{u})$ based on the dataset $\mathcal{D} = \{(\mathbf{u}^{(i)}, \mathbf{t}^{(i)}) \sim \pi(\mathbf{u}, \mathbf{t}) | 0 < i \leq N\}$ of identically independently distributed samples.

A standard approach is to regard the model f_{θ} as defining a parametrized conditional probability density function $p_{\theta}(\mathbf{t}|\mathbf{u})$. This association of a random variable

whose distribution is somehow parametrized by f_θ allows us to use knowledge from statistics and probability theory.

Classification

For classification, the task involves assigning a discrete label to every input example \mathbf{u} . A classical example is represented by the MNIST dataset, where the input examples are images depicting a single digit. The task required by this dataset is to recognize the digit in the current image and produce the right label t . The set of labels therefore is $\mathbb{T} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Two-way classification refers to the special case when the cardinality of \mathbb{T} is 2. In this case we regard the output of the model f_θ as being the mean of a Bernoulli distribution

$$p_\theta(t|\mathbf{u}) = \begin{cases} 1 - f_\theta(\mathbf{u}) & , t = 0 \\ f_\theta(\mathbf{u}) & , t = 1 \end{cases}$$

We use the *cross-entropy* (which can be regarded as a similarity measure for distributions) between this conditional Bernoulli distribution and the true conditional distribution $\pi(t|\mathbf{u})$ as our loss which results in :

$$\begin{aligned} \mathcal{L}_{\text{CE}}((t, \mathbf{u}), f_\theta) &= -t \log p_\theta(t|\mathbf{u}) - (1 - t) \log p_\theta(1 - t|\mathbf{u}) \\ &= -t \log f_\theta(\mathbf{u}) - (1 - t) \log(1 - f_\theta(\mathbf{u})) \end{aligned} \quad (2.6)$$

Multiple binary classification, a special case of *Multitask learning*, refers to the case when for the same data point one is required to make several independent binary predictions. One example could be *automatic tagging* for music, where for the same song one would need to predict if several possible tags apply to this specific song or not. In such case we model each binary decision as a two-way classification and use an average over the cross-entropy of each random variable as our loss or objective.

Multi-way classification describes the case when there are more than two possible labels (which are mutually exclusive). It is a standard approach to use in this case a *one-hot* encoding of the output. That is the output of the model is a n dimensional vector, where n is the number of possible classes. To each dimension we

assign a class and the value along that dimension depicts the score (or probability) for the provided input to correspond to that class. The label \mathbf{t} is therefore a vector of 0, with value 1 on dimension k if the corresponding input belongs to class k .

In this case we rely on a Multinoulli¹ distribution for $p_\theta(\mathbf{t}|\mathbf{u})$. We use the negative log likelihood as a loss measure and one can see that in the case of only two classes this reduces to the cross-entropy used for the binary case. The formula for negative log likelihood is as follows:

$$\mathcal{L}_{\text{NLL}}((\mathbf{t}, \mathbf{u}), f_\theta) = \sum_j -\mathbf{t}_j \log p_\theta(\mathbf{t}_j|\mathbf{u}) = \sum_j -\mathbf{t}_j \log f_\theta(\mathbf{u})_j \quad (2.7)$$

Note that $p_\theta(\mathbf{t}_j|\mathbf{u})$ refers to the probability of example \mathbf{u} to belong to class j and $f_\theta(\mathbf{u})_j$ refers to the j -th component of the vector $f_\theta(\mathbf{u})$.

Regression

For regression the set \mathbb{T} is continuous. Such situations arise, for example, when the task involves modeling a reward function in reinforcement learning or motion capture data (where the angle of each joint is a continuous value), etc. One usually makes the assumption that $p_\theta(\mathbf{t}|\mathbf{u})$ is an isotropic Gaussian distribution with some fixed standard deviation σ and the mean given by f_θ , i.e. $p_\theta(\mathbf{t}|\mathbf{u}) = \mathcal{N}(\mathbf{t}; f_\theta(\mathbf{u}), \sigma)$. The loss function in this case is usually taken to be the well known *mean square error* criterion:

$$\mathcal{L}_{\text{MSE}}((\mathbf{t}, \mathbf{u}), f_\theta) = (\mathbf{t} - f_\theta)^2 \quad (2.8)$$

This criterion can be recovered from the negative log likelihood criterion under the Gaussian assumption if we assume σ to be constant and equal to 1, and remove terms that do not depend on θ .

1. Multinoulli refers to a multinomial distribution where the number of trials is fixed to 1. The name reflects the correspondence between Bernoulli and the Binomial distribution. See [Murphy \(2012\)](#) for more information.

2.1.3 Unsupervised Learning

For *unsupervised learning* we do not have a label assigned to each input example \mathbf{u} and the task is about discovering some structure hidden in the set of examples. There are a wide range of algorithms that fall into this category.

Among the main subcategories we have *clustering approaches* such as k-Means (Lloyd, 1982) which attempts to divide the data into several clusters according to the similarity between the examples. *Dimensionality reduction* refers to techniques for reducing the dimensionality of the input while preserving the most important characteristics of the data. Approaches include Principal Component Analysis (PCA) or t-SNE (van der Maaten and Hinton, 2008). *Density estimation* requires one to estimate the underlying distribution π from which the examples in the training set \mathcal{D} were sampled, as how, for example, is done by the RBM model (Freund and Haussler, 1994).

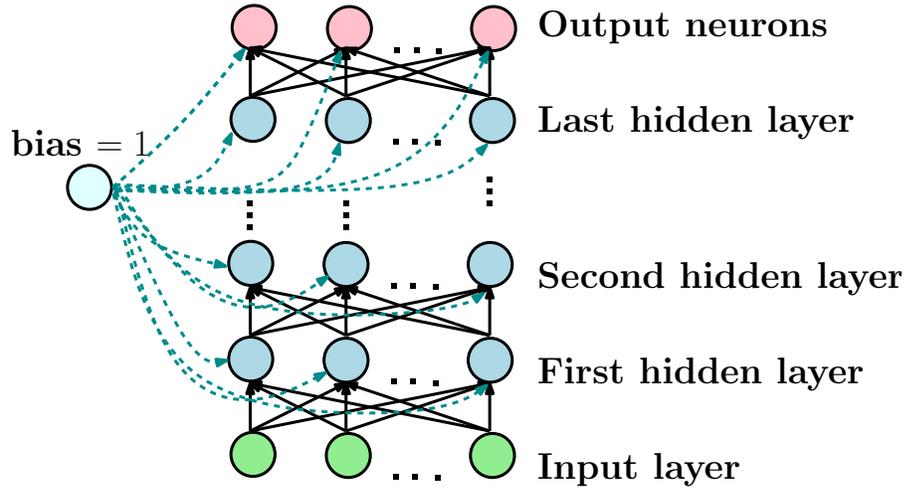
This work is not focused on unsupervised learning and therefore we will not go into any depth to explain these different techniques. However, it is worth mentioning that the task of next step prediction, used to train recurrent neural networks, is a form of unsupervised learning. In this specific situation we provide a sequence \mathbf{u} which is given by a list of consecutive values $\mathbf{u}_{[1]}, \mathbf{u}_{[2]}, \dots, \mathbf{u}_{[T]}$. The model needs to learn to predict $\mathbf{u}_{[k]}$ given $\mathbf{u}_{[1]}, \dots, \mathbf{u}_{[k-1]}$, and therefore the task can be framed as learning a conditional distribution $\pi(\mathbf{u}_{[k]} | \mathbf{u}_{[k-1]}, \dots, \mathbf{u}_{[1]})$. One can choose to regard this as a supervised task where $\mathbf{t} = \mathbf{u}_{[k]}$, though, fundamentally, we are learning the structure of \mathbf{u} in the absence of any label.

2.2 Neural Networks

Artificial Neural Networks (ANN) are a specific family of models. Figure 2.1 depicts the layout of a typical Multilayer Perceptron (MLP). The basic idea behind this model can be traced back to Rosenblatt (1958). Let us first intuitively introduce this model.

Any neural network is formed from a set of units (neurons) that are split into layers (subsets of neurons). Conceptually, there are usually three types of layers: input layers, hidden layers and output layers. The values of the neurons in a *input*

Figure 2.1: Multilayer Perceptron. Neurons are represented as circles, and layers as rows of neurons. Dark cyan dashed arrows represent the bias of each neuron, which is illustrated here as a weighted connection from a special unit that always has a constant value of 1. Black solid arrows are the normal inter-neuronal connections.



layer are fixed by (part of) the provided input to the model (\mathbf{u} introduced in (2.5)). A *hidden layer* is a set of unobserved units. Their values represent intermediate computations of the ANN. Finally, an *output layer* is observed and the union of all output layers is the output ($\mathbf{y} \in \mathbb{T}$) computed by the model. The parameters of such models (the weights and biases) are learnt by an iterative optimization technique such that the output of the model \mathbf{y} matches the target \mathbf{t} provided by the user (or, from a probabilistic perspective, \mathbf{y} matches some property of the distribution $p(\mathbf{t}|\mathbf{u})$, like the mean in the case when $p(\mathbf{t}|\mathbf{u})$ is assumed to be a Gaussian). The specific distance measure used depends on the model and task.

In Figure 2.1, each neuron is represented by a circle. The layers are represented as rows of circles. The input layer is the first row from the bottom of the figure, where the circles are filled with a light green color. In our notation the input layer will be denoted by \mathbf{u} . The following layers, up to the last one are hidden layers, and their value collected in $\mathbf{h}^{(1)}, \mathbf{h}^{(2)}, \dots, \mathbf{h}^{(k)}$. k is the number of hidden layers the model has. Each hidden layer is usually some nonlinear transformation of the input. The last layer is the output layer, denoted by \mathbf{y} . Each layer, usually, has a bias associated, which is also part of the parameters of the model. They are meant to model the average activation of the corresponding neuron, and there is a bias term for every unit in a layer. In Figure 2.1 biases are represented as connections with

a special unit (called bias in the figure) that has a constant value of 1. In our notations we would use $\mathbf{b}^{(1)}, \mathbf{b}^{(2)}, \dots, \mathbf{b}^{(k)}$ for biases. These vectors collect all the dark cyan dashed arrows (weights) in Figure 2.1. The connections between any two consecutive layers are collected in matrices denoted as $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \dots, \mathbf{W}^{(k)}$. The weights and biases $(\mathbf{W}; \mathbf{b})$ represent the parameters of the model, which we will denote by θ sometimes. The role of this network is that when provided some input \mathbf{u} , it should produce some output \mathbf{y} which is close (or identical) to the corresponding target for that input \mathbf{t} . Learning is the process of minimizing the discrepancy between the output of a model and the desired target.

The connectivity pattern of a neural network is usually decided at the layer level, by specifying which layer is connected to which. This forms a computational graph. If the connectivity generates cycles (or this graph contains cycles), then we say the network is recurrent, otherwise it is a feedforward model. If the network has more than one hidden layer (layer of unobserved variables) we say the network is deep, otherwise it is shallow. By adding more hidden layers to a model, a shallow model can be converted into a deep one. Recurrent networks, even if they have a single hidden layer, are usually regarded as deep models.

Given a set of layers and the connectivity pattern, we can now define mathematically the computation carried out by the network. The value of any neuron j , in layer i is defined as the weighted sum of all incoming connections (that is all neurons belonging to layers connected to layer i , each multiplied with the weight of the corresponding connection) and of the corresponding bias term $b_j^{(i)}$, to which an activation function $\sigma^{(i)} : \mathbb{R} \rightarrow \mathbb{R}$ is applied (the activation function is usually applied element-wise, independently to each neuron of the layer). For the MLP, the hidden layers are ordered. Each layer is connected to the next one, where the input is connected to the first hidden layer and the last hidden layer is connected to the output layer. This translates into equations of the following form (for the i -th layer):

$$\mathbf{h}^{(i)} = \sigma^{(i)}(\mathbf{W}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)}) \quad (2.9)$$

where $\sigma^{(i)}$ is a function fixed beforehand and can be layer specific, or the same function for all layers. $\mathbf{h}^{(i)}$ represents the i -th layer in this ordering, and $\mathbf{h}^{(i-1)}$ the previous layer. $\mathbf{W}^{(i)}$ is the weight matrix associated with this layer and $\mathbf{b}^{(i)}$ is the bias. Note that one can not compute the value of layer i without first computing

the value of the layer below. The computations carried here can be seen as a successive composition of simpler functions. A hidden unit at layer i can be seen as detector of a feature in the input, feature somehow encoded in the weights needed to compute this value. One intuition of deep learning is that units in higher layer will discover more complex feature compared to those on the first layer.

Popular activation functions σ are the sigmoid, tanh or rectifier function (which in practice are applied element-wise):

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}} \quad (2.10)$$

$$\text{tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (2.11)$$

$$\text{rect}(x) = \begin{cases} x & , x > 0 \\ 0 & , \text{otherwise} \end{cases} \quad (2.12)$$

For the output layer the activation function usually depends on the task at hand. For example in the case of binary classification or multiple binary classification the sigmoid activation from Equation (2.10) is used (which is guaranteed to have values in $[0, 1]$). For multi-way classification one relies on the softmax activation described below, which is not applied independently to each of the O neurons of the output layer, but rather relies on the whole vector of output activations. For regression one typically uses the identity function id .

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=0}^O e^{x_j}} \quad (2.13)$$

$$\text{id}(x) = x \quad (2.14)$$

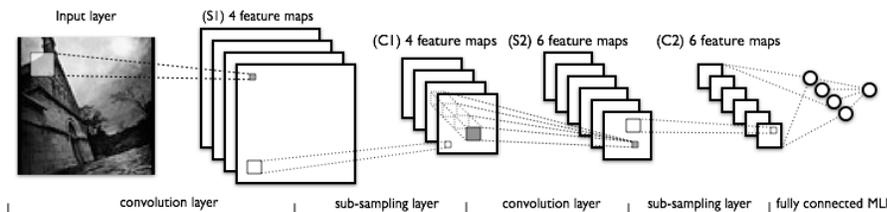
2.2.1 Convolutional neural networks

Convolutional neural network are a special kind of feedforward models inspired by the early work of Hubel and Wiesel on the cat's visual cortex (Hubel and Wiesel, 1968). From this work we know that neurons of the visual system have a complex arrangement and that these cells respond to only a small region of their input space called a *receptive field*. This local connectivity allows for better exploiting the spatial structure of the visual stream. These observations had inspired work

such as the NeoCognitron (Fukushima, 1980) or LeNet-5 (LeCun et al., 1998).

Convolutional networks, directly based on LeNet-5, are composed from several convolutional layers. Each layer’s output is computed by convolving the input (which usually preserves its 2D topological structure) with a set of kernels. The kernels themselves, usually called filters, are learnt. They are the parameters of the layer. Some nonlinear function like the sigmoid or rectifier is applied to the result of this convolution. This result is sometimes referred to as a feature map. The different feature maps obtained for different kernels are also called channels. Additionally, one also applies a pooling layer (or sub-sampling layer), where the output of the convolution with some filter is divided in tiled squared regions and each region is summarized by a single value. This value can be the maximal response within the region in case of max-pooling or the average value of the region in case of mean-pooling. Other pooling operations are possible. Figure 2.2, taken from Desjardins (2009) depicts this architecture.

Figure 2.2: Depiction of the operations carried out by a convolutional neural network. This picture was taken from Desjardins (2009)



2.2.2 Universality of single hidden layer networks

An argument for why neural networks are a viable model for learning is the *universal approximation* property. This property states that any smooth function $f : \mathbb{I}^U \rightarrow \mathbb{R}^O$, from the unit hypercube to \mathbb{R}^O , can be approximated arbitrarily well by a single hidden layer MLP (Hornik et al., 1989). The proof is non-constructive. Constructive proofs (for two hidden layer MLPs) can be seen in Poggio and Girosi (1989); Scarselli and Tsoi (1998).

The “gotcha” with these proofs, is that they suggest the need of a large (prohibitive) number of hidden units, even for conceptually simple functions, which would make the model impractical for any application.

In reality, little is known about the minimum required number of units, or even about the optimal layout. Barron (1994) provides the following lower bound on the risk of a one hidden layer MLP f_{MLP} , trained to approximate a function with one dimensional output $y = f(\mathbf{u})$:

$$R(f_{MLP}) = O\left(\frac{C_f^2}{H}\right) + O\left(\frac{HU}{N} \log(N)\right),$$

where H is the number of hidden units, U is the dimensionality of the input space, $\mathbf{u} \in \mathbb{I}^U$, N is the number of data points in the training set \mathcal{D} , and C_f is a certain spectral measure of complexity of the target function f . This equation suggests that, with an increased number of hidden units, more samples N are needed to train the model (by this we mean that for our optimization algorithm to actually learn the relationship between \mathbf{u} and \mathbf{t} , rather than learning the training set by heart we need to have more training examples). This means that large networks are not impractical only because of their computational complexity, but also because of their data demands which can be very costly or impossible to collect.

Barron (1993) also provides a bound for the risk in case of a fixed number of hidden units H , namely:

$$R(f_{MLP \text{ with } L \text{ units}}) \leq \frac{4C_f}{\sqrt{n_\theta}}, \quad (2.15)$$

where n_θ is the number of adjustable weights in the model. In the same paper, Barron (1993), the bound for a linear mixture of fixed n_θ basis functions is also provided. This is a widely used technique based on approximating function (Taylor expansions, Fourier expansions).

$$R(\text{linear combination of } n \text{ basis functions}) \leq C \frac{C_f}{U} \left(\frac{1}{n_\theta}\right)^{\frac{1}{\nu}}, \quad (2.16)$$

where C is some constant. Comparing these two results is a nice theoretical incentive to study neural networks, since it says that, as the dimensionality of the input increases, you are better off using neural networks to approximate the function than using more standard methods like Taylor expansions.

2.2.3 Backpropagation algorithm

Neural networks define complex non-linear functions (when we have at least one hidden layer) for which there is no closed form solution to compute the optimal parameters. Instead, one has to recurse to iterative methods like *Gradient Descent* (GD) that rely on the gradient at each step to decide how the weights and biases need to be changed.

Intuitively, computing the gradient of the loss with respect to the parameters seems to require $O(n_\theta^2)$ computations. Think of looping over the n_θ parameters and compute the gradient of the loss with respect to the current parameter (order $O(n_\theta)$) by, say, a finite differentiation approach. In reality there are a lot of shared computations between these iterations of the loop, meaning that these gradients can be all computed in $O(n_\theta)$. The backpropagation algorithm, introduced in [Rumelhart et al. \(1986\)](#), does this by relying on two passes through the model, the *forward pass* and the *backward pass*. In the forward pass the output of each neuron is computed and stored (starting from the input towards the output units). The *backward pass* works in reverse. It starts from the output and moves towards the input applying the chain rule at each step to compute the partial derivative of the output with respect to that layer.

2.2.4 Pretraining

In the case of neural networks, *pretraining* is a strategy proposed in [Hinton and Salakhutdinov \(2006\)](#) for training deep models. According to this strategy, for each layer of a deep network some unsupervised model is considered, as for example a Restricted Boltzmann Machine or denosing autoencoder. These models are trained to approximate the distribution of their input and they share parameters with the deep feedforward model.

Specifically one starts with the first layer and trains the unsupervised model to model the input distribution. Once this process is done, the weights of the unsupervised models are used to initialize the weights of the first layer of the deep feedforward model. Then the input is projected through this first layer and the unsupervised model corresponding to the second layer is trained on this projection.

Once the whole model is initialized using this strategy, we have a final stage of training where the deep model is trained on the desired task (process that is called

fine-tuning the model). Recently it has been shown that this complex process is not needed in order to train deep models and therefore we will not introduce the concept of pretraining in any more details.

2.3 Recurrent Neural Networks

Recurrent Neural Networks (RNN) are a special class of neural network models, that have at least one recurrent connection (connection that creates a loop in the graph). The following introduction is loosely based on chapter 9 from [Jaeger \(2009\)](#). Figure 2.3 shows the information flow for an RNN compared to a feedforward network.

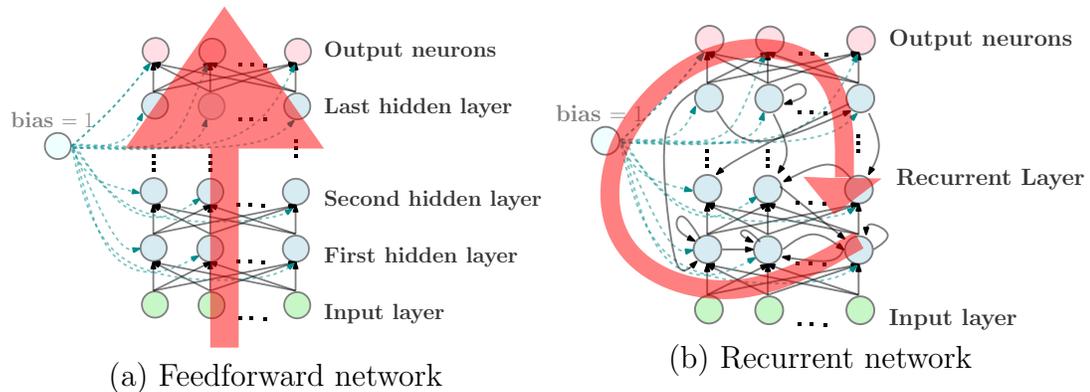


Figure 2.3: Depiction of how information flows in (a) a feedforward network versus (b) a recurrent network. In the first case the information travels from the input to the outputs as indicated by the red arrow going from inputs towards the outputs. In the second, it travels through loops. The red arrow forming a loop indicates that information fed to the model can get trapped inside the model and stay in its memory for several steps or forever (by this we mean that previously received inputs can influence the result of new inputs for an undefined period of time). Note the recurrent connections in (b).

RNNs are qualitatively very different because they do not operate only on a fixed size input space but also on an internal state space containing a transformation of what the network has seen up to the current time step. This is equivalent to an Iterated Function System ([Kalinke and Lehmann, 1998](#)) or a Dynamical System ([Horne et al., 1998](#); [Casey, 1996](#)). The state allows the modelling of temporally/sequentially extended dependencies over unspecified (and potentially infinite)

intervals of time. Compared to feedforward networks, that can approximate non-linear functions arbitrarily well, RNNs are computationally as powerful as any Turing machine (Siegelmann and Sontag, 1991).

Restating this claim, feedforward models learn arbitrary maps that go from some input \mathbf{u} to a desired target \mathbf{t} . Recurrent neural networks have memory, and can *approximate* any dynamical system (similar results, with an emphasis on dynamical systems, are shown in Siegelmann and Sontag (1995)). This means that a recurrent network can encode and execute any “computational algorithm” that one can spell out. Based on the inputs seen in the past, the network can simulate branching behaviour like *if-else* or simply use past inputs to decide on the current ones. These theoretical advantages emphasize the importance of recurrent connections and implicitly the importance of memory.

The proof provided in Siegelmann and Sontag (1991) focuses on showing that a recurrent neural network can compute any function that is computable by a Turing machine, which implicitly means it can simulate the logic of any computer algorithm. Showing the reverse, that a Turing machine can simulate a function generated by a recurrent model is easy if we assume the state of the model to only take rational numbers \mathbb{Q} . If we allow the recurrent neural network to use actual real numbers (which can not be simulated on a computer), Siegelmann (1995) argues that the model can simulate some chaotic dynamical systems like the “shift map” that might not be computable by a Turing machine. *Hypercomputation* is the term coined to describe such behaviours.

Beside assuming the activations to be rational, the proof assumes the use of a hard sigmoid activation function σ for the hidden layers, which is 0 for any value below 0, linear up to 1 and 1 for any value larger than 1. The core idea is to simulate a push-down automaton with three unary stacks which has been previously shown to be as powerful as a Turing machine. The content of each stack is given by a some positive integer s , which can be represented by the binary rational number $0.\underbrace{1\dots 1}_{s \text{ times}}$. The pop operation on the stack is given by $q_s \mapsto 2q_s - 1$, while push corresponds to $q_s \mapsto \frac{1}{2}q_s + \frac{1}{2}$. The control logic (possible decisions that the model has to do based on its state, the stacks and the input symbol) can be hand defined in the weights of the recurrent model. To avoid the need for higher order connections (i.e. multiplications that can be used to gate the desired behaviour based on the control signal), one can rely on negative values which will be truncated to 0 by

the activation function. We invite the interested reader to check the original paper (Siegelmann and Sontag, 1991) for more details.

Another approach for showing the ability of the recurrent network to simulate a Turing machine is taken in Hyötyniemi (1996). Here, a specific structure for the recurrent connection is given to define the four basic operations of any computational language: no operation, increment, decrement and conditional branch. The model uses a hidden unit for each variable of the program, and a hidden unit for each row of the program, with the exception of conditional branches which require three different hidden units. The activation of the recurrent network is the rectifier activation function (equation (2.12)), and the weights are either 1 or -1 . The values of any variable are, therefore, positive integers. Each operation is given by a specific connectivity pattern between the corresponding rows and variables. For example doing nothing at row i simply corresponds to a connection of weight 1 between the unit corresponding to row i and the unit corresponding to row $i + 1$. Please see the original paper for details of these connection patterns. The proof that a recurrent network can simulate any Turing machine is beyond the scope of this introduction and we will not go into any further detail.

Research on recurrent networks has been carried on since the 80's (Elman, 1990; Werbos, 1988, see, e.g.) though it had limited success. One main reason brought up to explain this is the difficulty of training these models by gradient descent (Hochreiter, 1991; Bengio et al., 1994; Hochreiter and Schmidhuber, 1997). There is, however, a plethora of architectures and specific training algorithms that have been proposed in the literature.

The Time-Delay Neural Network (TDNN) (Waibel, 1989; Sejnowski and Rosenberg, 1988) is a compromise between feedforward networks and recurrent neural networks. In order to provide context, a fixed window of the input data is provided at every time step to a feedforward model. The window is composed of the current frame plus a delayed version of previous inputs at each level of the architecture.

While this model can be trained using backpropagation and can take advantage of all the findings for feedforward networks, the maximal acceptable context has to be fixed beforehand. This means that any correlations with events that happened before the maximal delay will not be considered by the model. Another drawback is that the dimensionality of the input increases considerably, imposing larger models that require larger amounts of data and time.

In a simple recurrent neural network, or Elman network, introduced in [Elman and Zipser \(1988\)](#); [Elman \(1990\)](#), there is a hidden layer, a context layer and an output layer. The hidden layer receives connections from the context and the input, while the output receives connections from the hidden layer. The context receives *fixed* 1:1 connections from the hidden layer (these feedback connections become the identity matrix). For a Jordan network ([Jordan, 1990](#); [Cruse, 2009](#)), the context layer receives connections from the output instead of the hidden layer and additionally there are *fixed* 1:1 connections from the context layer to itself. Typically the context layer for a Jordan network is called a state layer. The fixed connections makes learning easy as one can rely on the classical backpropagation algorithm.

Long-Short Term Memory Networks have a considerably more complex connectivity pattern, in which central recurrent connections are only between a unit and itself ([Hochreiter, 1991](#); [Hochreiter and Schmidhuber, 1997](#)). A more detailed description of LSTMs will be provided in Section 2.3.5.

Hierarchical Recurrent Neural Networks are yet another proposal ([El Hihi and Bengio, 1996](#)), where we have several hidden layers, each working at a different time scale, in order to answer to some of the issues of training recurrent networks. Previously, in [Schmidhuber \(1992\)](#), a stacked (or deep) version of RNN is also provided. However, all layers in this architecture receive as input the training data samples ¹, and higher layers are mostly used to improve performance on the misclassified examples of the lower layers. In contrast, [El Hihi and Bengio \(1996\)](#) trains the layers jointly, where the higher layers receive input from the lower ones. This corresponds to a more standard understanding of a deep model.

Recursive Self Organized Maps ([Voegtlin, 2002](#)) have a very specific training algorithm, based on the self organizing strategies used by Kohonen maps ([Kohonen et al., 2001](#)).

Among training algorithms, Backpropagation Through Time ([Werbos, 1988](#)), an extension of backpropagation to recurrent neural networks is the most used approach. Another algorithm is Real Time Recurrent Learning ([Williams and Zipser, 1989](#)) which is well suited for online learning though fairly expensive in terms of computations. Extended Kalman Filter methods ([Puskorius and Feldkamp, 1994](#)) relies on the curvature of the error function and was reported to perform well in

1. Higher layers also receive some encoding of the index of the current time step.

practice. The Atiya-Parlos learning rule (Atiya and Parlos, 2000) constructs virtual targets for the hidden states and then learns the optimal weights to obtain said targets. Hessian-Free Optimization (Martens and Sutskever, 2011) is a second order method that was recently proposed. We will describe these algorithms in more detail in Section 2.3.3.

Evolutionary strategies (Schmidhuber et al., 2007), have also been used to train recurrent networks. In the case of Echo State Networks or Liquid State Machines (Jaeger, 2001; Maass and Bishop, 2001; Lukoševičius and Jaeger, 2009), the recurrent weights are not learnt, but actually sampled from hand-tuned distributions. And this is just a short enumeration of a large variety of models and training techniques.

2.3.1 Formal description of Recurrent Neural Networks

Recurrent Neural Networks expect the input to be presented as a time-series, defined as follows:

Definition 1 (Time-series). A time-series is an ordered list of values $\mathbf{u}_{[1]}, \mathbf{u}_{[2]}, \dots, \mathbf{u}_{[T]}$, where $\mathbf{u}_{[t]} \in \mathbb{U}$. We are interested in modelling time-series where there is a temporal correlation between events at different time steps, that is the sequence is generated by a stochastic process \mathcal{U} such that :

$$\mathbf{u}_{[t]} \sim P(\mathcal{U}_{[t]} | \mathcal{U}_{[t-1]} = \mathbf{u}_{[t-1]}, \mathcal{U}_{[t-2]} = \mathbf{u}_{[t-2]}, \dots) \quad (2.17)$$

We will call two time-series $\{\mathbf{u}\}_{1 \leq t \leq T}$ and $\{\mathbf{y}\}_{1 \leq t \leq T}$ dependent, if the underlying stochastic processes that generate those two series are dependent.

If we consider a dataset \mathcal{D} where every pair $(\mathbf{u}^{(i)}, \mathbf{t}^{(i)})$ is formed of two dependent time-series of the same length, then, considering only discrete time recurrent neural networks, we can formally define these models by the following recurrent relations:

$$\mathbf{h}_{[t]} = \sigma(\mathbf{W}^{(rec)} \mathbf{h}_{[t-1]} + \mathbf{W}^{(in)} \mathbf{u}_{[t]}^{(i)} + \mathbf{b}^{(rec)}) \quad (2.18)$$

$$\mathbf{y}_{[t]} = \sigma^{(out)}(\mathbf{W}^{(out)} \mathbf{h}_{[t]} + \mathbf{b}^{(out)}) \quad (2.19)$$

where $t \in \mathbb{N}$, a positive integer, $t \leq T$, indicating the current time step. Note that the subscript $[t]$ refers to the time step (i.e. which of the *input step* from the

sequence of inputs, and subsequently the value of the model for the corresponding input step). This is different from the upperscript we use to denote the value of different layers (whose value corresponds to the same static input). Or when the subscript or upperscript is used for the input or target, the subscript refers to the time step within the sequence (which element of the sequence), while the upperscript refers to an index among the different sequences in the dataset (which sequence). The weights $\mathbf{W}^{(rec)}$, $\mathbf{W}^{(in)}$ and $\mathbf{W}^{(out)}$ are the weights for the connections amongst hidden units, the connections from the input to the hidden units and from the hidden units to the output units respectively. The bias terms are given by $\mathbf{b}^{(rec)}$ and $\mathbf{b}^{(out)}$.

$\mathbf{u}_{[t]}^{(i)} \in \mathbb{U}$, $\mathbf{y}_{[t]} \in \mathbb{T}$, $\mathbf{h}_{[t]} \in \mathbb{R}^H$ represent the values, at time step t , of the input units, output units (prediction of the model) and hidden units respectively. The vector $\mathbf{h}_{[0]}$ is usually fixed to a constant. σ is the activation function of the hidden layer, usually tanh. $\sigma^{(out)}$ is the activation function of the output layer. The purpose of training is to make the recurrent model, when provided with a sequence $\mathbf{u}^{(i)}$, to produce a sequence similar to $\mathbf{t}^{(i)}$.

Recurrent networks can be applied to non-temporal data by either considering one of the space dimension to be the “time”, as done originally in Graves et al. (2007) or we have done in Pascanu and Jaeger (2011) or by repeating the same input at each time step, or even by just providing no input or an input just for the first step. In the last situation we expect the model to show some specific behaviour based on the initial signal. For example, a recurrent neural network can be used to simulate inference in the MP-DBM model Goodfellow et al. (2013), case where it only receives an initial input at the first time step.

Also, while formally it is easier to define the target as being of the same length as the input, the value of the output units might be irrelevant for many of the time steps. One common case is when we care only about the output value at the end of the input sequence, or when the output is irrelevant for some subset of steps. One example of the second case is given in Honkela et al. (2006). In these cases, when computing the cost, we simply add only the step-wise errors obtained at the relevant time steps.

2.3.2 Backpropagation Through Time

Backpropagation Through Time (BPTT) is an extension of the popular backpropagation algorithm from feedforward network and the most widely used technique to evaluate the gradients of RNNs.

We start by deriving the gradients with respect to the output weights $\mathbf{W}^{(out)}$, gradients that do not depend on the recurrent weights:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(out)}} = \sum_{1 \leq t \leq T} \frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{y}_{[t]}} \frac{\partial \sigma_{out}(\mathbf{W}^{(out)} \mathbf{h}_{[t]} + \mathbf{b}^{(out)})}{\partial \mathbf{W}^{(out)}} \quad (2.20)$$

\mathcal{L} represents some discrepancy measure between the output of the model and the desired target (and it is a choice that the practitioner has to make). The gradient with respect to the output bias $\mathbf{b}^{(out)}$ has a similar form. The tricky gradients are those involving the recurrent weights. To resolve the recurrent connections we rely on *unfolding* the model in time, as depicted in Figure 2.4. That is, for each time step, we construct a clone of the hidden state and replace the recurrent connections of the model to direct connections that go from the hidden state at time $t-1$ to the hidden state at time t . By doing so we obtain a very deep network, but which has tied weights between every layer. The backpropagation through time algorithm is obtained by simply applying the backpropagation algorithm on this graph.

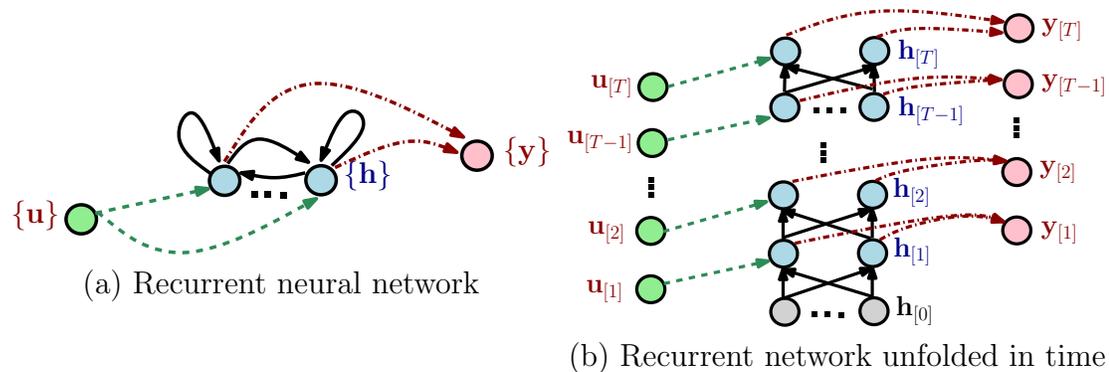


Figure 2.4: Unfolding a recurrent neural network in time. (a) the original network. (b) the deep feed forward network obtained by unfolding. The weights between any two layers (black solid arrows) are tied. The same holds for the input to hidden weights (dashed green arrows) or hidden to output weights (dot-dashed red arrows). The bias was omitted for clarity, as well as some possible connections (e.g. the input is connected to all hidden units)

We will first introduce some new notation before providing the equation describing the gradients with respect to the recurrent weights $\mathbf{W}^{(rec)}$. The gradients

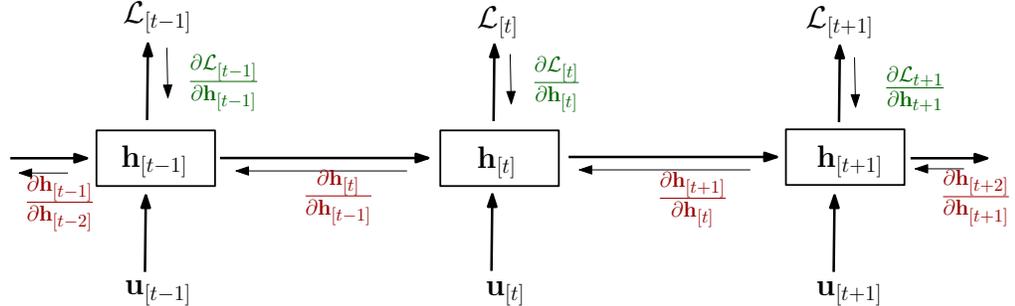
with respect to $\mathbf{W}^{(in)}$ or $\mathbf{b}^{(rec)}$ will have the same form.

Definition 2 (Immediate derivative). Let $\mathbf{h}_{[t]}$ be the result of applying t times some recursive equation $\mathbf{h}_{[k]} = f(\mathbf{h}_{[k-1]}, \theta)$. The ‘‘immediate’’ partial derivative $\frac{\partial^+ \mathbf{h}_{[t]}}{\partial \theta}$ is the derivative of $\mathbf{h}_{[t]}$ with respect to θ where $\mathbf{h}_{[t-1]}$ is considered a constant with respect to θ .

Using Definition 2 we can write the gradient as:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(rec)}} &= \sum_{t=1}^T \frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{h}_{[t]}} \frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{W}^{(rec)}} \\
&= \frac{\partial \mathcal{L}_{[T]}}{\partial \mathbf{h}_{[T]}} \frac{\partial^+ \mathbf{h}_{[T]}}{\partial \mathbf{W}^{(rec)}} + \frac{\partial \mathcal{L}_{[T]}}{\partial \mathbf{h}_{[T]}} \frac{\partial \mathbf{h}_{[T]}}{\partial \mathbf{h}_{[T-1]}} \frac{\partial^+ \mathbf{h}_{[T-1]}}{\partial \mathbf{W}^{(rec)}} + \sum_{t=1}^{T-2} \frac{\partial \mathcal{L}_{[T]}}{\partial \mathbf{h}_{[T]}} \frac{\partial \mathbf{h}_{[T]}}{\partial \mathbf{h}_{[t]}} \frac{\partial^+ \mathbf{h}_{[t]}}{\partial \mathbf{W}^{(rec)}} \\
&\quad + \frac{\partial \mathcal{L}_{[T-1]}}{\partial \mathbf{h}_{[T-1]}} \frac{\partial^+ \mathbf{h}_{[T-1]}}{\partial \mathbf{W}^{(rec)}} + \sum_{t=1}^{T-2} \frac{\partial \mathcal{L}_{[T-1]}}{\partial \mathbf{h}_{[T-1]}} \frac{\partial \mathbf{h}_{[T-1]}}{\partial \mathbf{h}_{[t]}} \frac{\partial^+ \mathbf{h}_{[t]}}{\partial \mathbf{W}^{(rec)}} \\
&\quad + \dots \\
&\quad + \frac{\partial \mathcal{L}_{[1]}}{\partial \mathbf{h}_{[1]}} \frac{\partial \mathbf{h}_{[1]}}{\partial \mathbf{W}^{(rec)}} \\
&= \sum_{t=1}^T \frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{h}_{[t]}} \left[\sum_{k=1}^t \frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}} \frac{\partial^+ \mathbf{h}_{[k]}}{\partial \mathbf{W}^{(rec)}} \right] \tag{2.21}
\end{aligned}$$

Figure 2.5: Flow of gradient signals in a recurrent model. Green indicates the instantaneous component, while red the contribution coming from the future.



In the derivation of Equation (2.21) we use the blue color to indicate the two terms that get multiplied by $\frac{\partial^+ \mathbf{h}_{[T-1]}}{\partial \mathbf{W}^{(rec)}}$. These are $\frac{\partial \mathcal{L}_{[T]}}{\partial \mathbf{h}_{[T]}} \frac{\partial \mathbf{h}_{[T]}}{\partial \mathbf{h}_{[T-1]}}$ and $\frac{\partial \mathcal{L}_{[T-1]}}{\partial \mathbf{h}_{[T-1]}}$. In red we show the term that gets multiplied by $\frac{\partial^+ \mathbf{h}_{[T]}}{\partial \mathbf{W}^{(rec)}}$, namely $\frac{\partial \mathcal{L}_{[T]}}{\partial \mathbf{h}_{[T]}}$. Note that one can

obtain the sum of the terms for $\frac{\partial^+ \mathbf{h}_{[T-1]}}{\partial \mathbf{W}^{(rec)}}$ from the one for $\frac{\partial^+ \mathbf{h}_{[T]}}{\partial \mathbf{W}^{(rec)}}$ by first multiplying with $\frac{\partial \mathbf{h}_{[T]}}{\partial \mathbf{h}_{[T-1]}}$ and then adding $\frac{\partial \mathcal{L}_{[T-1]}}{\partial \mathbf{h}_{[T-1]}}$. This holds also for the subsequent steps and indeed it is the key observation behind BPTT.

Algorithm 1 Backpropagation Through Time

Let \mathbf{h}_0 be the initial state of the recurrent network, $\mathbf{W}^{(rec)}$, $\mathbf{W}^{(in)}$, $\mathbf{W}^{(out)}$ the weights, \mathcal{L} the loss function, $\{\mathbf{u}\}$ the input and $\{\mathbf{t}\}$ the target.

- 1: Initialize $\mathbf{gW}^{(rec)}$, $\mathbf{gW}^{(out)}$, $\mathbf{gW}^{(in)}$ with 0.
 - 2: **for** $t = 1$ to N **do**
 - 3: $\mathbf{h}_{[t]} \leftarrow \sigma(\mathbf{W}^{(rec)} \mathbf{h}_{[t-1]} + \mathbf{W}^{(in)} \mathbf{u}_{[t]})$
 - 4: $\mathbf{y}_{[t]} \leftarrow \sigma_{out}(\mathbf{W}^{(out)} \mathbf{h}_{[t]})$
 - 5: **end for**
 - 6: $\mathbf{gW}^{(out)} \leftarrow \left(\frac{\partial \mathcal{L}(\mathbf{y}_{[T]}, \mathbf{t}_{[T]})}{\partial \mathbf{W}^{(out)}} \right)$
 - 7: $g_{\mathbf{h}} \leftarrow \left(\frac{\partial \mathcal{L}(\mathbf{y}_{[T]}, \mathbf{t}_{[T]})}{\partial \mathbf{h}_{[T]}} \right)$
 - 8: $\mathbf{gW}^{(in)} \leftarrow g_{\mathbf{h}} \cdot \left(\frac{\partial^+ \mathbf{h}_{[T]}}{\partial \mathbf{W}^{(in)}} \right)$
 - 9: $\mathbf{gW}^{(rec)} \leftarrow g_{\mathbf{h}} \cdot \left(\frac{\partial^+ \mathbf{h}_{[T]}}{\partial \mathbf{W}^{(rec)}} \right)$
 - 10: **for** $t = T-1$ down to 1 **do**
 - 11: $g_{\mathbf{h}} \leftarrow g_{\mathbf{h}} \cdot \left(\frac{\partial \mathbf{h}_{[t+1]}}{\partial \mathbf{h}_{[t]}} \right) + \left(\frac{\partial \mathcal{L}(\mathbf{y}_{[t]}, \mathbf{t}_{[t]})}{\partial \mathbf{h}_{[t]}} \right)$
 - 12: $\mathbf{gW}^{(out)} \leftarrow \mathbf{gW}^{(out)} + \left(\frac{\partial \mathcal{L}(\mathbf{y}_{[t]}, \mathbf{t}_{[t]})}{\partial \mathbf{W}^{(out)}} \right)$
 - 13: $\mathbf{gW}^{(in)} \leftarrow \mathbf{gW}^{(in)} + g_{\mathbf{h}} \cdot \left(\frac{\partial^+ \mathbf{h}_{[t]}}{\partial \mathbf{W}^{(in)}} \right)$
 - 14: $\mathbf{gW}^{(rec)} \leftarrow \mathbf{gW}^{(rec)} + g_{\mathbf{h}} \cdot \left(\frac{\partial^+ \mathbf{h}_{[t]}}{\partial \mathbf{W}^{(rec)}} \right)$
 - 15: **end for**
-

We can now define an efficient algorithm for computing the gradients by looping backwards in time, from T to 1 and constructing recursively the terms that have to be multiplied by $\frac{\partial^+ \mathbf{h}_{[k]}}{\partial \mathbf{W}^{(rec)}}$. This results in a time complexity of $\Theta(n_{\theta}T)$, with a space complexity of $\Theta(HT)$ (Williams and Peng, 1990), where n_{θ} is the number of weights, H is the number of hidden units and T is the length of the sequence.

Pseudo-code describing BPTT is provided as algorithm 1. Figure 2.5 depicts the flow of the gradients backwards in time and can be used as a map to figure out the BPTT algorithm. It shows, at each time step going backwards, the two terms that need to be considered, namely the instantaneous contribution and the rescaled contribution from the future.

2.3.3 Other algorithms for evaluating the gradients and for training recurrent models

Real Time Recurrent Learning (RTRL) (Williams and Zipser, 1989) is another algorithm for evaluating the gradients of a recurrent model. The approach is mathematically straightforward and, at least in principle, suitable for online learning (where we only have access of a training sample at each step of the iterative algorithm, and can not access the whole dataset before hand). The algorithm is obtained by differentiating the equations describing the network dynamics. We provide below the equation for $\mathbf{W}^{(rec)}$. For other weight matrices the equations are very similar.

$$\frac{\partial \mathbf{h}_{[t+1]}}{\partial \mathbf{W}^{(rec)}} = \frac{\partial \mathbf{h}_{[t+1]}}{\partial \mathbf{h}_{[t]}} \frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{W}^{(rec)}} + \frac{\partial^+ \mathbf{h}_{[t+1]}}{\partial \mathbf{W}^{(rec)}} \quad (2.22)$$

The first term of the equation corresponds to the implicit effect due to the network dynamics (it describes how, via these parameters, the current state influences future states of the model), while the second term represents the explicit or instantaneous contribution to the gradients. In blue we indicate the factor $\frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{W}^{(rec)}}$ which is computed recursively.

Implementing RTRL means propagating Equation (2.22) forward in time. This yields a complexity in the order of $\Theta(n_\theta^2 T)$, where n_θ is the total number of weights in the model and T is the number of steps taken by the model (length of the input sequence). Because it grows with the square of n_θ , this algorithm can be prohibitive even for medium sized networks.

A different approach for training RNNs, introduced in Puskorius and Feldkamp (1994), is to apply the Extended Kalman Filter algorithm. In this view, we assume there exist a set of weights θ^* that solves the task. We then assume that these weights are the state of the Kalman filter, and that the output is a time dependent observation function of these weights (we need to make the input of the model part of the output function and hence we get the time dependent function).

Training the network in this framework becomes equivalent to estimating the state θ^* from an initial guess and a sequence of outputs. Compared to SGD where the gradients are evaluated by either BPTT or RTRL, EKF falls into the category of second order methods since it makes use of the curvature information (the Hessian). Another method in this same category is the Hessian-Free Optimization (Martens

and Sutskever, 2011) which is a truncated Newton approach to a second order method. We showed in Pascanu and Bengio (2014) that Hessian-Free Optimization is equivalent to natural gradient descent and therefore one can see it as a first order method rather than a second order one.

The Atyia-Parlos rule provides another approach for computing the gradients of the model. In this approach we construct “virtual targets” for the hidden state by considering the gradient of the cost with respect to the unfolded hidden states of the model. Based on these targets, one can now use a closed form solution that gives the optimum weights for achieving the desired behaviour. This algorithm was further modified in Steil (2004), where the Backpropagation Decorrelation algorithm, an approximation of the original Atyia-Parlos rule with a lower computational cost is explored.

As we will also state in Section 2.3.7, there is no good benchmarking of these different approaches on the same task. So, in some sense, there is no good evidence that some method is better than another. As argued in Atiya and Parlos (2000) many of these algorithms result in computing the same gradients. In general convergence analysis is not available for these models as they are non-convex.

2.3.4 Difficulties of training recurrent networks

In Hochreiter (1991); Bengio et al. (1994); Hochreiter and Schmidhuber (1997) two specific problems with training recurrent neural networks by gradient descent are described. To quickly summarize them let us repeat the equation of the gradients with respect to the recurrent weights, previously given in equation (2.21), and use colour coding to indicate the different terms involved.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(rec)}} = \sum_{t=1}^T \frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{h}_{[t]}} \left(\sum_{k=1}^t \frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}} \frac{\partial^+ \mathbf{h}_{[k]}}{\partial \mathbf{W}^{(rec)}} \right) \quad (2.23)$$

The most important term in this equation is the Jacobian $\frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}}$ depicted in red which takes the form of a product of Jacobians

$$\frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}} = \prod_{j=k}^{t-1} \frac{\partial \mathbf{h}_{[j+1]}}{\partial \mathbf{h}_{[j]}}.$$

All these Jacobians have a very similar form, being equal to the multiplication of a diagonal matrix which has $\sigma'(\mathbf{h}_{[j+1]})$ on the diagonal and $\mathbf{W}^{(rec)T}$.

Bengio et al. (1994) argues that the product of $t - k$ matrices can behave similarly to a product of $t - k$ real numbers. If all numbers are smaller than 1, and we multiply them together, their product will go exponentially fast to 0 (exponential in $t - k$). The same is true for the norm of $t - k$ matrices if all their singular values are smaller than 1. This behaviour is called the *vanishing gradient* problem.

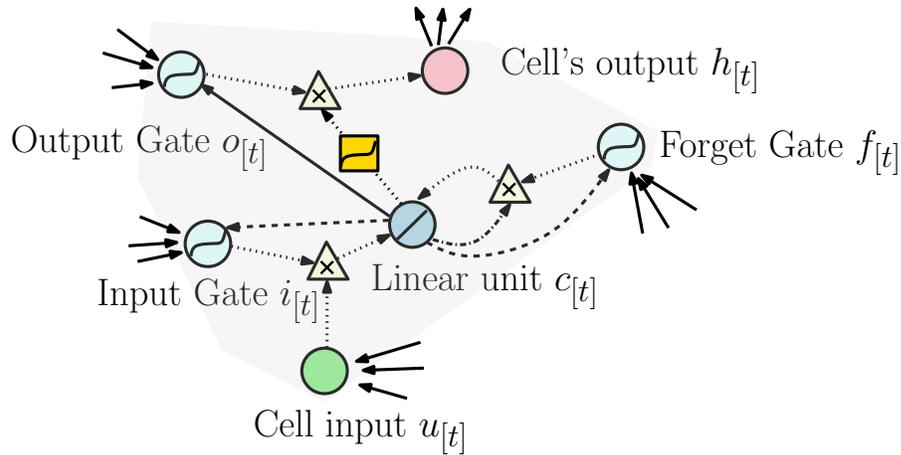
If all numbers are larger than 1, on the other hand, the norm of their product will quickly grow towards infinity, exponential in $t - k$. If the matrices are aligned along some direction (the eigenvector of the dominant eigenvalue are aligned), and the corresponding singular value is larger than 1, a similar behaviour can happen when multiplying $t - k$ matrices. This results in an explosion of the gradient norm, and hence it bears the name of the *exploding gradient* problem.

2.3.5 Long-Short Term Memory Networks

In Hochreiter (1991); Hochreiter and Schmidhuber (1997) the Long short-term memory network is proposed, which is meant to address the *vanishing gradient* problem. This is achieved by altering the structure of the network, where a subset of neurons (or all of them) are replaced by memory cells. Figure 2.6 depicts such a cell. Because of the linear activation of the single neuron in the cell, with a self connection weight of 1, the unit behaves like a memory buffer, being able to store one bit of information for potentially unbounded periods of time. The linear activation and self connection of value 1 also means that the gradient through this connection does not lose norm, and hence does not *vanish*.

Each memory cell in the recurrent model is connected to any other memory cell. The model can also have simple cells among the memory cells described above. Equation (2.24) describes (in vector format) the computations done by a LSTM network of only memory cells, where \mathbf{i} stands for the input gates, \mathbf{f} for the forget gate, \mathbf{o} for the output gate, \mathbf{c} for the value of the linear unit at the center of the cell and \mathbf{u} for the input and \mathbf{h} for the output of the cells. We use superscripts to identify the different weight matrices connecting the different kinds of units. Similar equations are given in Graves (2013).

Figure 2.6: Long Short-Term Memory Network’s memory cell. Circles stand for units, while triangles depict a multiplication between the incoming connections. At the center of the cell we have a single linear unit with a self connection of value 1 (though its value is multiplied by the forget gate). The input and output of the cell are guarded by the input and output gate. The dashed and dashed-dotted arrows indicate how the recurrent connection is solved, by showing which connections rely on the previous value of the linear unit. Namely, we first evaluate the input gate and forget gate using the value of the linear unit at the previous time. We use the same previous value of the unit to compute the new one, relying on the forget gate. The dashed-dotted and dotted arrows indicate a fixed weight of 1. After the new activation of the linear unit is computed, the output gate and the output of the cell is evaluated. The square indicates that a sigmoid activation is applied before multiplying the value of the linear cell with the output gate. All the gates have a sigmoidal activation and can take values between 0 and 1. See equation (2.24) for the computations carried out by LSTMs.



$$\begin{aligned}
 \mathbf{i}_{[t]} &= \text{sigmoid}(\mathbf{W}^{(ui)}\mathbf{u}_{[t]} + \mathbf{W}^{(hi)}\mathbf{h}_{[t-1]} + \mathbf{W}^{(ci)}\mathbf{c}_{[t-1]} + \mathbf{b}^{(i)}) \\
 \mathbf{f}_{[t]} &= \text{sigmoid}(\mathbf{W}^{(uf)}\mathbf{u}_{[t]} + \mathbf{W}^{(hf)}\mathbf{h}_{[t-1]} + \mathbf{W}^{(cf)}\mathbf{c}_{[t-1]} + \mathbf{b}^{(f)}) \\
 \mathbf{c}_{[t]} &= \mathbf{f}_{[t]}\mathbf{c}_{[t-1]} + \mathbf{i}_{[t]}\sigma(\mathbf{W}^{(uc)}\mathbf{u}_{[t]} + \mathbf{W}^{(hc)}\mathbf{h}_{[t-1]} + \mathbf{b}^{(c)}) \\
 \mathbf{o}_{[t]} &= \text{sigmoid}(\mathbf{W}^{(uo)}\mathbf{u}_{[t]} + \mathbf{W}^{(ho)}\mathbf{h}_{[t-1]} + \mathbf{W}^{(co)}\mathbf{c}_{[t]} + \mathbf{b}^{(o)}) \\
 \mathbf{h}_{[t]} &= \mathbf{o}_{[t]}\sigma(\mathbf{c}_{[t]})
 \end{aligned} \tag{2.24}$$

If the input gate is mostly open, or even in other circumstances, the gradients can *explode*. LSTM networks are not meant to address the exploding gradient problem. Note also that the bias for all the gates involved are usually initialized to some large negative value to ensure that the gates are closed most of the time.

The network proved to be quite successful on a set of artificial tasks, specifically designed to measure long term dependencies (Hochreiter and Schmidhuber, 1997). The basic model, as well as alterations of it, proved to also be successful giving

state of the art results on tasks, including music modeling (Eck and Schmidhuber, 2002), speech (Graves et al., 2013), online handwritten recognition (Graves et al., 2009).

2.3.6 Reservoir Computing

Reservoir Computing (RC) is a paradigm for training recurrent networks. The seminal work of Jaeger (2001) introduced this concept for machine learning, Maass and Bishop (2001) introduced the idea in computational neuroscience, Dominey et al. (1995) for cognitive science and Steil (2004) for cognition and robotics.

The approach is intended to take advantage of the rich dynamics of randomly initialized recurrent neural networks to construct a random *temporal* projection of the input in some high dimensional space. From this random projections a linear map is computed that regresses the random values to the desired targets.

This model has shown to be able to succeed on many tasks, from wireless communication (Jaeger and Haas, 2004), to financial data prediction (Ilies et al., 2007) and speech recognition (Verstraeten et al., 2006). For a survey of the field see Lukoševičius and Jaeger (2009).

Note that in order to preserve stability of the random projection, the random weights need to satisfy the *echo state property* (ESP) which implies that any impulse fed to the model has to die out in due time. In Buehner and Young (2006) a sufficient condition is provided for a model to have the echo state property. The recurrent weight matrix needs to be diagonally Schur stable. In practice, however, most people control the magnitude of the spectral radius (absolute value of the largest eigenvalue) of the recurrent weight. This is usually set to be less than unity, even though this is neither a sufficient nor necessary condition. A sufficient strategy would be to set the spectral radius to one for the matrix obtained by taking the absolute value entry-wise to the recurrent weight matrix (Buehner and Young, 2006).

This property also entails that the model always exhibits fading memory, and can not deal with long term dependencies. In Lukosevicius et al. (2007), leaky-integrating neurons are introduced as a method of coping with longer term dependencies. A leaky-integration neuron is defined in the following equation:

$$\mathbf{h}_{[t+1]} = (1 - \eta)\sigma(\mathbf{W}^{(rec)}\mathbf{h}_{[t]} + \mathbf{W}^{(in)}\mathbf{u}_{[t+1]} + \mathbf{b}^{(rec)}) + \eta\mathbf{h}_{[t]} \quad (2.25)$$

By controlling the leak rate $\eta \in [0, 1]$, the neuron activation changes at a slower or faster time scale. The formulation above is equivalent to applying a low-pass filter on each neuron, where the leak-rate η controls the cutoff frequency of the filter. This concept can be extended to other filters as well (band-pass, high-pass) as shown in [Holzmann and Hauser \(2009\)](#).

2.3.7 Research on recurrent networks

While recurrent models have a long history, they have been understudied because of the difficulties associated with training them. Recent successes of deep learning, and some important results recently reported on LSTM networks ([Graves et al., 2013, 2009](#)), classical RNNs ([Pascanu, Mikolov, and Bengio \(2013\)](#); [Pascanu, Gulcehre, Cho, and Bengio \(2014\)](#) and [Sutskever et al. \(2013\)](#)), RNNs trained with Hessian-Free Optimization ([Martens and Sutskever, 2011](#); [Sutskever et al., 2011](#)) or ESN networks ([Jaeger and Haas, 2004](#); [Jaeger, 2013](#)) seem to have raised the interest in these models again.

Unfortunately, there is no *de facto* best approach to train RNNs or even a *de facto* RNN structure. A head-to-head comparison between many of the approaches enumerated in this introduction (see Section 2.3.3) is lacking. Recent work seems to be biased however towards relying on BPTT to compute the gradients and to rely on either SGD with some minor modifications or a modified Hessian-Free Optimization algorithm. In terms of the structure, in practice it is also not clear what are the advantages and disadvantages of LSTM networks compared to RNNs or ESN models¹. It is also highly possible that these advantages or disadvantages will be task dependent, based on the kind of temporal correlations the data shows, or the amount of memory required to address the task.

In our work we will focus on standard RNNs trained with SGD where the

1. It is well understood that an ESN constructs a random temporal projection from which it composes the desired target. This approach behaves well when the projection is done in a higher dimensional space, so as to avoid collisions between different input patterns. If the complexity of the input increases this approach is bound to do worse. However, it is unclear to what extent this limits the model. For example, a *divide et impera* strategy can be used to break the complex task into simpler ones that can be well addressed by multiple ESNs.

gradients are given by the BPTT algorithm. We believe that many of our findings extend to the other structures and learning algorithms.

2.4 Optimization

As we discussed previously in Section 2.1, Equation (2.2), learning neural networks can be formulated as an optimization task, where we need to find the optimum function f^* from a family of functions \mathcal{F} . The optimization perspective to learning can be at times misleading. This is mostly caused by the over-fitting effect. Simply stated, we usually end up minimizing a surrogate cost, the empirical risk on some training dataset. As such, we do not care for getting the best model that minimizes this cost; we care about the model that minimizes the expected loss. Note that for neural networks, this cost usually is non-convex and a proper mathematical analysis of the convergence properties of this optimization technique is unpractical.

Usually this discrepancy can be resolved by using techniques such as regularization, which we mentioned in Section 2.1.1. However, while learning involves also dealing with the over-fitting effect, at the core of it, minimizing the empirical risk is usually done via some optimization technique.

In the subsequent subsections we will focus on some optimization techniques usually employed for learning. Specifically we will look at a few gradient based optimization techniques.

2.4.1 Gradient Descent and Stochastic Gradient Descent

Gradient Descent (GD) relies on the first order derivatives of the differentiable loss function \mathcal{L} (which typically is non-convex and measures some distance between the output of the model and the desired targets) with respect to the parameters of the model θ . The algorithm is iterative. At each step t , if $\theta_{[t]}$ is the current value of the parameters, we consider a first order Taylor expansion of the loss function \mathcal{L} around $\theta_{[t]}$. We can see that taking a step in the opposite direction of the gradient leads to decreasing the value of this Taylor expansion. If the step is small enough,

such that the approximation of \mathcal{L} is reliable, then the step will result in minimizing the actual function \mathcal{L} as well in practice.

Note that gradients are given by the Jacobian $\mathbf{J} = \left[\frac{\partial \mathcal{L}}{\partial \theta_1}, \frac{\partial \mathcal{L}}{\partial \theta_2}, \dots, \frac{\partial \mathcal{L}}{\partial \theta_{n_\theta}} \right]$, which is a row vector. We will also use the notation $\nabla \mathcal{L}$ for this quantity. In general, we will represent gradients as row vectors. The algorithm for implementing gradient descent is given below. The step size is controlled by the learning rate $\lambda \in \mathbb{R}_+$ which is a small value multiplied with each gradient.

Algorithm 2 Gradient Descent Algorithm

```
1: Initialize the model by  $\theta_{[0]}$ .
2: while Stopping condition is not met do
3:    $\Delta\theta \leftarrow 0$ 
4:   for all  $\mathbf{x} \in \mathcal{D}$  do
5:      $\Delta\theta \leftarrow \Delta\theta - \left( \nabla \mathcal{L} \left( \mathbf{x}, f_{\theta_{[t]}} \right) \right)^T$ 
6:   end for
7:    $\theta_{[t+1]} \leftarrow \theta_{[t]} + \lambda \Delta\theta$ 
8: end while
```

Gradient Descent can prove to be quite inefficient in practice, especially when one has a large dataset \mathcal{D} . A better alternative is to approximate the step $\Delta\theta$ taken by GD using a single data example, or just a few of them, leading to better convergence speed (LeCun et al., 1998). One obtains, in this way, Stochastic Gradient Descent (SGD) or mini-batch Stochastic Gradient Descent (MSGD). A *mini-batch* in this context refers to a subset of examples from the training set that are used to obtain the gradient (i.e. the training set is divided in a large number of mini-batches and at each step one of these mini-batches are used to obtain the gradient). We will also use the term *epoch* to refer to the number of updates needed to go once over all the examples in the training dataset (in the case of MSGD, this equals to the number of training examples divided by the mini-batch size).

These approaches exploit the redundancies in the data and rely on estimates of the gradient that will point roughly in the right direction. This is sufficient to make progress towards a (local) minimum. It has been argued, additionally, that the noise introduced by this stochasticity might also be useful for escaping narrow minima besides simply speeding up learning.

MSGD is an improvement over SGD in the sense that it reduces the variance in the estimation of the gradients by using several examples at a most 0 cost on

modern computers that can take advantage of SIMD instruction and parallelize these computations. Reduced variance allows for larger steps (larger learning rate λ) and, arguably, faster convergence. The algorithm is also easily parallelizable along different examples which makes it optimal for modern computer architectures.

SGD and MSGD have seen several alteration meant to speed up convergence. One of the more popular approaches being the addition of *momentum* (Nocedal and Wright, 2006) which uses a running average of the gradients. Other approaches mostly focus on adapting the learning rate (sometimes considering a different learning rate for each parameter). Some of these approaches include AdaGrad (Duchi et al., 2011), the adaptive learning rate technique from Schaul and LeCun (2013), etc.

2.4.2 Using curvature

Another technique to improve convergence is to rely on a better approximation of the function \mathcal{L} . Specifically one can use a second order Taylor expansion rather than a first order one.

If one does this, one gets the widely known *Newton's Method* (Nocedal and Wright, 2006). The algorithm is described by the equation below, where $\Delta\theta$ is given by the step required to minimize this second order Taylor approximation of the function \mathcal{L} .

$$\Delta\theta \leftarrow \arg \min_{\Delta\theta} \mathcal{L}(\theta) + \nabla\mathcal{L}\Delta\theta + \Delta\theta^T\mathbf{H}\Delta\theta \quad (2.26)$$

We denote the second order derivative, or Hessian matrix, $\frac{\partial^2\mathcal{L}}{\partial\theta^2}$ by \mathbf{H} to improve readability. The solution of Equation (2.26), provided below, is obtained by forcing the first derivative with respect to $\Delta\theta$ of $\mathcal{L}(\theta) + \nabla\mathcal{L}\Delta\theta + \Delta\theta^T\mathbf{H}\Delta\theta$ to be 0.

$$\Delta\theta = -\mathbf{H}^{-1}\nabla\mathcal{L}^T \quad (2.27)$$

One important issue with second order methods is that the size of the Hessian grows with $\Theta(n_\theta^2)$ (quadratic in n_θ). For neural networks, this quickly becomes large enough that it is problematic to store the matrix in memory, let alone to invert it.

To address this problem, one has to rely on approximating this quantity rather

than computing it exactly. One popular approach is to approximate the Hessian by a diagonal matrix which reduces the memory consumption from $\Theta(n_\theta^2)$ to $\Theta(n_\theta)$ (Bishop, 2006). The inversion of this diagonal matrix is also trivial. However by employing this approximation we lose all information about how different directions interact with each other.

Alternatively one can use a truncated Newton approach to invert the matrix, as described, for example, in Nocedal and Wright (2006) (which provide a good introduction to these optimization techniques), a book that provides a good introduction to the field of numerical optimization. In this approach, the desired step is obtained by solving the linear equation $\mathbf{H}\mathbf{x} = -\nabla\mathcal{L}^T$ for the variable \mathbf{x} . Yet another approach is to use a Broyden-Fletcher-Goldfarb-Shanno (BFGS) approximation of the inverse described in the same book (Nocedal and Wright, 2006). The underlying assumption is that the Hessian changes smoothly (if at all) when going from one step to the other, and therefore the inverse of the Hessian is approximated using also computations carried out at the previous time steps. Some of these approaches will be explained in more depth in Section 4.1.

As for the first order method, the second order approximation of the function \mathcal{L} is only true locally, around $\theta_{[t]}$. To address this issue, one can multiply the step $\Delta\theta$ by some step size, or learning rate λ . Typically the value of λ in this case is given by a line search.

Another approach to enforce the step size to be reasonable is via a *trust region* (Nocedal and Wright, 2006). That is we convert Equation (2.26) into a constrained optimization where we ask the norm of $\Delta\theta$ to be smaller than some maximal value. Solving the constraint regularization one gets the following formula for $\Delta\theta$:

$$\Delta\theta = -(\mathbf{H} + \alpha\mathbf{I})^{-1} \nabla\mathcal{L}^T \quad (2.28)$$

The matrix \mathbf{I} is the identity matrix, and α is usually referred to as the damping coefficient and controls the radius of the trust region.

2.4.3 Using the structure of the parameter manifold

Yet another family of gradient based optimization techniques are those that take into account the structure of the underlying parameter manifold.

This line of research can be traced back to Amari’s work on information geom-

etry (Amari, 1985) and its application to various neural networks (Amari et al., 1992; Amari, 1997), though a more in-depth introduction can be found in Amari (1998); Park et al. (2000); Arnold et al. (2011). The algorithm has also been successfully applied in the reinforcement-learning community (Kakade, 2001; Peters and Schaal, 2008) and for stochastic search (Sun et al., 2009).

We know that each parameter value θ identifies a specific member f_θ of our family of parametric models \mathcal{F} . However the Euclidean distance between two parameters θ_1 and θ_2 is not necessarily reflected in the “distance” between the functions they realize, f_{θ_1} and f_{θ_2} . Leaving aside the issue of how one defines a distance between functions, it is intuitively easy to see that while θ_1 and θ_2 are very different (large Euclidean distance) the function they induce could have almost identical behaviour, or the other way around, small changes in the parameter can result in widely different behaviours in the model. For example, an extreme case would be when some component of the parameter vectors θ_1 and θ_2 is not connected to the output. In this instance we can artificially increase the distance between θ_1 and θ_2 as much as we want, with the two settings realizing the exact same function.

It is therefore fruitful to move in the functional manifold induced by the mapping from θ to f_θ rather than the parameter space and natural gradient descent attempts to do exactly this. To formalize this mathematically, we rely on the Riemannian structure of the underlying manifold. That is, the space is smooth and equipped with an inner product on the tangential space at each point p of the manifold. We can therefore understand the geometry of the space locally to some point p by looking at the tangential space and specifically at the metric of the tangential space that defines the inner product (and distances) between vectors in this tangential space.

We can move on the manifold by relying on the tangential space at each point to define the local geometry of the manifold and indicate the right descent direction. If we take small steps, we can assume that the tangential space approximates well the manifold locally, therefore the direction we picked makes sense. Under these assumptions, moving along the manifold is equivalent to correcting the Euclidean gradient by multiplying it with the inverse of the Riemannian metric at position $\theta_{[t]}$ in order to account for the geometry of the tangential space.

In order to do so, we need to first define this Riemannian structure for which it is sufficient to define the *distance measure* between the functions in \mathcal{F} . In Amari

(1998) this is done by relying on the probabilistic interpretation of our model (specifically the model can be seen as modeling the conditional distribution $p(\mathbf{t}|\mathbf{u})$). We know that *locally* the Kullback-Leibler divergence (given in the equation below) between two probability density functions behaves as a distance measure. This is true because the second order Taylor expansion of the KL, when comparing p_θ and $p_{\theta+\Delta\theta}$, has all its terms zero except the second order one. The second order term (which gives the metric) has the form $\Delta\theta^T \mathbf{F} \Delta\theta$, where \mathbf{F} is symmetric, making the whole approximation of the KL symmetric. Since the second order approximation is reliably close to θ , then it corresponds to the KL divergence.

$$D_{KL}(P||Q) = \int_{-\inf}^{\inf} \ln \left(\frac{p(x)}{q(x)} \right) p(x) dx \quad (2.29)$$

Equation (2.29) describes the KL divergence between two probability density functions p and q . It measures the amount of information lost when one of the probabilities is used to approximate the other. The KL divergence as a distance measure is meaningful as it describes how similar two probability density functions are. From our perspective, in addition to this it is also useful because it looks at the behaviour of the probability density function rather than its parametrization (different re-parametrizations result in the same value). Therefore it is a distance between the functions, and not their parameters. We identify the distance between two functions f_{θ_1} and f_{θ_2} by the KL-divergence between their corresponding probabilistic interpretations p_{θ_1} and p_{θ_2} . Note that the tangential space approximates the manifold only locally, so the fact that the KL is not globally a distance measure is of no importance.

The second order term of the Taylor expansion of the KL-divergence results in the well known Fisher Information Matrix (FIM), defined below.

$$\mathbf{F} = \mathbf{E}_{\mathbf{t} \sim p_\theta(\mathbf{t}|\mathbf{u})} \left[\left(\frac{\partial \log p_\theta}{\partial \theta} \right)^T \frac{\partial \log p_\theta}{\partial \theta} \right] \quad (2.30)$$

Putting everything together we get the following step, which is very similar to equation (2.27) for the Newton method, except that the Hessian is now replaced by the FIM matrix:

$$\Delta\theta = -\mathbf{F}^{-1} \nabla \mathcal{L}^T \quad (2.31)$$

The same issues pointed out in the previous section about the Hessian matrix apply here as well. The metric \mathbf{F} can quickly become very large (it also scales with n_θ^2) and inverting it would be prohibitive. However, the same techniques as those used for Hessian can be used for the FIM as well.

3

The importance of depth for neural networks

The first question we will attempt to address is: *Why do we want to use deep networks?* This is an important question, as shallow models can have very useful properties, as for example defining a convex problem. Such properties can be used to theoretical analyse the behaviour of this model and provide guaranties regarding convergence time and so on. Deep models, on the other hand, are non-convex and hard to analyse. As a consequence most approaches come with no guarantee.

One can easily allow oneself to be convinced of the utility of deep networks by examining the wealth of existing empirical evidence. Deep learning approaches obtained state of the art on various tasks, like image classification (Krizhevsky et al., 2012; Zeiler and Fergus, 2013; Goodfellow et al., 2014), speech (Dahl et al., 2010; Hinton et al., 2012; Graves et al., 2013), language modelling (Pascanu et al., 2014; Mikolov et al., 2011), etc.

There are also intuitive arguments for depth. In Bengio (2009), for example, it is argued that depth allows the construction of a hierarchy of features, where the features on the higher layers are more complex, being constructed from the ones on the layers below. This is a *divide et impera* strategy, which can efficiently compute exponentially in terms of depth, or more complex features. In fact, this strategy is employed successfully in many core computer science algorithms, from sorting (e.g. merge sort) to computing the discrete Fourier transform, dynamic programming approaches, etc. From this perspective, the usefulness of depth relies on the fact that deep models can be exponentially more efficient at representing certain families of highly structured functions compared to shallow models.

Evidence of such behaviour in deep networks has been provided in Zeiler and Fergus (2013); Lee et al. (2009), where the internal behaviour of units in higher layers of a deep convolutional network¹ was empirically explored. These works show units in higher layers responding to more complex patterns than units in the

1. Models from who the hidden layer i was obtain by convolving the output of the previous layer with several kernels that are learnt; the kernels are part of the parameters of the model

lower layers. The brain is also believed to have a deep structure (Serre et al., 2007), with at least 5 and up to 10 layers for the visual system alone.

Additionally, this exponential efficiency has been analysed theoretically on particular families of deep models. For example, Håstad (1986) explores networks of logical gates, Håstad and Goldmann (1991); Hajnal et al. (1993) looks at networks of threshold units, and, in Bengio and Delalleau (2011), certain families of polynomials are represented more efficiently by deep sum and product networks (Poon and Domingos, 2011). The representational power of generative models based on Boltzmann machines has also been researched (Montúfar et al., 2011; Martens et al., 2013), deep belief networks are analyzed in Sutskever and Hinton (2008); Le Roux and Bengio (2010); Montúfar and Ay (2011), and, in Montúfar and Morton (2012), mixture of experts models are compared to products of experts.

Our work falls into the last category of arguments, namely it is a theoretical treatment of this fundamental question. In this chapter we will analyze deep models that rely on a piecewise linear activation function such as the rectifier function. Compared to other works, the models analyzed here have been successfully used in practice to obtain state of art results on hard tasks. Furthermore, the rectifier activation function is becoming the *de facto* activation of choice for deep models. We also provide an intuitive geometrical understanding of deep neural networks that can be used to further investigate how the model behaves.

We describe one basic mechanism employed by deep models (regardless of the activation function) to gain this exponential efficiency. It consists in *identifying* different regions of the input space. We call two regions, A and B , *identified* by some function f (e.g. the output of some hidden unit) if the function f has the same response on both A and B . This mechanism (that will be properly introduced later on) can also reveal some insight into the structure of the family of functions that can indeed be represented efficiently by deep models. Namely identifying regions is only useful if the functions we want to model are highly symmetric, or, in other words, if they are invariant to certain transformations. If that is the case we can rely on these invariances or symmetries to identify regions on which we expect to have the same or very similar behaviour.

The organization of the subsequent sections is as follows. In Section 3.1 we describe the specific problem that we want to analyze. In Section 3.2 we explore shallow models, while Section 3.3 introduces the main contribution of this chapter.

Finally, Section 3.6 provides some specific conclusions on this topic.

The content of this chapter overlaps with two publications (Pascanu, Montufar, and Bengio, 2014; Montufar, Pascanu, Cho, and Bengio, 2014). In writing this chapter I *borrowed* proofs, figures and paragraphs from these works. Please see Section 1.1 for a detailed description of my personal contribution to each of the two papers.

3.1 Preliminaries

The rectifier activation function (Glorot et al., 2011a; Nair and Hinton, 2010), defined below, has become a popular choice for neural networks.

$$\text{rect}(x) = \max\{0, x\} = x \cdot \mathbb{I}_{x>0}(x) = \begin{cases} x & , \text{ iff } x > 0 \\ 0 & , \text{ otherwise} \end{cases} , \quad (3.1)$$

where $x \in \mathbb{R}$ and \mathbb{I} is the indicator function:

$$\mathbb{I}_{x>b}(x) = \begin{cases} 1 & , \text{ iff } x > b \\ 0 & , \text{ otherwise} \end{cases} . \quad (3.2)$$

The reason for its success is believed to be related to the optimization issues surrounding neural networks. Compared to sigmoid or tanh activation function, for which the gradient can become exponentially small as the norm of the input to the function increases, for rectifier units the norm of the gradient is 1 regardless on the norm of the input (as long as it is positive). This makes the gradients through the rectifier networks better behaved, leading to good solutions even for deep models that have not been pretrained¹.

One can view each hidden unit in a single layer MLP with rectifiers as dividing the input space into two, where the boundary between the two regions is a hyperplane. On one side of the hyperplane we have the unit being *active*, which means that it has a positive value, while on the other side of the hyperplane the

1. Pretraining is a technique originally used to improve the solution found for a deep neural network, where the network is trained into stages. In the first stage the network is trained to reconstruct its input (or model the input distribution), while in the second stage the network is trained to predict the desired target.

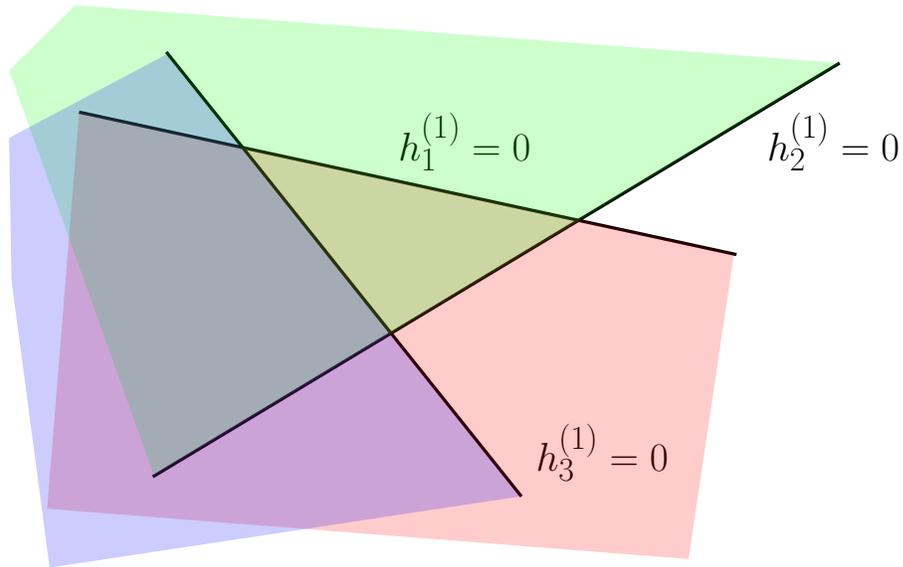


Figure 3.1: Illustration of how the input space (in this case the plane) is divided by the first hidden layer (of only three hidden units) of a feedforward network with rectifier units. The figure is best seen in colour.

unit is *inactive* or *dead*, which is equivalent to it being equal to 0. If we consider all the hidden units of the hidden layer, then they define a partition of the space into regions by using several hyperplanes as shown in Figure 3.1. Each hyperplane corresponds to one of the hidden units.

By using a linear output layer on top of this hidden layer, we are learning, for each input region, a different linear map. This means that if we divide the space into sufficiently many small such linear regions, then we can reliably learn any continuous and differentiable function f .

Deep models with rectifiers, being concatenations of piecewise linear functions, are themselves piecewise linear functions. Therefore, in a similar fashion, they will also divide the input space into many linear regions. We will argue that a deep model can split the space into exponentially more regions than its shallow counterpart with the same number of units/parameters. However, due to the shared parameters between the different linear pieces of the deep model, the mappings learnt for each input region will not be independent of each other.

We regard this dependency between the different mappings as an advantage of deep models rather than a shortcoming. If deep models, beside producing exponentially more linear regions, would also be able to learn independent linear maps

for each of these regions, then they would severely over-fit any task.

This constraint between the linear maps behaves like a regularizer. It is a constraint on the kind of functions deep models can represent, and hence it puts a prior on how the model behaves on unseen data, a prior that leads to better generalization error. We will return to this observation later on, when we will try to characterize the family of functions these deep models can represent. But, as long as the true solution of some task belongs to this restricted family of functions, deep models will be better suited at solving this specific task (at least in theory).

The number of linear regions behaves as a proxy for the flexibility of the trained model, shallow or deep. The more linear regions we have, the better we can approximate some specific set of functions. For example, to perfectly approximate some quadratic function, one would need infinitely many such linear regions. Assuming the relationship between the regions can be captured in the shared parameters of the deep model, the deep model would be much more efficient at representing this quadratic if, for the same number of units, it partitions the space into exponentially more regions.

We start by reformulating our intuition that deep models can be (exponentially) more efficient, by saying that deep models are piecewise linear functions that can have (exponentially) more linear pieces than shallow models when the number of hidden units (or parameters) is kept constant. Figure 3.2 exemplifies this idea. We compare two models (a single layer MLP vs a two layer one) with the same total number of hidden units. The models are learning to separate two classes. The boundary between these classes is given by a sinusoidal shape. One can see that the deep model uses more segments to construct this boundary and does obtain fewer errors.

Although in the above example we used the same number of hidden units to control for the complexity of the deep versus shallow model, a more popular choice is to use the number of parameters. Our reason for relying on the number of hidden units is that it directly corresponds, for the shallow model, to the number of hyperplanes that it can use to partition the space. However, for fairness, we will show that our results hold also when one controls for the number of parameters rather than the number of hidden units.

Finally, for simplicity, we assume that the output layer is linear. This is not a restriction *per se*. As long as the output activation function $\sigma^{(out)}$ is not itself

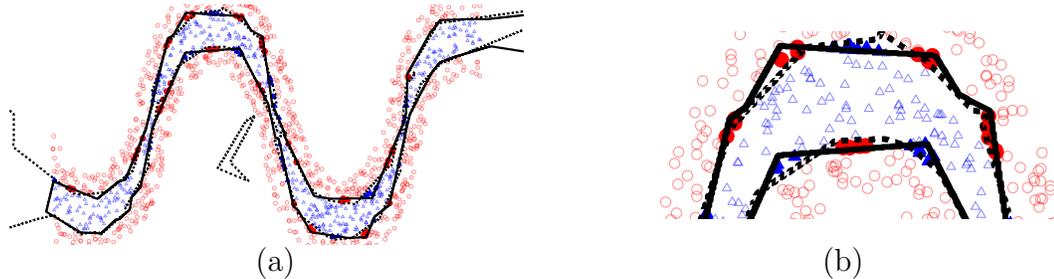


Figure 3.2: Illustration of the learnt boundary by a single layer MLP with 20 rectifier hidden units (solid line), versus a deep (2 layer) model with 10 rectifier units on each layer (dashed line). In (a) we show the whole dataset, (b) provides a zoom into a segment of the boundary. Note how the deep model relies on more linear segments to construct the boundary, specifically see (b), and hence performs better. Filled markers indicate errors made by the shallow model.

parametrized and there exists some continuous function $\sigma^{(out)^{-1}}$ such that

$$\sigma^{(out)} \circ \sigma^{(out)^{-1}}(\mathbf{y}) = \mathbf{y}$$

for all $\mathbf{y} \in \mathbb{T}$, our proof holds with simple adaptations. Our argument is simple. One can project the targets through $\sigma^{(out)^{-1}}$ and compare a linear-output shallow model versus a linear-output deep model, both trained to predict these transformed targets. This comparison will give an indirect measure of how the $\sigma^{(out)}$ shallow model compares to its deep counterpart.

Furthermore, the number of linear output units also does not affect the maximal number of regions into which some model can split the space. This follows from the linearity of the output layer. In order to prove this statement, let us first introduce some notation (taken from [Pascanu, Montufar, and Bengio \(2014\)](#)).

Definition 3. Given a vector of natural numbers $\mathbf{n} = (U, n_1, \dots, n_L)$, we denote by $\mathcal{F}_{\mathbf{n}}$ the set of all functions $\mathbb{R}^U \rightarrow \mathbb{R}^{n_L}$ that can be computed by a rectifier feedforward network with U inputs and n_l rectifier units in layer l for $l \in [L]$. The output layer of this feedforward network is linear. The elements of $\mathcal{F}_{\mathbf{n}}$ are continuous piecewise linear functions.

We denote by $\mathcal{R}(\mathbf{n})$ the maximum of the number of regions of linearity or *response regions* over all functions from $\mathcal{F}_{\mathbf{n}}$. For clarity, given a function $f: \mathbb{R}^U \rightarrow \mathbb{R}^{n_L}$, a connected open subset $R \subseteq \mathbb{R}^U$ is called a *region of linearity* or *linear region* or *response region* of f if the restriction $f|_R$ is a linear function and for any open set $\tilde{R} \supsetneq R$ the restriction $f|_{\tilde{R}}$ is not a linear function. In the next sections we will

compute bounds on $\mathcal{R}(\mathbf{n})$ for different choices of \mathbf{n} . We are especially interested in the comparison of shallow networks with one single very wide hidden layer and deep networks with many narrow hidden layers.

The next lemma states that a piecewise linear function $f = (f_i)_{i \in [k]}$ has as many regions of linearity as there are distinct intersections of regions of linearity of the coordinates f_i .

Lemma 1. *Consider a width k layer of rectifier units. Let $R^i = \{R_1^i, \dots, R_{N_i}^i\}$ be the regions of linearity of the function $f_i: \mathbb{R}^U \rightarrow \mathbb{R}$ computed by the i -th unit, for all $i \in [k]$. Then the regions of linearity of the function $f = (f_i)_{i \in [k]}: \mathbb{R}^U \rightarrow \mathbb{R}^k$ computed by the rectifier layer are the elements of the set $\{R_{j_1, \dots, j_k} = R_{j_1}^1 \cap \dots \cap R_{j_k}^k\}_{(j_1, \dots, j_k) \in [N_1] \times \dots \times [N_k]}$.*

Proof. A function $f = (f_1, \dots, f_k): \mathbb{R}^n \rightarrow \mathbb{R}^k$ is linear iff all its coordinates f_1, \dots, f_k are. \square

In regard to the number of regions of linearity of the functions represented by rectifier networks, the number of output dimensions, i.e., the number of linear output units, is irrelevant. This is the statement of the next lemma.

Lemma 2. *The number of (linear) output units of a rectifier feedforward network does not affect the maximal number of regions of linearity that it can realize.*

Proof. Let $f: \mathbb{R}^U \rightarrow \mathbb{R}^k$ be the map of inputs to activations in the last hidden layer of a deep feedforward rectifier model. Let $h = g \circ f$ be the map of inputs to activations of the output units, given by composition of f with the linear output layer, $h(\mathbf{x}) = \mathbf{W}^{(\text{out})} f(\mathbf{x}) + \mathbf{b}^{(\text{out})}$. If the row span of $\mathbf{W}^{(\text{out})}$ is not orthogonal to any difference of gradients of neighbouring regions of linearity of f , then g captures all discontinuities of ∇f . In this case both functions f and h have the same number of regions of linearity.

If the number of regions of f is finite, then the number of differences of gradients is finite and there is a vector outside the union of their orthogonal spaces. Hence a matrix with a single row (a single output unit) suffices to capture all transitions between different regions of linearity of f . \square

3.2 Single hidden layer feedforward model

To justify our claims we have to start by first analysing the shallow models. As it turns out, the property we are trying to measure, the number of linear regions in the input space, is a well defined quantity in computational geometry. Specifically, this quantity is given by Zaslavsky’s theorem (Zaslavsky, 1975, Theorem A).

Let us further formalize the question and its specific answer. As we outlined in the previous section, a single layer model divides the input space using hyperplanes. Each hyperplane corresponds to a hidden unit and divides the space into the region where this given unit is active, versus the region where it is inactive or dead.

In geometry, a set of n hyperplanes in a U -dimensional Euclidean space is called an U -dimensional *hyperplane arrangement* \mathcal{A} , and the hidden units of a single layer MLP define exactly this geometrical object.

Such an arrangement is said to be in general position if its topology is stable under random small perturbations. For the two dimensional case, this means that any pair of lines in the arrangement intersect at a distinct point. When adding noise to the slope of these lines, any pair will still intersect at some distinct point. If three lines intersect at a single point, however, under noise, it will be very unlikely for them to still do so.

Generalized to higher dimensional input spaces, the property can be stated as saying that the intersection of any number p of hyperplanes from the arrangement is either a $(U - p)$ sub-dimensional space, if $p \leq U$, or, otherwise, it is the empty set. We are interested in hyperplanes arrangements that are in general position as they split the space in the maximal number of regions. This follows from Zaslavsky’s general theorem. Using this notation, the problem we want to solve is:

Problem 1. Into maximally how many regions is the space \mathbb{R}^U split by n hyperplanes?

For arrangements in general position, Zaslavsky’s theorem can be stated in the following way Stanley (see 2004, Proposition 2.4).

Proposition 1. *Let \mathcal{A} be an arrangement of m hyperplanes in general position in \mathbb{R}^U . Then*

$$r(\mathcal{A}) = \sum_{s=0}^U \binom{m}{s},$$

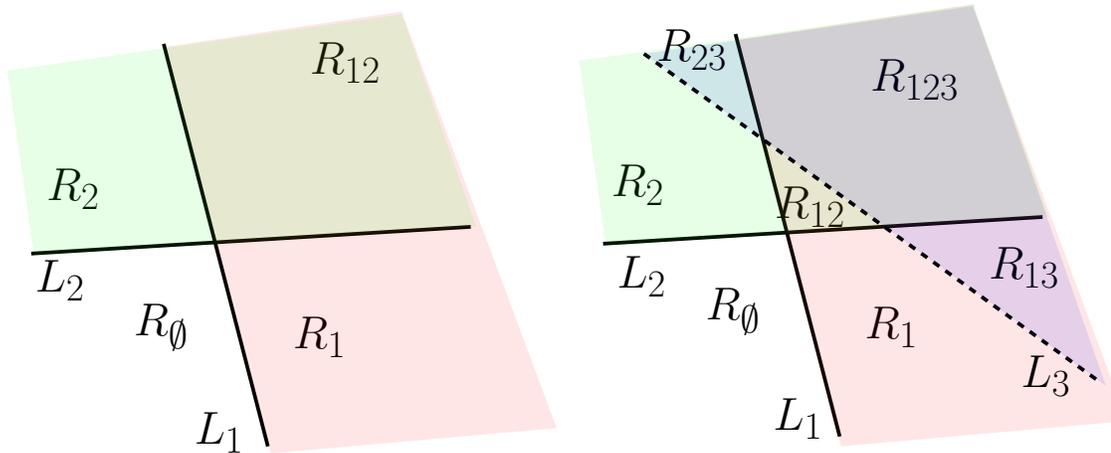


Figure 3.3: Induction step of the hyperplane sweep method for counting the regions of line arrangements in the plane.

where $r(\mathcal{A})$ is the number of regions it constructs.

In particular, the number of regions of a 2-dimensional arrangement \mathcal{A}_m of m lines in general position is equal to

$$r(\mathcal{A}_m) = \binom{m}{2} + m + 1. \quad (3.3)$$

For the purpose of illustration, we sketch a proof of Equation (3.3) using the *sweep hyperplane* method¹. We proceed by induction over the number of lines m .

Base case $m = 0$. It is obvious that in this case there is a single region, corresponding to the entire plane. Therefore, $r(\mathcal{A}_0) = 1$.

Induction step. Assume that for m lines the number of regions is $r(\mathcal{A}_m) = \binom{m}{2} + m + 1$, and add a new line L_{m+1} to the arrangement. Since we assumed the lines are in general position, L_{m+1} intersects each of the existing lines L_k at a different point. Figure 3.3 depicts the situation for $m = 2$.

The m intersection points split the line L_{m+1} into $m + 1$ segments. Each of these segments splits a region of \mathcal{A}_m in two pieces. Therefore, by adding the line L_{m+1} we get $m + 1$ new regions. In Figure 3.3 the two intersection points result in

1. This proof is a well known result in computational geometry and does not belong to me. See, for example, Stanley (2004).

three segments that split each of the regions R_1, R_{01}, R_0 in two. Hence

$$\begin{aligned} r(\mathcal{A}_{m+1}) &= r(\mathcal{A}_m) + m + 1 = \frac{m(m-1)}{2} + m + 1 + m + 1 = \frac{m(m+1)}{2} + (m+1) + 1 \\ &= \binom{m+1}{2} + (m+1) + 1. \end{aligned}$$

For the number of response regions of MLPs with one single hidden layer we obtain the following.

Proposition 2. *The regions of linearity of a function in the model $\mathcal{F}_{(U, n_1, O)}$ with U inputs and n_1 hidden units are given by the regions of an arrangement \mathcal{A} of n_1 hyperplanes in U -dimensional space. The maximal number of regions of such an arrangement is $r(\mathcal{A}) = \mathcal{R}(U, n_1, O) = \sum_{j=0}^U \binom{n_1}{j}$.*

Proof. This is a consequence of Lemma 1 and the maximal number of regions is produced by an U -dimensional arrangement of n_1 hyperplanes in general position, which is given in Proposition 1. \square

3.3 Deep networks

We can now turn our attention to deep rectifier MLPs. If we look at some layer of this deep MLP, say the second (hidden) layer, it is easy to see that, while each hidden unit still has two possible modes, active or inactive, in the input space the boundary between these modes is not a hyperplane anymore.

Let us consider a simple two-dimensional example. Assume we have three hidden units on the first layer, $h_1^{(1)}, h_2^{(1)}$ and $h_3^{(1)}$. We will examine the behaviour of a unit $h_1^{(2)}$ on the second layer, that receives input from three units $h_1^{(1)}, h_2^{(1)}$ and $h_3^{(1)}$. The pre-activation value of $h_1^{(2)}$ is given below:

$$\begin{aligned}
\hat{h}_1^{(2)} &= w_1 h_1^{(1)} + w_2 h_2^{(2)} + w_3 h_3^{(3)} + b & (3.4) \\
h_1^{(1)} &= \text{rect}(u_1 x + u_2 y + c) \\
h_2^{(1)} &= \text{rect}(v_1 x + v_2 y + d) \\
h_3^{(1)} &= \text{rect}(t_1 x + t_2 y + e)
\end{aligned}$$

where $w_1, w_2, w_3, u_1, u_2, v_1, v_2, t_1, t_2, b, c, d, e \in \mathbb{R}$ are the parameters of the model and $\hat{h}_1^{(2)}$ indicates the value of the hidden unit $h_1^{(2)}$ before applying the activation function of the unit. As for the single layer model, in order to define the boundary that divides the region of the input where this unit is active from the region where it is inactive, we need to solve the equation $\hat{h}_1^{(2)} = 0$. We can do this by explicitly solving the equation on each branch of $h_1^{(1)}, h_2^{(1)}$ and $h_3^{(1)}$:

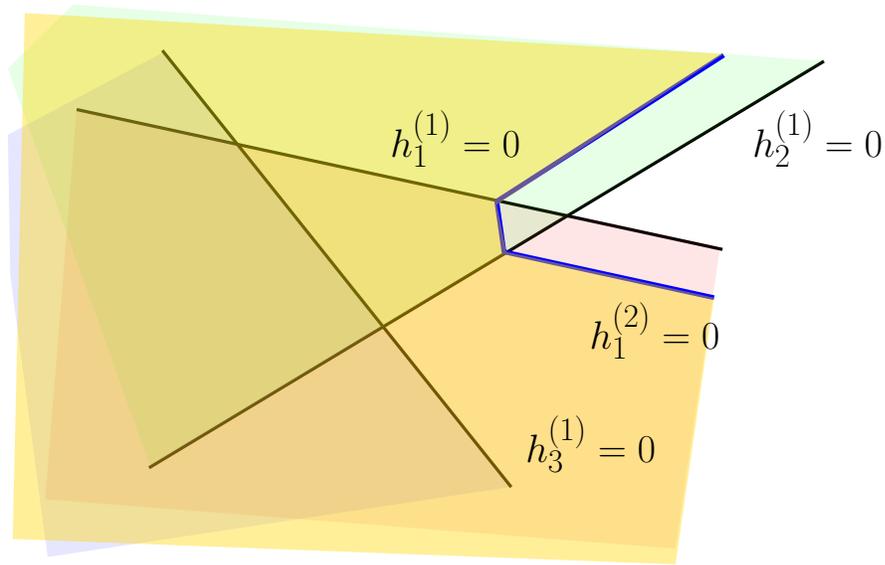


Figure 3.4: The piecewise linear boundary defined by a hidden unit on the second layer of a rectifier MLP with only 3 hidden units on the first layer. This picture is best seen in color.

$$\hat{h}_1^{(2)} = \begin{cases} b & , \text{ iff all units are inactive} \\ w_1u_1x + w_1u_2y + (b + cw_1) & , \text{ iff } h_2^{(1)} \text{ and } h_3^{(1)} \text{ are inactive} \\ w_2v_1x + w_2v_2y + (b + dw_2) & , \text{ iff } h_1^{(1)} \text{ and } h_3^{(1)} \text{ are inactive} \\ w_3t_1x + w_3t_2y + (b + ew_3) & , \text{ iff } h_1^{(1)} \text{ and } h_2^{(1)} \text{ are inactive} \\ (w_1u_1 + w_2v_1)x + (w_1u_2 + w_2v_2)y \\ \quad + (b + cw_1 + dw_2) & , \text{ iff } h_3^{(1)} \text{ is inactive} \\ (w_1u_1 + w_3t_1)x + (w_1u_2 + w_3t_2)y \\ \quad + (b + cw_1 + ew_3) & , \text{ iff } h_2^{(1)} \text{ is inactive} \\ (w_2v_1 + w_3t_1)x + (w_2v_2 + w_3t_2)y \\ \quad + (b + dw_2 + ew_3) & , \text{ iff } h_1^{(1)} \text{ is inactive} \\ (w_1u_1 + w_2v_1 + w_3t_1)x \\ \quad + (w_1u_2 + w_2v_2 + w_3t_2)y \\ \quad + (b + cw_1 + dw_2 + ew_3) & , \text{ iff all units are active} \end{cases} \quad (3.5)$$

This is now a piecewise linear function. Figure 3.4 depicts a instantiation of $\hat{h}_1^{(2)} = 0$, while in Figure 3.5 we can see the regions formed by two units on a higher layer in a similar setup.

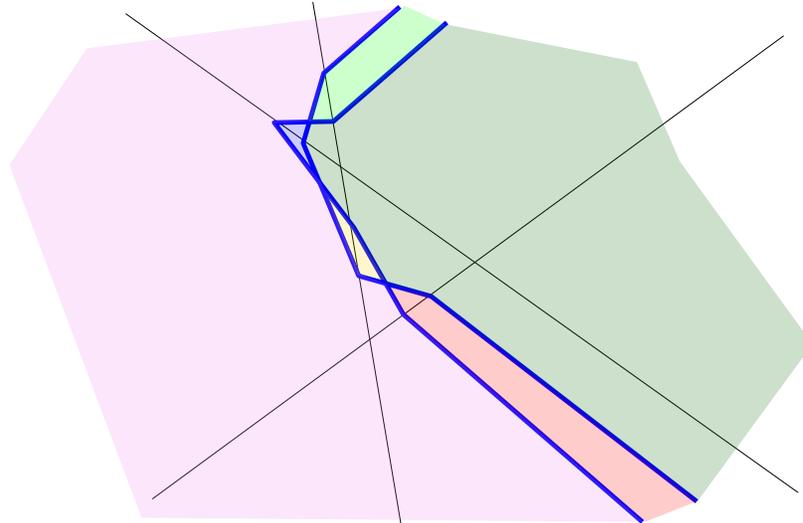


Figure 3.5: Depiction of two piecewise linear functions that intersect in two different points, splitting the space into more than 4 regions (which is what you would get from an arrangement of two lines).

From the *sweep hyperplane method*, employed for computing the number of

linear regions of a hyperplane arrangement, we can see that the number of distinct intersection points between the boundaries of the different hidden units is related with the final number of linear regions. In fact it is not hard to imagine that it would be possible to generalize this method to piecewise linear boundaries, and, even in this case, the more distinct intersection points we have the more regions we will end up with.

This observation is advantageous for deep models, as two distinct piecewise linear functions can intersect in more than a single point, therefore resulting in more intersection points. Figure 3.5 illustrates this behaviour and, fundamentally, this is the reason why we can construct (exponentially) more regions using deep models compared to shallow ones.

3.3.1 Understanding units in higher layers

The idea of assigning meaning to hidden units in a deep model is, in many instances, very useful. It can help understanding how the model *functions*, how it can be debugged, when and why it will underperform, and how it can be extended. However doing so is difficult.

Before proceeding to compare a deep model with a shallow model, in this section we will look at how this geometrical perspective on piecewise linear models can potentially be used to understand the behaviour of units in higher layers.

For a shallow model, we can usually look at the pre-activation value of each hidden unit as some sort of score. Since this value is given by a linear map, the score itself is just the cosine between the input vector \mathbf{x} and the corresponding weight row-vector of incoming connections \mathbf{W}_i : (the i -th row of the matrix \mathbf{W}).

This makes the interpretation of the unit easy. The weight vectors of the different units \mathbf{W}_i can be seen as templates, and each hidden unit on the first layer fires according to how well these templates are represented by the input example \mathbf{x} . In the specific case of images, we usually refer to the vectors \mathbf{W}_i as filters. They live in the same space as the original input image \mathbf{x} and therefore they can be visualized, providing some intuition to what kind of patterns each unit responds to. Usually, one expects or hopes these filters to look like Gabor filters (which is the typical response of neurons in the V1 region of the brain).

If we use a piecewise linear activation function for a deep model, as argued

before, we get that any hidden unit (regardless of the layer) is also a piecewise linear function. This comes from the simple fact that a composition of piecewise linear functions is itself piecewise linear.

This observation can be key to provide, for units on higher layers, a similar interpretation as the one given to the units of the first layer. Any piecewise linear function is fully defined by the different linear pieces from which it is composed. Each piece is given by its domain, a region of the input space $R_i \subseteq \mathbb{R}^U$, and the linear map f_i that describes its behaviour on R_i . Because f_i is an affine map, it can be interpreted in the same way hidden units in a shallow model are. Namely, we can write f_i as:

$$\hat{h} = f_i(\mathbf{x}) = \mathbf{u}^T \mathbf{x} + c, \quad \mathbf{x} \in R_i,$$

with \mathbf{u}^T a row vector, $\mathbf{u}^T \in \mathbb{R}^U$. \hat{h} is the pre-activation value of some hidden unit of the single layer neural model. Then f_i computes the cosine between \mathbf{x} and \mathbf{u}^T . If \mathbf{x} can be interpreted as some image (say in case of a vision task), \mathbf{u}^T can also be interpreted an image and shows the pattern (template) to which the unit responds whenever $\mathbf{x} \in R_i$.

If $(R_1^{(h_j^{(k)})}, f_1^{(h_j^{(k)})}), \dots, (R_p^{(h_j^{(k)})}, f_p^{(h_j^{(k)})})$ describe the behaviour of some hidden unit $h_j^{(k)}$ on layer k , then, by the statement above, if we can obtain the different linear function $f_i^{(h_j^{(k)})}$, i.e., we can fully interpret $h_j^{(k)}$.

Given some input example \mathbf{x} from some arbitrary region of the input space, $\mathbf{x} \in R_i^{(h_j^{(k)})}$, then we can construct the corresponding linear map $f_i^{(h_j^{(k)})}$. Specifically the weight \mathbf{u}^T is described in the equation below, and a similar computation can be done for the bias.

$$\mathbf{u}^T = \mathbf{W}_{j:}^{(k)} \text{diag}(\mathbb{I}_{\mathbf{h}^{(k-1)} > 0}(\mathbf{x})) \mathbf{W}^{(k-1)} \dots \text{diag}(\mathbb{I}_{\mathbf{h}^{(1)} > 0}(\mathbf{x})) \mathbf{W}^{(1)}. \quad (3.6)$$

Equation (3.6) computes the linear map by keeping track of which linear piece is used for all piecewise linear functions (hidden units on the layers below) that compose the function describing the unit we are inspecting. The formula itself is specific to a deep rectifier MLP, though it can easily be adapted for other piecewise linear models, including convolutional networks with rectifiers or maxout activation.

In order to properly describe some hidden unit $h_j^{(k)}$, in principle, we would need to identify and visualize (or inspect) each of the linear segments. To do so, one has

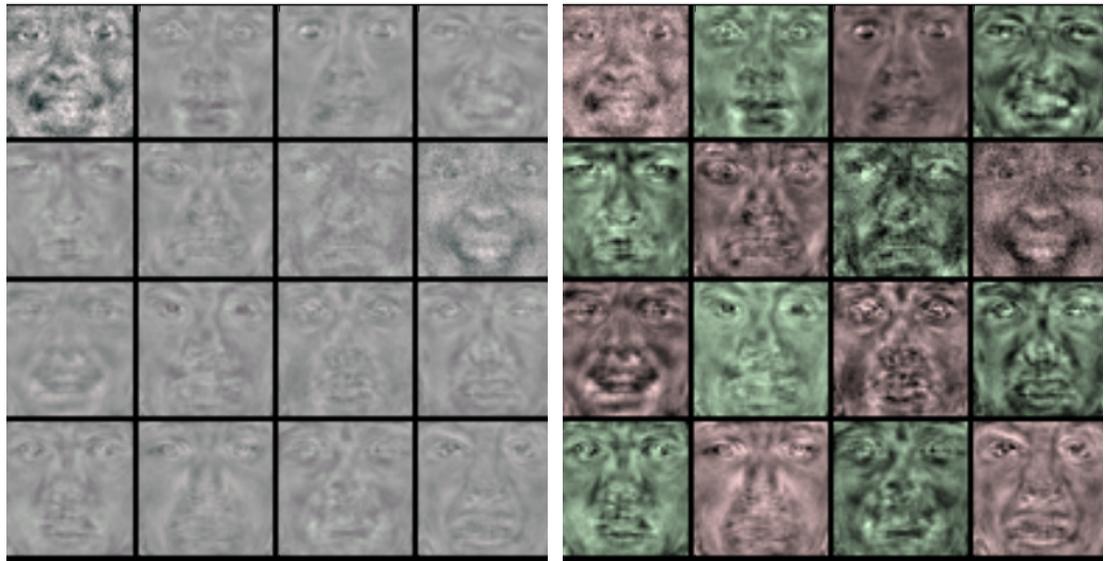
to traverse the whole input space and identify the linear function at each point. The set formed by these linear functions represents all the linear pieces of the function. This is not feasible. Instead we propose to use a set of points, as for instance the validation set, and identify all the linear responses that a hidden unit exhibits on this set of points. The hope is that what we will have is a sufficiently rich set of behaviours that should provide a better understanding of the hidden unit.

Figures 3.6, 3.7 and 3.8 show such a visualization applied to a rectifier MLP with 3 hidden layers. The first two hidden layers have 1000 hidden units, while the last one has only 100. The model was trained on the Toronto Faces Dataset (TFD) (Susskind et al., 2010). We randomly picked 20 units for the second and third layer and visualized the most interesting 4 units out of the subset (based on the maximal Euclidean distance between the different linear responses of each unit). To pick linear responses of the unit, we considered a K-means clustering of the linear responses obtained by going over the training examples. We clustered these responses in 4 classes and picked a representative of each class. For the first hidden layer each unit has a single response (linear piece), and we just visualize 16 different units. For the output layer, we visualize all 7 output units.

We trained the model using stochastic gradient descent. We used, as regularization, an L_2 penalty¹ with a coefficient of 10^{-3} , dropout on the first two hidden layers² (with a drop probability of 0.5) and we enforced the weights to have unit norm column-wise by projecting the weights after each SGD step. We used a learning rate of 0.1 and an output layer composed of sigmoid units (instead of a softmax layer which would be a more popular choice for classification). The purpose of these regularization schemes, and the sigmoid output layer, is to obtain cleaner and sharper filters. The model is trained on only a subset of the training data³ and achieves an error of 20.49% which is reasonable for this dataset and a non-convolutional model.

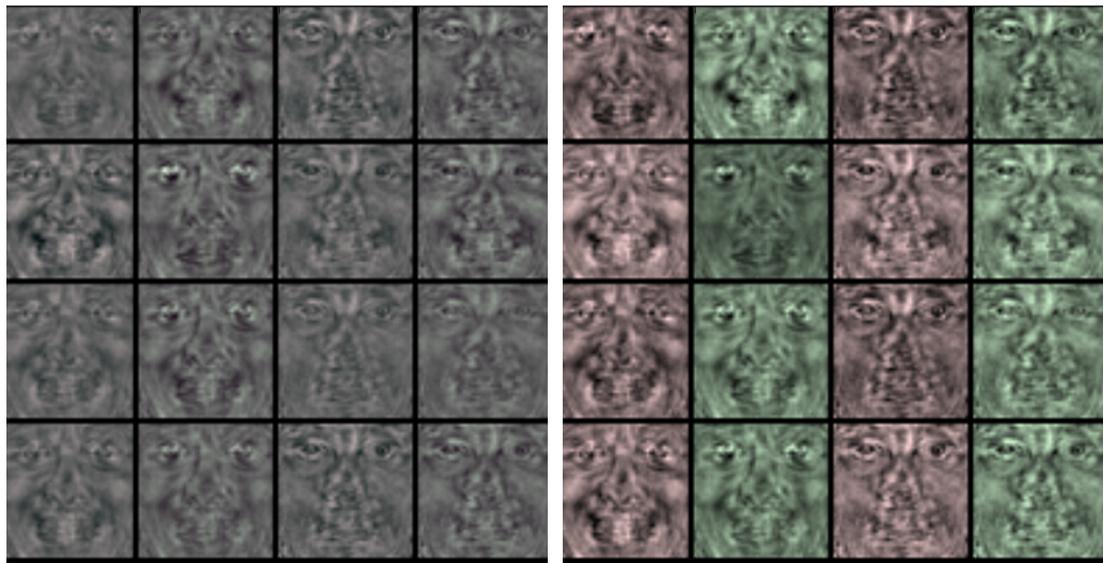
By looking at the resulting filters (figures 3.6, 3.7 and 3.8) we can see that higher order units become invariant to interesting transformations. See for example the linear responses of the second unit visualized for the second hidden layer. For the last visualized unit of the third layer we also provide differences between the

1. The L_2 penalty simply sums the square of the parameters of the model
2. The dropout regularization consist in simply dropping with some probability hidden units from the computation (Hinton et al., 2012)
3. The original dataset is split in 5 folds. We only rely on the first fold of this dataset



(a) 1st hidden layer filters

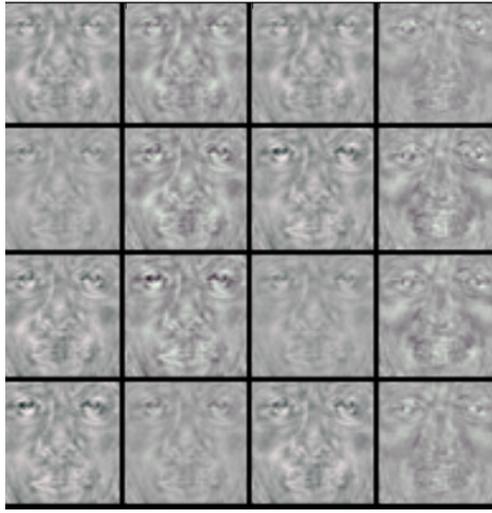
(b) 1st hidden layer normalized filters



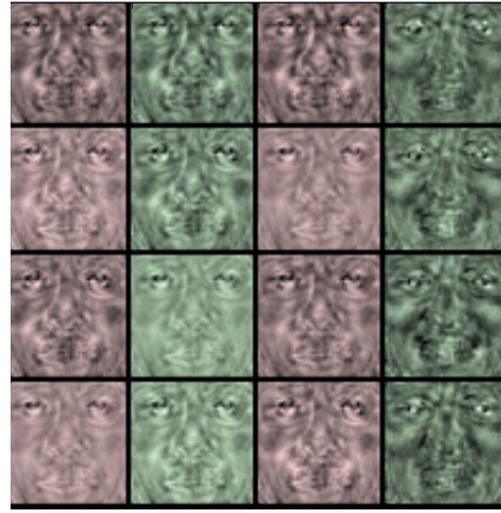
(c) 2nd hidden layer filters (each column represents 4 different linear responses of a given hidden unit)

(d) 2nd hidden layer normalized (each column represents 4 different linear responses of a given hidden unit)

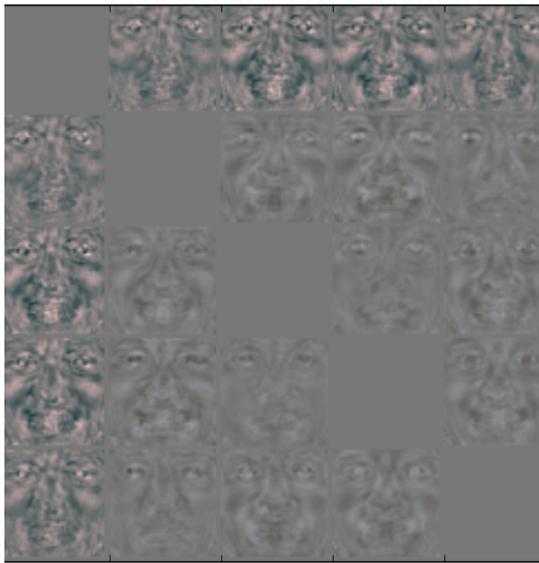
Figure 3.6: 1000-1000-100 hidden units rectifier MLP trained on TFD dataset. In (a) and (b) we visualize filters of the first layer (columns of the first weight matrix). Each unit has a single response, each visualized template of the first layer belongs to a different unit. In (b) each filter is normalized (showing only the direction, the shape of the filter). Colours are only used to improve visualization (the data contains no colour information). In (c) and (d) we visualize units on the second hidden layer. For each unit we visualize 4 different linear responses arranged in a column. As before, in (d) colour is only for visualization, and each filter is normalized independently.



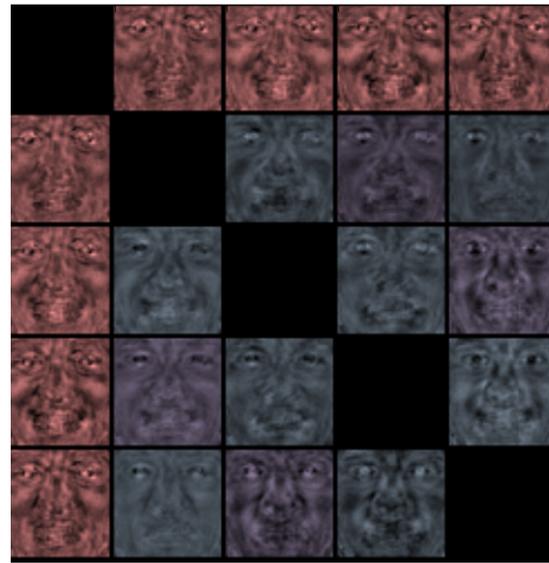
(a) 3rd hidden layer filters (each column represents 4 linear responses of a given unit)



(b) 3rd hidden layer normalized filters (each column represents 4 linear responses of a given unit)

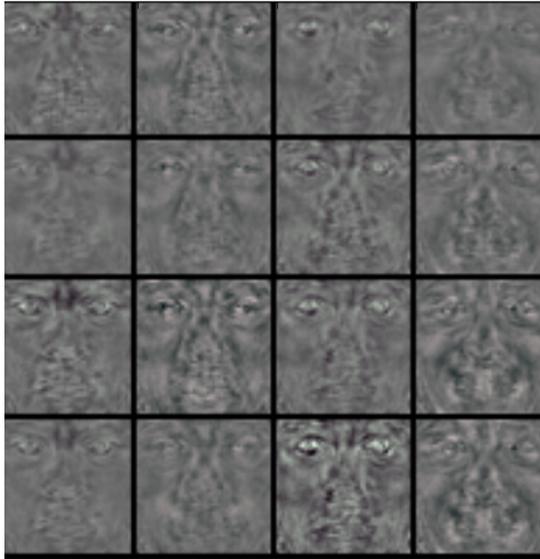


(c) Differences between the different linear responses of a unit in the 3rd hidden layer

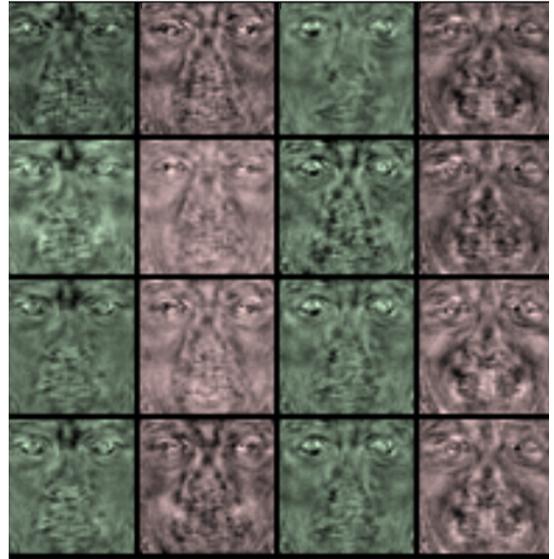


(d) Normalized differences between the different linear responses of a unit in the 3rd hidden layer

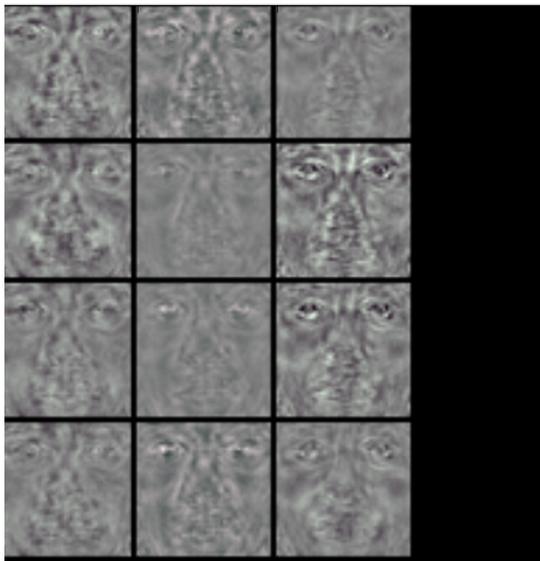
Figure 3.7: 1000-1000-100 hidden units rectifier MLP trained on TFD dataset. In (a) and (b) we show 4 different linear responses for units in the third hidden layer. (b) uses colours for better visualization, and normalizes each filter independently, showing the shape of the different filters. In (c) and (d), for the last visualized unit, we show the differences between the different linear responses. These deltas approximate the transformation that would transform one linear response into another. The unit is invariant to these transformations. Note that filters on the first column and first row are the actual linear responses. Each delta is the difference between a filter corresponding to the row minus the filter corresponding to the column.



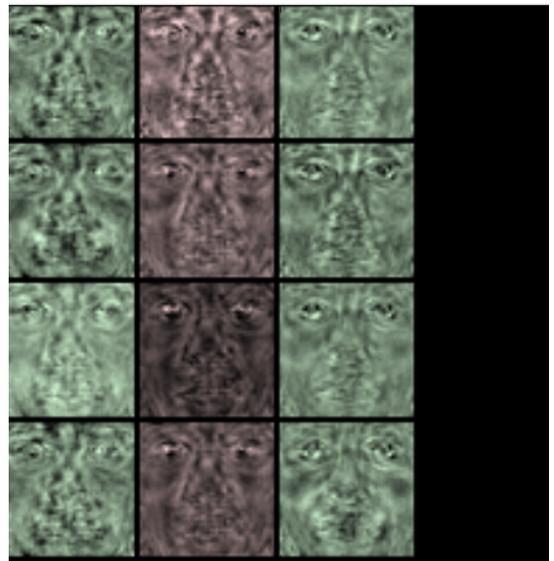
(a) Filters for the output layer, first 4 units (each column represents different linear responses for a given unit)



(b) Normalized filters for the output layer, first 4 units (each column represents different linear responses for a given unit)



(c) Filters for the output layer, last 4 units (each column represents different linear responses for a given unit)



(d) Normalized filters for the output layer, last 4 units (each column represents different linear responses for a given unit)

Figure 3.8: 1000-1000-100 hidden units rectifier MLP trained on TFD dataset. In (a) and (b) we show 4 different linear responses (a column) of the first 4 output units of the model. These represent the classes anger, disgust, fear and happy. In (b) we normalized each filter independently to only show the shape of the filter. Note the invariances that these linear pieces exhibit. In (c) and (d) we show linear responses (4 align in a column) for the last 3 output units, representing the classes sad, surprise, neutral.

different linear responses. These deltas represent a first order approximation of the transformation that takes you from one linear response to another. We argue that the unit learns, in some sense, to be invariant to these transformations by responding similarly to an input and its transformed counterpart.

[Zeiler and Fergus \(2013\)](#) also attempt to visualize the behaviour of units in the upper layer (in the case of a deep convolutional network with rectifiers). Our visualization approach is, to some extent, quite similar. One difference is that the approach we introduce here does not make any further assumptions besides the piecewise linear nature of the unit. This approach can be used for any piecewise linear activation function and any model structure.

Another important difference is in the justification of the method. [Zeiler and Fergus \(2013\)](#) attempts to invert (or approximate the inversion of) the computations carried on by the model and therefore it is a top down process. We provide a bottom up process, which avoids approximating the inverse (or reconstruction) function of the hidden unit. Our algorithm recovers the linear function that describes locally the piecewise linear functions and we use our understanding of linear functions to visualize this behaviour.

The resulting process is very similar in the computations that have to be carried out (except the order), with some minor differences. One example of such a difference is resolving rectifiers. Because [Zeiler and Fergus \(2013\)](#) attempts to reconstruct (or identify the part of input) that results in a maximal activation of a unit, it has to invert the rectifier activation (which by construction does not have an inverse). The inverse has to be approximated, and in this case, they claim that the rectifier itself is a good approximation. In contrast, we are attempting to identify the linear map that describes locally the behaviour of a unit in the deep model. Therefore we only need to identify which linear branches of the units below are used, and compose them. In case of the rectifier, if the unit is inactive when we evaluate the example \mathbf{x} that defines the input region, then it means that the corresponding column of weights is zeroed out and it does not contribute to the linear map represented by the unit. Otherwise the unit has a linear response. In other words, we end up using the mask provided by the forward computations, instead of applying the rectification function on the reconstruction.

3.3.2 A special class of deep models

In this section we want to show that deep models are in some sense more expressive than to shallow models. As before, we use the number of response regions as a measure of expressivity.

To answer the question of how deep models can construct, with the same number of hidden units (or parameters) a piecewise linear function with more linear regions, let us first look at a restricted class of deep models. In the subsequent sections we will generalize this class to any deep model, once we have identified the mechanism that allows one to obtain more linear regions.

We start by constructing a layer of $n = 4$ rectifiers f_1, f_2, f_3 and f_4 on top of the 2-dimensional Euclidean input, followed by a linear projection. We can construct the layer in such a way that it divides the input space into four ‘square’ cones; each of them corresponding to the inputs where two of the rectifier units are active.

We define the four rectifiers as:

$$\begin{aligned}f_1(\mathbf{x}) &= \max \{0, [+1, 0] \mathbf{x}\}, \\f_2(\mathbf{x}) &= \max \{0, [-1, 0] \mathbf{x}\}, \\f_3(\mathbf{x}) &= \max \{0, [0, +1] \mathbf{x}\}, \\f_4(\mathbf{x}) &= \max \{0, [0, -1] \mathbf{x}\},\end{aligned}$$

where $\mathbf{x} = [x_1, x_2]^\top \in \mathbb{R}^U$. By linearly projecting $\mathbf{f} = [f_1, f_2, f_3, f_4]^\top$, we can effectively mimic a layer with two absolute-value units g_1 and g_2 :

$$\begin{bmatrix} g_1(\mathbf{x}) \\ g_2(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ f_3(\mathbf{x}) \\ f_4(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \text{abs}(x_1) \\ \text{abs}(x_2) \end{bmatrix}. \quad (3.7)$$

Each absolute-value ‘unit’ g_i divides the input space into two regions along the i -th coordinate axis. The combination of g_1 and g_2 , the input space is divided into

four regions:

$$\begin{aligned}\mathcal{S}_1 &= \{(x_1, x_2) \mid x_1 \geq 0, x_2 \geq 0\} \\ \mathcal{S}_2 &= \{(x_1, x_2) \mid x_1 \geq 0, x_2 < 0\} \\ \mathcal{S}_3 &= \{(x_1, x_2) \mid x_1 < 0, x_2 \geq 0\} \\ \mathcal{S}_4 &= \{(x_1, x_2) \mid x_1 < 0, x_2 < 0\}.\end{aligned}$$

Additionally, the values of g_i 's are symmetric with respect to the i -th axes, meaning that for any point $\mathbf{x} \in \mathcal{S}_i$ we will always have a corresponding point $\mathbf{y} \in \mathcal{S}_j$ with $\mathbf{g}(\mathbf{x}) = \mathbf{g}(\mathbf{y})$, for all i and j . This behaviour is illustrated in Figure 3.9.

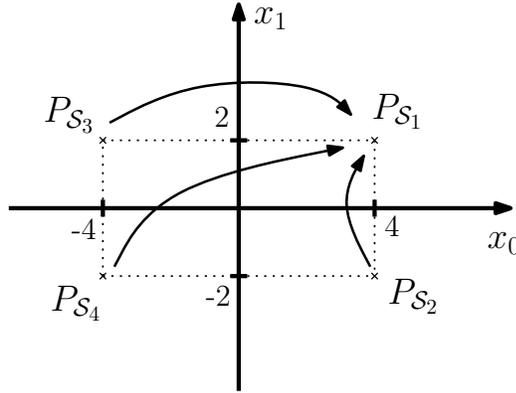


Figure 3.9: Depiction of the points in $P_{\mathcal{S}_1} \in \mathcal{S}_1, P_{\mathcal{S}_2} \in \mathcal{S}_2, P_{\mathcal{S}_3} \in \mathcal{S}_3, P_{\mathcal{S}_4} \in \mathcal{S}_4$ that get mapped to the same values by the functions $g_1 + g_2$.

It is possible to apply the same strategy to partition the output \mathcal{S}' of the hidden layer given by g_1, g_2 . This will result in a connectivity structure as the one in Figure 3.10. Note that this network has 3 hidden layers, with the second layer being the linear project that gives g_1 and g_2 . Because this second layer is linear, it can be folded into the weight matrix going from layer 2 to layer 3, meaning that there exists a two layer rectifier model that has exactly the same behaviour. This is summarized in the following lemma:

Lemma 3. *A layer of n rectifier units with U inputs can compute any function that can be computed by the composition of a linear layer with U inputs and U' outputs and a rectifier layer with U' inputs and n_1 outputs, for any $U, U', n_1 \in \mathbb{N}$.*

Proof. A rectifier layer computes functions of the form $\mathbf{x} \mapsto \text{rect}(\mathbf{W}\mathbf{x} + \mathbf{b})$, with $\mathbf{W} \in \mathbb{R}^{n_1 \times U}$ and $\mathbf{b} \in \mathbb{R}^{n_1}$. The argument $\mathbf{W}\mathbf{x} + \mathbf{b}$ is an affine function of \mathbf{x} . The

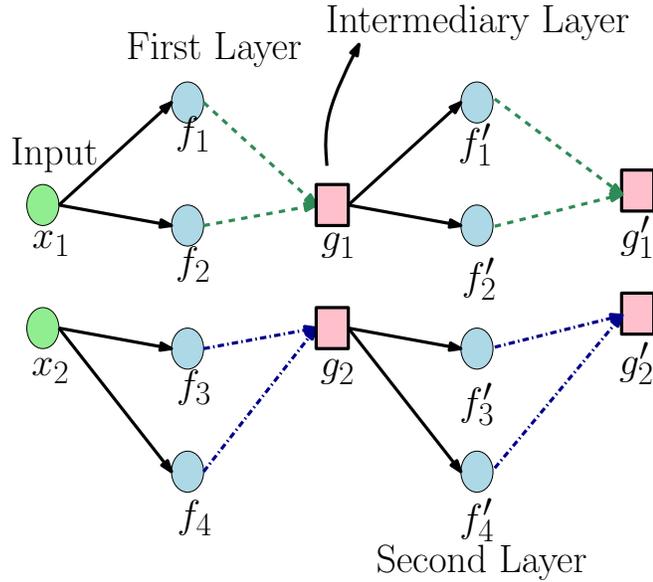


Figure 3.10: Depiction of a 2 layer rectifier MLP that has a intermediary linear layer whose activations correspond to g_1 and g_2 . Because the intermediary layer is linear, it can be folded into the weight and bias from layer 1 to layer 2.

claim follows from the fact that any composition of affine functions is an affine function. \square

Applying the absolute value function to coordinates in \mathcal{S}' is not very useful on its own, as these coordinates are positive by construction being the result of applying the absolute value function to (x_1, x_2) . However we can use a large negative bias on each dimension to get negative values. If we construct 4 rectifier units, f'_1, f'_2, f'_3 and f'_4 , in an analogous way as before, such that we get g'_1 and g'_2 , where, for $\mathbf{x} \in \mathcal{S}'$,

$$g'_1 = \text{abs}(x_1 - b)$$

$$g'_2 = \text{abs}(x_2 - b),$$

we again partition \mathcal{S}' into four square regions. One of the regions will have a finite area. This is because in the re-centered \mathcal{S}' (after we have subtracted the bias b), we only have negative numbers from $[-b, 0)$.

This partition of \mathcal{S}' will be *copied* to each \mathcal{S}_i in the original input space. Figure 3.11 (a) illustrates this behaviour by showing different points of the input (in the different input regions of the partition) that get mapped to the same value by

this two layer deep construction. Figure 3.11 (b) generalize this to regions, showing that a segment drawn in the space defined by the top layer results into 16 different segments in the original input space, one in each region of the partition.

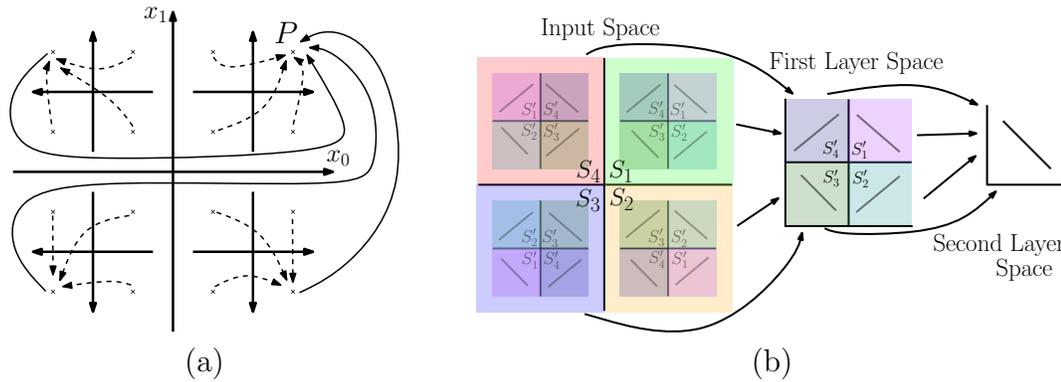


Figure 3.11: (a) Illustration of points in the different input regions of the input that get mapped to the same value by this two layer deep model. (b) Illustration of a recursive subdivision of the input space into four squares. Shading is used to indicate the different regions of the input. The plot shows how a segment drawn in the space defined by the top layer corresponds to 16 different segments, one in each input partition.

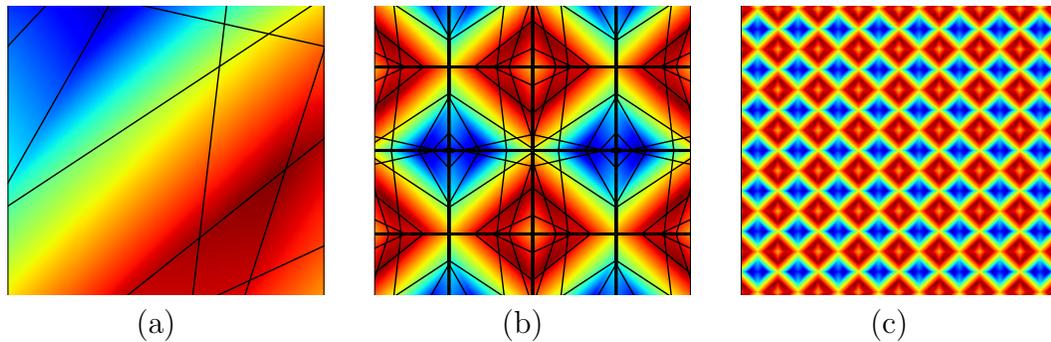


Figure 3.12: (a) Illustration of the partition computed by 8 rectifier units on the outputs (x_1, x_2) of the preceding layer. The color is a heat map (the values of the functions are map to colors from blue to red, with blue being lower) of $x_1 - x_2$. (b) Heat map of a function computed by a rectifier network with 2 inputs, 2 hidden layers of width 4, and one linear output unit. The black lines delimit the regions of linearity of the function. (c) Heat map of a function computed by a 4 layer model with a total of 24 hidden units. It takes at least 137 hidden units on a shallow model to represent the same function.

Note the exponential growth (with the number of layers) of the number of repetitions in the input space of any partition done on the last layer. For example look at the partition done on the second layer in Figure 3.11. We divided S' into 4 regions. If we trace these partition back to the input space, we see that the

partition gets replicated in all of the four square cones $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$ and \mathcal{S}_4 . This results in 16 regions in the input space.

Generally, on the last layer, we may place rectifiers in any way suitable for the task of interest (e.g., classification). The partition computed by the last layer will be copied to each of the input space regions that produced the same input for the last layer. Figure 3.12 shows a function that can be implemented efficiently by a deep model using this mechanism.

Roughly speaking for each layer up to the last one we get to multiply the total number of regions by 4. On the last layer we can do an arrangement in general position, which results in

$$4^{k-1} \left(1 + 4 + \binom{4}{2} \right)$$

regions of a deep model constructed as detailed above. This means that even for a small number of layers k , we can have many more linear regions in a deep model than in a shallow one. For example, a shallow model with 8 units will have at most 37 linear regions. The equivalent deep model with two layers of 4 units can produce 44 linear regions. For 12 hidden units the shallow model computes at most 79 regions, while the equivalent three layer model can compute 176 regions.

3.3.3 Folding the space

The mechanism used by the special class of deep models introduced above relies on the concept of *identified regions by g* . We will use freely the word *identify* to express this concept, as in, e.g., g *identifies* two regions.

Definition 4. A map g between continuous spaces **identifies** two input neighborhoods S and T if $g(S) = g(T)$. In this case we also say that S and T are identified by g .

Optionally we will refer to a set of neighborhoods or regions as being *identified* by some function g with the following meaning:

Definition 5. A map g **identifies** a set of neighbourhoods $\{S_1, S_2, \dots, S_k\}$ if for any two neighbourhoods S_i and S_j are **identified** by g .

Within this definition, for instance, in the special class of deep models introduced previously, the four quadrants of 2-D Euclidean space are *identified* by the

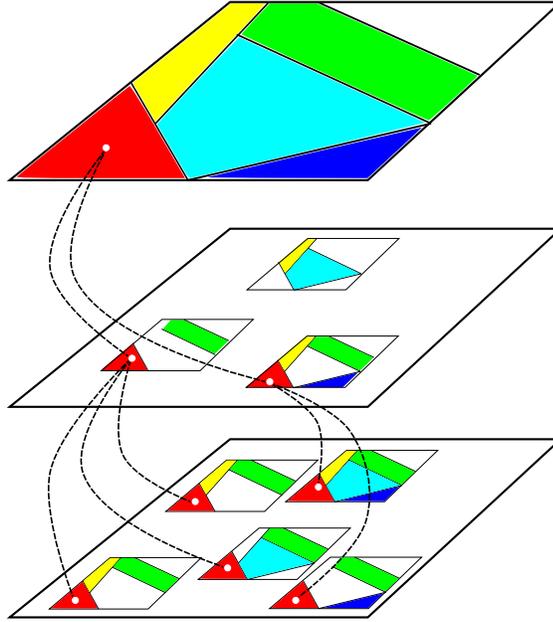


Figure 3.13: Identification of regions across the layers of a deep model. Colors are used to indicate regions that are identified with each other. This is just a illustration.

coordinate-wise absolute value function $g: \mathbb{R}^2 \rightarrow \mathbb{R}^2$;

$$g(x_1, x_2) = \begin{bmatrix} |x_1| \\ |x_2| \end{bmatrix}. \quad (3.8)$$

By identifying pieces of the input space of a neural network, each subsequent layer computation can be focused on a single output region, effectively acting on many inputs that have been identified by the previous layer. One can define the subsequent layers of the network in such a way that any of their computations concentrate on that single output region, thus replicating that complex computation on all identified inputs and generating an overall complicated-looking function.

To reiterate this idea, let us consider again the special class of deep models introduced in the previous section. The second layer in this model employs a large negative bias, and, by mimicking the coordinate-wise absolute value, it divides the input of the layer into four regions. The input of the second layer in this case is given by the first quadrant of the plane (as the first layer applies the absolute value to each coordinate). Therefore, this split into four regions gets repeated into each of the four quadrants. One can see this by, for example, looking for all the input

points that result in one of the boundaries learnt on the last layer. This behaviour is depicted in Figure 3.11. In Figure 3.13 we show a more generic illustration of this mechanism for a generic network.

Each hidden layer j computes a function g_j on the image of the preceding layer. We denote the image of the j th layer by $S_j \subseteq \mathbb{R}^{n_j}$. Given a subset $R \subseteq S_j$, we denote by P_R^j the set of subsets $\bar{R}_1, \dots, \bar{R}_k \subseteq S_{j-1}$ that are mapped injectively by g_j onto R ; that is, the subsets that satisfy $g_j(\bar{R}_1) = \dots = g_j(\bar{R}_k) = R$.

The number of separate input-space regions that are mapped to a common region $R \subseteq S_j \subseteq \mathbb{R}^{n_j}$ of the output-space of the j th layer can be given recursively as

$$\mathcal{N}_R^j = \sum_{R' \in P_R^j} \mathcal{N}_{R'}^{j-1}, \quad \mathcal{N}_R^0 = 1, \text{ for each region } R \subseteq \mathbb{R}^U. \quad (3.9)$$

For example, P_R^1 is the set of all input-space regions that are identified by the first layer such that their image by $g_1(\mathbf{W}_1 \cdot + \mathbf{b}_1)$ contains the region $R \subseteq S_1 \subseteq \mathbb{R}^{n_1}$.

Let η_j be the function that computes the activations of the hidden layer j starting from the input. Namely $\eta_j = g_j \circ g_{j-1} \circ \dots \circ g_1$. $\mathcal{N}_R^j \in \mathbb{N}$ is the number of regions in the input space that are identifiable with respect to the function η_j such that the image of these identifiable regions through η_j contains the region R .

This effectively builds a tree rooted at the region in the output and counts recursively the number of leaf nodes (see Figure 3.13). In other words, Equation (3.9) counts the number of times that each separate region in the output space is copied in the input space.

Based on the recursion from Equation (3.9), one can estimate the number of input space regions as follows.

Lemma 4. *The maximal number of regions of linearity of the functions computed by a neural network with U input variables and L hidden layers of n_i rectifiers for $i \in [L]$ is at least*

$$\mathcal{N} = \sum_{R \in \bar{R}^L} \mathcal{N}_R^L,$$

where \mathcal{N}_R^L is defined by Equation (3.9) and the maximal cardinality of \bar{R}^L is given by $\sum_{j=0}^U \binom{n_L}{j}$.

Proof. The equation for the maximal cardinality of \bar{R}^L is a consequence of Za-

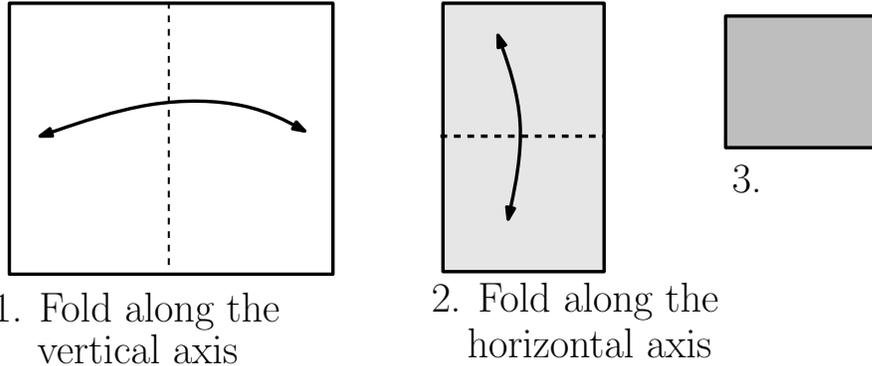


Figure 3.14: Space folding of 2-D Euclidean space along the two axes.

slavsky’s theorem, as discussed previously in Section 3.2. Each output region R is replicated within all the identifiable regions of the input whose image through η_L contains R . This value is given by \mathcal{N}_R^L (see the definition in Equation (3.9)). Therefore a lower bound on the maximal number of regions is given by summing up how many times each output region is replicated in the input space, which is just $\mathcal{N} = \sum_{R \in \bar{R}^L} \mathcal{N}_R^L$. \square

3.3.4 Identification of Inputs as Space Foldings

Before providing a specific construction for a deep model where one can easily compute the value of \mathcal{N}_R^L in Theorem 4, let us discuss further the concept of *identify*. One can understand the mechanism outlined above in terms of *space folding*.

A map g that identifies two subsets \mathcal{S} and \mathcal{S}' of a space can be considered (loosely) as a folding operator that *folds* the space such that the subsets \mathcal{S} and \mathcal{S}' coincide. For instance, the coordinate-wise absolute function $g: \mathbb{R}^2 \rightarrow \mathbb{R}^2$ in Equation (3.8) folds the input space once along each coordinate axis. In this case, this folding is equivalent to saying that the four quadrants of the 2-D Euclidean space are identified by the map g . See Figure 3.14 for an illustration. The same map can be used again to fold the resulting image of the original input space.

One can easily see that each space folding corresponds to a single hidden layer of a deep neural network. Each hidden layer can folds the space defined by the layer immediately below with respect to a specific map. A deep neural network effectively folds its input recursively, starting from the first layer.

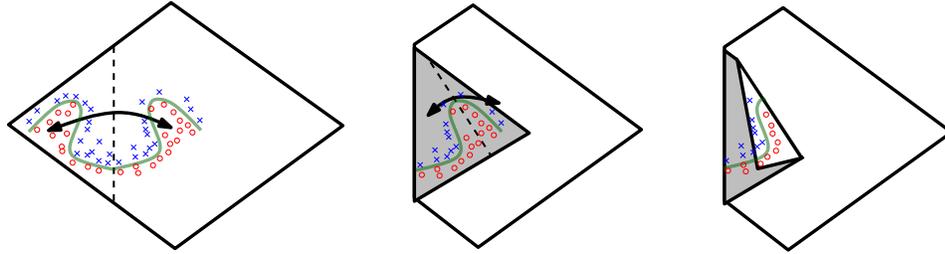


Figure 3.15: Space folding of 2-D space in a non-trivial way. Note how the folding can potentially identify symmetries in the boundary that it needs to learn.

The consequence of a recursive folding is that any partitioning of the final folded space will apply to all the collapsed subsets (identified by the map corresponding to the multiple levels of folding). Effectively, this means that any partitioning of the output space of a deep neural network is replicated over all the subsets of the input space which are identified by a map defined by a stack of hidden layers. See Figure 3.13 for an illustration of this replication property.

The space folding is not restricted to be done along the axes of the space, nor to preserve lengths. Rather, how the space is folded depends on the structure and nonlinear function used at each hidden layer. A set of complicated folding schemes may be applied to a space, which also means that the precise shapes of the identified subsets may differ from each other. See Figure 3.15 for one more illustration.

3.3.5 Symmetries in \mathbb{R}^U

One important question left is *when is it useful to fold the space?*. As we pointed out earlier, while deep models can have more response regions than shallow models, the linear maps for these regions are not independent from each other. They share a lot of the parameters of the model. So in reality, the choice is between few independent linear maps and many linear maps that share parameters and are constrained by each other.

The *folding space* metaphor provides us the intuition of how these linear maps generated by a deep model are constrained. Namely they are the result from regions of the input being *identified* by the model. If say two regions R_1 and R_2 are identified by the model, then it means that the linear maps g_1 and g_2 that describe the behaviour over these regions have to be such that $g_1(R_1)$ is the same (or a subset) of $g_2(R_2)$.

This property is not useful for example if we try to model white noise. In principle, white noise is such that there are no regions of the input that one could identify such that the resulting noise on these regions is the same. Deep models would *not* be able to better model white noise compared to shallow models, because they will not be able to identify any regions of the space and fold them on top of each other.

For deep models to be able to take advantage of this *folding* mechanism, the function that we want to model has to be highly *symmetric* on its input space. Or in other words the function has to be *invariant* to certain changes of the input. If the function shows this property, then the deep model can *identify* the input regions for which the function is symmetric and *fold* them on top of each other. Most functions we want to learn using machine learning approaches do exhibit such properties. This means deep models can learn them more efficiently than shallow model, while at the same time avoiding over-fitting by relying on the prior that the learnt function has to be symmetric.

To make the observation above less abstract, we can illustrate it as follows. Assume we want to learn a gender classifier based on images of faces. And let us further assume that the images we try to classify exhibit two types of emotions, they are either sad or happy. We want our trained model to be *invariant* to the emotion. We want to get the right gender regardless of whether the face is sad or happy. To do so, in principle a shallow model will have to learn separately the boundary between happy female face and happy male face, happy female face and sad male face, sad female face and happy male face and finally sad female face and sad male face. On the other hand, a deep model could *identify* the region of happy faces with the region of sad faces and *fold* them on top of each other. This means, in its upper layer, it will only need to learn one boundary, between female faces and male faces.

See also Figures 3.6 and 3.7 and 3.8. These visualizations show the different responses a unit in a higher layer can have.

Figure 3.16 shows three examples of such inputs that get mapped to the same value by some unit on the last hidden layer. The exact inputs were obtained as follows. We picked randomly a hidden unit on the third hidden layer. We then selected all training examples that resulted in an activation close to 2.5 for said hidden unit. Using the linear response of the unit for each of these examples, we

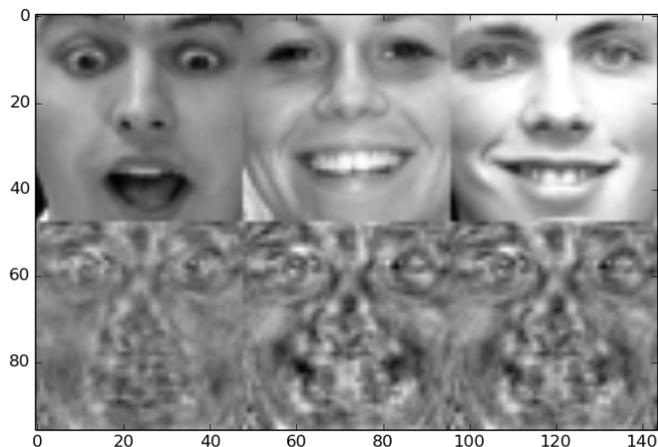


Figure 3.16: Visualization of three different input points that get mapped to the exact same value of 2.5 by a hidden unit on the last hidden layer. The first row shows the input examples that were obtained by interpolating training examples along the linear response of the unit in each instance. The second row shows the filter of the linear response of the unit in each case. See text for more details.

slightly altered these examples (interpolated them along the direction given by the filter) until they resulted in an activation of 2.5. The first row of Figure 3.16 shows the new obtained input examples (that do not belong to the training set, though are arguably undistinguishable from the training set). The second row of Figure 3.16 shows the linear response of the unit in each case.

There are two ways in which a unit in an intermediary layer can result in the same activation for two different inputs. If the two inputs belong to the same linear region of the unit, i.e. the unit has the same linear response for each, then this can be achieved if the difference between the two input examples is orthogonal to the response of the unit. The second approach, which is the focus of this chapter, regards the case when the two input examples belong to different linear regions of the input space. Given the input of the third hidden layer, the map that results in the activation of the hidden unit is constant. Given that the difference of the activations in the second layer for the two inputs is not orthogonal to the weights of the hidden unit, then the first two hidden layer have to *fold* or *identify* these two examples, resulting in the *same* activations in the second layer.

Alternatively, we can understand this property by looking at the response filter of the unit in each case. Because the unit has different responses for the different

inputs, it means that the unit is invariant to the transformations that convert one response into another. In Figure 3.7 we visualize the deltas between the 4 linear responses of a unit in the third hidden layer. These deltas indicate the kind of transformations that the unit is invariant to.

3.3.6 Extending the special class of deep models

In this section we will extend the analysis previously introduced class of deep models (see Section 3.3.2). Specifically we will show how to drop the constraint of having only two input units and having, on each layer, two times the number of input units.

Let us consider a single hidden layer of n rectifiers with U input variables, where $n \geq U$. We can partition the set of rectifiers into U (non-overlapping) subsets of cardinality $p = \lfloor \frac{n}{U} \rfloor$.

We can now look at one of the U subsets and construct p rectifiers such that

$$\begin{aligned} f_1(\mathbf{x}) &= \max \{0, \mathbf{w}^\top \mathbf{x}\}, \\ f_2(\mathbf{x}) &= \max \{0, 2\mathbf{w}^\top \mathbf{x} - 1\}, \\ f_3(\mathbf{x}) &= \max \{0, 2\mathbf{w}^\top \mathbf{x} - 2\}, \\ &\vdots \\ f_p(\mathbf{x}) &= \max \{0, 2\mathbf{w}^\top \mathbf{x} - (p - 1)\}, \end{aligned}$$

where $\mathbf{w} = [0, \dots, 0, 1, 0, \dots, 0]$ is a vector that chooses a single coordinate from the input space.

We can linearly aggregate these p rectifiers into a single scalar value:

$$g(\mathbf{x}) = \begin{bmatrix} 1 & -1 & 1 & -1 & 1 & \dots \end{bmatrix} [f_1(\mathbf{x}), f_2(\mathbf{x}), f_3(\mathbf{x}), f_4(\mathbf{x}), \dots]^\top. \quad (3.10)$$

Since each of these p rectifiers acts only on one common input dimension (marked with 1 in \mathbf{w}), the constructed g effectively is a function that divides the input space $(-\infty, \infty)$ into p segments

$$(-\infty, \infty) = (-\infty, 0] \cup (0, 1] \cup (1, 2] \cup \dots \cup (p - 1, \infty).$$

For each segment, g computes a linear function whose image contains the interval

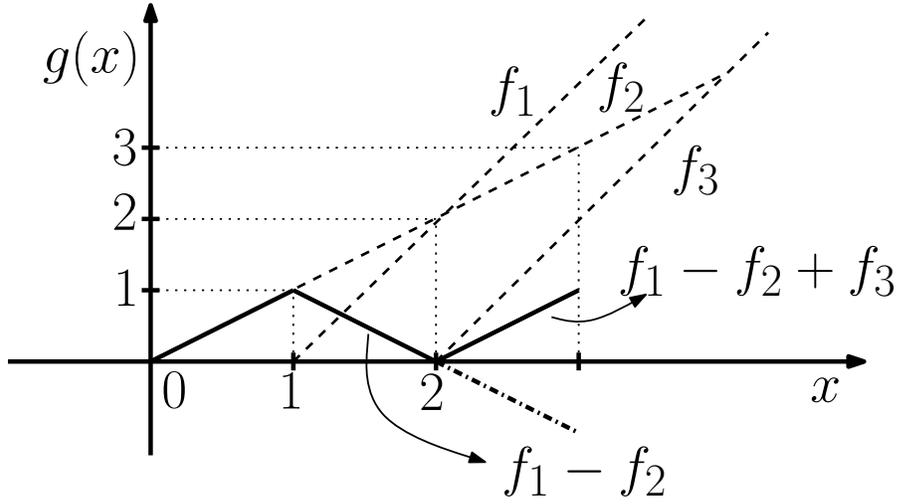


Figure 3.17: Folding of the real line into equal-length segments by a sum of rectifiers.

$(0, 1)$, as shown in Figure 3.17.

By considering all U subsets of rectifiers in a given layer, we obtain an identification of p^U hypercubes. The output of the hidden layer $\mathbf{g} = [g_1, g_2, \dots, g_p]^\top$ is symmetric about the hyperplane between any two hypercubes. Further computations by deeper layers on the image of the input space computed by g will apply to each of these p^U hypercubes.

3.3.7 Formal Result

We can generalize the procedure described above to the case where there are U input variables and L hidden layers of widths $n_i \geq U$ for all $i \in [L]$. In this case, the maximal number of regions is bounded below by the following theorem.

Theorem 1. *The maximal number of regions of linearity of the functions computed by a neural network with U input units and L hidden layers, with $n_i \geq U$ rectifiers at the i -th layer, is bounded below by*

$$\left(\prod_{i=1}^{L-1} \left\lfloor \frac{n_i}{U} \right\rfloor^U \right) \sum_{j=0}^U \binom{n_L}{j}.$$

Proof. The proof is done by counting the number of regions for a suitable choice of network parameters.

We organize the units such that at each layer j the set of units is partitioned into U subsets (or groups) of cardinality $\lfloor \frac{n_j}{U} \rfloor$. Each subset responds to a coordinate of a non-singular affine transformation of U -dimensional space. Specifically, as done in Section 3.3.2, based on Lemma 3, we insert an intermediary linear layer of U units between any two rectifier hidden layers. The weights and biases to each of the U subset of rectifier units of a given layer are arranged similar to the functions f_1, \dots, f_p of Section 3.3.6. The linear map constructs the function g for each of group by summing with alternating sign the responses of the corresponding rectifier units. This folds each coordinate into itself p times, once at each inflexion point of the different units in the group. This partitions the space into a grid of identified regions.

Specifically, for the weights used in Section 3.3.6, the function g computes a partial sum of the responses $(f_k)_{k \geq 1}$. It is sufficient to show that each of these partial sums, on the domain that they are defined, also takes values in the interval $(0, 1)$. By inspecting the values of f_k , we see that the intervals we need to explore are $(0, 1], (1, 2], \dots, (p-1, \infty)$.

For some $x \in (0, \infty)$ we denote $g(x) = S_l(y)$, where $l = \min(p, \lfloor x \rfloor)$ and $y = x - l$. The form of $g(x)$ becomes:

$$g(x) = x + \sum_{i=1}^l (2(x-i)(-1)^i) = S_l(y) = l + y + 2 \sum_{i=1}^l (y(-1)^i) + 2 \sum_{i=1}^l ((l-i)(-1)^i)$$

We solve this by induction, distinguishing between the case when l is odd or even. We first hypothesize that $S_l(y) = y$ if l is even, and $S_l(x) = 1 - y$ if l is odd.

The base cases are trivially true by the formula of the partial sum.

In the first induction step we consider l odd, implying $l-1$ is even. By induction we have

$$\begin{aligned} S_l(y) &= (l-1) + 1 + y + 2 \sum_{i=1}^{l-1} (l-1-i+y+1)(-1)^i - 2y \\ &= S_{l-1}(y) + \sum_{i=1}^{l-1} 2(-1)^i + 1 - 2y = y + 1 - 2y = 1 - y \end{aligned}$$

For the second induction step we assume l is even, implying $l - 1$ is odd, and

$$\begin{aligned} S_l(y) &= (l - 1) + 1 + y + 2 \sum_{i=1}^{l-1} (l - 1 - i + y + 1)(-1)^i - 2y \\ &= S_{l-1}(y) + \sum_{i=1}^{l-1} 2(-1)^i + 1 - 2y = 1 - y - 2 + 1 + 2y = y \end{aligned}$$

□

Stability with respect to perturbation

Our lower bounds on the complexity attainable by deep models are based on suitable choices of the network weights. This does not mean that the bounds only hold in singular cases.

The parametrization of the functions computed by a given network is continuous, that is, the map $\psi: \mathbb{R}^N \rightarrow C(\mathbb{R}^U; \mathbb{R}^{n_L}); \theta = \{\mathbf{W}, \mathbf{b}\} \mapsto f_\theta$ is continuous.

We considered the number of regions of linearity of the functions f_θ . By definition, each region of linearity contains an open neighborhood of the domain \mathbb{R}^U . Given a function f_θ , there is an $\epsilon > 0$ such that for each ϵ -perturbation of the parameter θ , the resulting function $f_{\theta+\epsilon}$ has at least as many regions as f_θ , assuming there is only a finite number of regions.

The regions of linearity of f_θ are preserved under small perturbations of the parameters, because they have a finite volume. It may happen, however, that the perturbed function has additional regions of linearity, emerging at the intersection of regions from the unperturbed function.

If we define a probability density on the space of parameters (say uniform on a bounded domain), what is the probability of the event that the function represented by the network has a given number of regions of linearity? By the above discussion, the probability of getting a number of regions at least as large as the number resulting from any particular choice of parameters (for a uniform measure within a bounded domain) is nonzero, even though it may be very small.

3.4 Asymptotic behaviour

Here we derive asymptotic expressions of the formulas contained in Proposition 2 and Theorem 1. We use the following standard notation:

- $f(n) = O(g(n))$ means that there is a positive constant c_2 such that $f(n) \leq c_2 g(n)$ for all n larger than some N .
- $f(n) = \Theta(g(n))$ means that there are two positive constants c_1 and c_2 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n larger than some N .
- $f(n) = \Omega(g(n))$ means that there is a positive constant c_1 such that $f(n) \geq c_1 g(n)$ for all n larger than some N .

The notation holds under the assumption that the functions take only positive values.

Proposition 3.

- Consider a single layer rectifier MLP with kn units and U inputs. Then the maximal number of regions of linearity of the functions represented by this network is

$$\mathcal{R}(U, kn, 1) = \sum_{s=0}^U \binom{kn}{s},$$

and

$$\mathcal{R}(U, kn, 1) = O(k^U n^U), \quad \text{when } U = O(1) \text{ (is constant)}.$$

- Consider a k layer rectified MLP with hidden layers of width n and U inputs. Then the maximal number of regions of linearity of the functions represented by this network satisfies

$$\mathcal{R}(U, n, \dots, n, 1) \geq \left(\prod_{i=1}^{k-1} \left\lfloor \frac{n}{U} \right\rfloor^U \right) \sum_{s=0}^U \binom{n}{s},$$

and

$$\mathcal{R}(U, n, \dots, n, 1) = \Omega \left(\left\lfloor \frac{n}{U} \right\rfloor^{U(k-1)} n^U \right), \quad \text{when } U = O(1).$$

Proof. Here only the asymptotic expressions remain to be shown. It is known that

$$\sum_{s=0}^U \binom{m}{s} = \Theta \left(\left(1 - \frac{2U}{m} \right)^{-1} \binom{m}{U} \right), \quad \text{when } U \leq \frac{m}{2} - \sqrt{m}. \quad (3.11)$$

Furthermore, it is known that

$$\binom{m}{s} = \frac{m^s}{s!} \left(1 + O\left(\frac{1}{m}\right)\right), \quad \text{when } s = O(1). \quad (3.12)$$

When U is constant, $U = O(1)$, we have that

$$\binom{kn}{U} = \frac{k^U}{U!} n^U \left(1 + O\left(\frac{1}{kn}\right)\right).$$

In this case, it follows that

$$\sum_{s=0}^U \binom{kn}{s} = \Theta\left(\left(1 - \frac{2U}{kn}\right)^{-1} \binom{kn}{U}\right) = \Theta(k^U n^U) \quad \text{and also} \quad \sum_{s=0}^U \binom{n}{s} = \Theta(n^U).$$

Furthermore,

$$\left(\prod_{i=1}^{k-1} \lfloor \frac{n}{U} \rfloor\right)^U \sum_{s=0}^U \binom{n}{s} = \Theta\left(\lfloor \frac{n}{U} \rfloor^{U(k-1)} n^U\right). \quad \square$$

When k and U are fixed, then $\lfloor n/U \rfloor^{U(k-1)}$ grows polynomially in n , and k^U is constant. On the other hand, when n is fixed with $n > 2U$, then $\lfloor n/U \rfloor^{U(k-1)}$ grows exponentially in k , and k^U grows polynomially in k .

We now analyze the number of response regions as a function of the number of parameters.

Proposition 4. *The number of parameters of a deep model with $U = O(1)$ inputs, $O = O(1)$ outputs, and k hidden layers of width n is*

$$(k-1)n^2 + (k+U+O)n + O = O(kn^2).$$

The number of parameters of a shallow model with $U = O(1)$ inputs, $O = O(1)$ outputs, and kn hidden units is

$$(U+O)kn + n + O = O(kn).$$

Proof. For the deep model, each layer, except the first and last, has a weight matrix with n^2 entries and a bias vector of length n . This gives a total of $(k-1)n^2 + (k-1)n$

parameters. The first layer has nU weights and n bias. The output layer has nO weight entries and O bias entries. If we sum these together we get

$$(k-1)n^2 + n(k+U+O) + O = O(kn^2).$$

For the shallow model, the hidden layer has knU weight entries and kn bias. The output weight matrix has knO entries and O bias entries. Summing these together we get

$$kn(U+O) + n + O = O(kn).$$

□

The number of linear regions per parameter can be given as follows.

Proposition 5. *Consider a fixed number of inputs U and a fixed number of outputs O . The maximal ratio of the number of response regions to the number of parameters of a deep model with k layers of width n is*

$$\Omega \left(\left\lfloor \frac{n}{U} \right\rfloor^{U(k-1)} \frac{n^{U-2}}{k} \right).$$

In the case of a shallow model with kn hidden units, the ratio is

$$O(k^{U-1}n^{U-1}).$$

Proof. This follows by combining Proposition 3 and Proposition 4. □

We see that even fixing the number of parameters, deep models can still compute functions with many more regions of linearity than those computable by shallow models. This comes from the fact that per parameter bounds for the deep model still grow faster than the shallow model ones. Specifically we have the same behaviour, the ratio grows exponentially with the number of layers k for deep models, versus polynomially for shallow.

3.5 Other piecewise linear models

The results in the previous sections can be extended to other piecewise linear activation functions as well. See, for example, [Montufar, Pascanu, Cho, and Bengio \(2014\)](#), where we provide a treatment for the *maxout* unit. We will only briefly discuss here convolutional models. These are models that, at each layer, convolve the input from below with a bunch of learnt kernels, operation that is followed by pooling. That is, the result of the convolution (after some activation function is applied to it) is divided into square regions, and then only the max value over that region is provided as output of the layer.

Convolutional models implement a form of folding of the input space that is enforced by their structure. When the model pools over some region of activations, it enforces the output unit that represents the result of the pooling operation to have the same response for several inputs. Specifically, any input that has the same maximal response at one position in the pooling region, but a different weaker response somewhere else will result in the exact same activation. This corresponds to a particular folding of the space which promotes robustness to local translations.

By relying on the same intuitions as before, we can see that having multiple convolutional layers versus one, we can gain more linear regions. Each layer identifies more regions between them, folding them on top of each other, while the last layer divides all the folded regions in some specific way. For the specific structure of convolutional networks this translates, for example, into having units in the higher layer be more translation invariant compared to those on the lower layers.

On top of this specific folding of the space (given by the pooling operator in conjunction with the convolution that evaluates the same filter at different positions), a typical convolutional model also applies rectifiers on the pooled responses. This provides a second level of folding, which compared to the first one, is learnt. One can freeze the pooling units by only considering inputs that result in a certain unit from each pooling region to be selected. Once the effects of pooling and convolutions are removed from the analysis by this choice, we can analyse the effect of the rectifier activation function in the same way we did for the deep MLP case.

3.6 Conclusion and Outlook

In this section we tried to understand why it is useful to use a deep model versus a shallow one. In particular we restricted ourselves to piecewise linear models in order to allow for a rigorous mathematical analysis. For this class of models we rephrased the question by looking at how deep vs shallow piecewise linear models partition the space.

Our main result shows that the piecewise linear function represented by a deep model can have exponentially more linear pieces compared to a shallow model with the same number of parameters. Deep models are able to do so by having intermediary units that *identify* different regions of the input. Because these intermediary units have the same response on different input regions, the upper part of the model will become invariant to the choice of the input region (which should translate to some specific type of invariance in the input).

We can think of identifying different regions of the input by some intermediary unit as the process of folding the space such that those region fall on top of each other. The process of “folding” or “identifying” regions of input can be formally expressed.

One key observation is that while deep models partition the input in more regions, these regions are not independent of each other. The dependency between the responses is given by the shared weights. It is tightly connected with why deep models are able to generalize, as it imposes a prior on how the model behaves on unseen examples. Our work shows that the function of interest has to be symmetric (invariant) to some changes in the input to be more efficiently modeled by a deep model. This comes from the *folding* strategy that the network uses to gain this efficiency. The model *identifies* these symmetries or invariances, and folds the space according to them so that it exhibits the same behaviour regardless.

For example let us consider a hypothetical gender classification task based on images of faces. We would like to fold on top of each other the input regions that represent faces that are happy, sad, angry, etc. This way we only need to learn one boundary between a female face and a male one, within this folded space. A shallow model, instead, has to learn different boundaries for all the specific subclasses of faces.

We believe that our proposed geometrical perspective of piecewise linear models

can be also useful for other types of analysis. We showed, for example, that this intuition can be used to provide a proper visualization (and understanding) of the behaviour of units in higher layers. Another interesting direction is to look at learning within this framework. Asking question of the kind: what kind of foldings can stochastic gradient descent learn? How does the discontinuity between switching linear pieces affect learning? Also regularization terms such as dropout can be viewed from this perspective.¹. We believe our work is just a glimpse of what this perspective can be used for.

Finally we add that some of our analysis might be possible to be transferred to non-piecewise linear models. For example, it is easy to argue that a sigmoid or tanh model can also, in principle, identify regions of the input space and, by the same arguments used within this section, become more efficient at representing certain families of functions.

1. The idea of studying dropout in this perspective belongs to Justin Bayer.

4

Learning and non-convex optimization

While the previous chapter looked exclusively at how efficient a neural network is at modelling different families of functions, it completely ignored the learning problem. That is, while certain deep models can be exponentially more efficient than shallow ones at modelling certain types of functions, it is not obvious that we can actually find these optimal models in our family of functions \mathcal{F} .

The process of finding these functions relies on minimizing the empirical risk. In the case of neural networks, we know this function to be non-convex, and therefore the standard gradient descent methods we use do not guarantee to find the global minimum. Even worse, the algorithm might not even find a minimum at all, and, for example, it can get stuck in the plateau around some saddle point.

In this chapter we turn our attention to optimization. While we can not provide a solution (and there might not even be one) for all these issues surrounding optimization of neural network models, we do hope to provide new insight into the problem through our work.

Parts of this chapter were *taken* from [Pascanu and Bengio \(2014\)](#) and [Pascanu, Dauphin, Ganguli, and Bengio \(2014\)](#). Figures, mathematical derivations and even paragraphs are *borrowed* from this published works. The first paper focuses on natural gradient descent and the relationship of this algorithm to other recent algorithms from the literature. The second paper introduces the saddle point problem. Please see Section 1.1 for details about my personal contribution to both these works.

We start the chapter with a more in depth introduction of existing work and specific concepts for optimization.

4.1 Literature review

Optimization methods attempt to solve equations of the form given below. In this work we will focus on functions \mathcal{L} that are continuous and twice differentiable¹.

$$\theta^* \leftarrow \arg \min_{\theta \in \mathbb{R}^n} \mathcal{L}(\theta) \quad (4.1)$$

Depending on the nature of the function \mathcal{L} one can usually divide the class of problems into the cases convex or non-convex.

While convex problems are well understood and studied, the same can not be said about the non-convex case. Therefore often practitioners rely on algorithms developed under the convexity assumption even for non-convex tasks and hope that the algorithm will still behave well. This approach is not hopeless and it should lead to a local minima, as any surface can be locally approximated by a convex one (e.g., the second order Taylor approximation of the function or a first order Taylor approximation). However there are many ways in which the non-convexity of the surface might affect the optimization algorithm.

As a note, due to the non-convex nature of the task, it is well accepted in the community that convergence to a local minima is acceptable. Also convergence analysis is not usually possible, so the algorithms are, in general, compared empirically. One looks at convergence time for some specific benchmark dataset (for some specific model). While this is a very imprecise measure, it does provide some way of ranking the different possible algorithms.

We first start by introducing some optimization approaches proposed in the literature.

4.1.1 Second Order methods

As stated before, one important approach for improving convergence speed is to rely, at each step of our iterative algorithm, on a second order Taylor approximation of the function we want to minimize. This approach allows one to move further as the second order approximation is more reliable than the first-order one. Figure 4.1 (a) depicts this algorithm on a generic non-convex surface. In Figure 4.1 (b)

1. In practice a widely used activation function is the rectifier which is continuous everywhere except at 0 (or other similar piece-wise linear activations). In such situations we simply ignore the discontinuity (at assume the gradient at 0 is 0), even though mathematically it is not correct.

we show a typical situation in which considering second order information helps, high curvature valleys. The second order information rescales each direction of the gradient independently, moving much faster in direction of low curvature and slower in those of high curvature.

The underlying assumption is that the function is locally convex, namely that the Hessian is positive definite. If this is not true one usually restricts the approximation to a small neighbourhood, which in the limit will be flat, and hence can be seen as a convex surface. Later on, in Section 4.4, we will discuss the problem of locally non-convex surfaces in more detail.

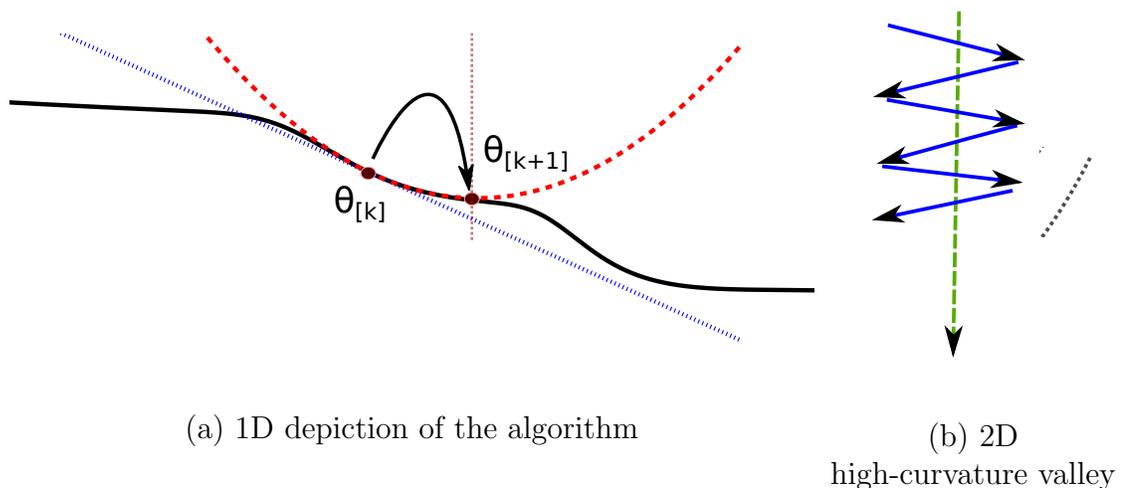


Figure 4.1: Depiction of a second order method. In (a) we show a Newton step. The red dashed line indicates the second order Taylor approximation of the function around $\theta_{[k]}$, and the dotted blue line the first order approximation. The dotted red line crossing the second order Taylor approximation shows the minimum of this approximation, and the arrow indicates the step we would take according to the Newton Method. In (b) we show a typical pattern that second order methods should address well, high curvature valleys. With blue solid arrows we show the behaviour of a first order method, while with green dashed arrow we show the step a second order method would take.

For neural networks, one of the main impediments against second order methods is the cost of computing the Newton step. In general, storing and computing the Hessian can already be prohibitive, and this with out even considering that we need to *invert* this matrix. In the following subsection we describe one technique to deal with these issues (but other approaches do exist).

Truncated Newton methods

One approach of computing the Newton step $\Delta\theta = -\mathbf{H}^{-1}\nabla\mathcal{L}^T$ is to reformulate it as a subproblem whose solution is the wanted quantity (Nocedal and Wright, 2006, Chapter 7.1). This approach is called a *truncated Newton method*, or an *inexact Newton method*. That is at each step of the optimization process we solve the following linear system:

$$\mathbf{H}\Delta\theta = -\nabla\mathcal{L}^T \tag{4.2}$$

Finding the solution of Equation (4.2) can be done without explicitly computing \mathbf{H} or its inverse if we employ a linear solver such as, for example conjugate gradient (CG) which we describe in Section 4.1.2. The only thing we need is to be able to compute efficiently the products $\mathbf{H}\mathbf{x}$ for arbitrarily \mathbf{x} .

Also we do not need the exact value of $\Delta\theta$ but rather a good estimate of it. Therefore, in practice, we do not need to run CG to convergence, we just need to do sufficiently many iterations to get a good estimate of $\Delta\theta$. This approach, compared to others, has the advantage that it uses the full Hessian matrix and hence considers the interactions between any two parameters. However, to make the approach tractable, the inversion of the matrix is approximated by truncating the maximal number of CG steps.

Another issue that one has to account for is a singular Hessian matrix or a not positive-definite one, which would contradict the assumptions made by CG. This is usually approached via damping (adding the identity matrix times some constant to the Hessian before inverting it), or, alternatively, one could use a different linear solver, as for example MinResQLP (Choi et al., 2011), that can deal with such ill-behaved matrices.

L and R operators

Let us first direct our attention on how to compute efficiently products of the form $\mathbf{H}\mathbf{x}$. There are two main approaches of doing so. The first one relies completely on the backpropagation algorithm, while the second one employs the R operator. We will introduce these strategies by first providing a bit of notation and by introducing the two mathematical operators L and R.

The R operator, introduced in [Pearlmutter \(1994\)](#), is just the directional derivative defined below:

$$\mathbf{R}\{f, \theta, \mathbf{s}\} = \lim_{r \rightarrow 0} \frac{f(\theta + r\mathbf{s}) - f(\theta)}{r} = \frac{\partial f(\theta)}{\partial \theta} \mathbf{s}. \quad (4.3)$$

The R operator can be used to apply the chain rule from right to the left, or, in other words, to apply the chain rule by starting at the inputs and moving towards the output of the computational graph representing the function. [Pearlmutter \(1994\)](#) shows that this operation can be implemented efficiently such that it only scales linearly with the number of parameters (and computational steps involved).

On the other hand, the L operator can be used to apply the chain rule from left to right (from the output moving towards the inputs). It is defined as follows:

$$\mathbf{L}\{f, \theta, \mathbf{s}\} = \mathbf{s} \frac{\partial f(\theta)}{\partial \theta}. \quad (4.4)$$

One can see that using the L operator we can re-derive the backpropagation algorithm. Specifically, if we have a composition of functions

$$f = f_1 \circ f_2 \circ \dots \circ f_k,$$

that results in a scalar function

$$f : \mathbb{R}^{n_\theta} \rightarrow \mathbb{R},$$

We can use the L operator to compute the partial derivative of f with respect to θ as follows:

$$\frac{\partial f}{\partial \theta} = \mathbf{L}\{f_k, \theta, \mathbf{L}\{f_{k-1}, f_k, \mathbf{L}\{\dots, \mathbf{L}\{f_1, f_2, 1\}\}\}\} \quad (4.5)$$

Note that the Equation (4.5) is an application of the chain rule from the left to the right (or from the output towards the input), where we use the scalar nature of the composition to initialize the recursive algorithm by multiplying to the left by 1. This is indeed just the backpropagation algorithm.

Furthermore, we can see that one way of expressing $\mathbf{H}\mathbf{x}$ is by using the equation below:

$$\mathbf{L} \left\{ \sum_{i=1}^{n_\theta} \mathbf{L} \{f, \theta, 1\}_i \mathbf{x}_i, \theta, 1 \right\} = \frac{\partial \sum_{i=1}^{n_\theta} \frac{\partial f}{\partial \theta_i} \mathbf{x}_i}{\partial \theta} = \begin{bmatrix} \sum_{i=1}^{n_\theta} \mathbf{x}_i \frac{\partial^2 f}{\partial \theta_i \partial \theta_1} \\ \dots \\ \sum_{i=1}^{n_\theta} \mathbf{x}_i \frac{\partial^2 f}{\partial \theta_i \partial \theta_{n_\theta}} \end{bmatrix} = \mathbf{H} \mathbf{x} \quad (4.6)$$

Equation (4.6) relies on multiplying the gradient element-wise with the vector \mathbf{x} and then taking the derivative of the element-wise sum of this product with respect to the parameter. Because the sum and partial derivative operations are linear, and \mathbf{x} is not a function of θ , the computations can be re-arranged, revealing that this is indeed the desired quantity. Both partial derivatives that we have to compute are partial derivatives of scalar functions, and therefore both can be computed using the backpropagation algorithm. Because the backpropagation algorithm scales linearly with the number of parameters so does this formulation.

Another approach relies on the R operator, and is described in the equation below:

$$\mathbf{R} \{ \mathbf{L} \{f, \theta, 1\}, \theta, \mathbf{x} \} = \lim_{r \rightarrow 0} \frac{\frac{\partial f(\theta + r\mathbf{x})}{\partial \theta} - \frac{\partial f(\theta)}{\partial \theta}}{\partial r} = \mathbf{H} \mathbf{x} \quad (4.7)$$

This approach involves applying the backpropagation algorithm, which is linear in the number of parameters, followed by applying the R operator, which is also linear in the number of parameters. Therefore the resulting algorithm does scale linearly in the number of parameters and is quite efficient in practice.

Trust region method

A trust region method is an approach of extending existing methods for convex optimization to the non-convex case. The concept of this approach is depicted in Figure 4.2. We know that the non-convex function f can only locally be assumed to be quadratic and convex (or the quadratic approximation of f is only true locally). We enforce this knowledge by limiting the size of the step we take to some ball in the parameter space whose radius indicates how far we can trust our approximation.

The radius of this ball is set heuristically and it can change during learning. The method can be seen as the dual of a line search (in some sense). When doing a line search we first fix a direction (by using an approximation of f) and then search for the size of the step that induces the minimum of f along that direction. For trust

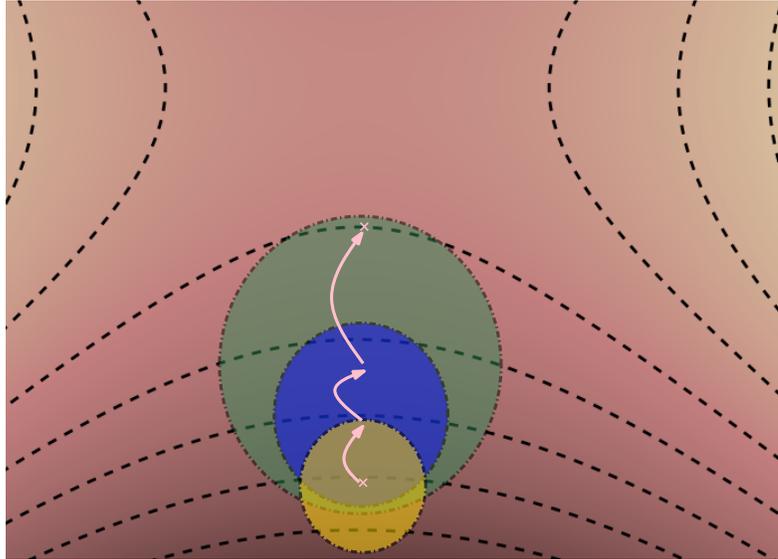


Figure 4.2: Depiction of the trust region approach. The figure is a heat map of the error surface (with dashed contour lines). The solid pink arrows indicate three steps taken by the algorithm. Note that this is just an illustration and not an actual experiment. The transparent disks with a dashed-dotted border depict the trust regions at each step. The step taken at each time is bounded by its corresponding region. Note that the region’s radius can change from step to step and that also the minimum of the quadratic, if it is indeed within the trust region, is not necessarily the minimum of the function. Picture is best seen in color.

regions, we first fix the maximal step size (the radius of the ball) and then check for the direction that provides the minimum of the second order approximation of f within this ball. For a more in depth introduction of this family of methods we suggest reading [Nocedal and Wright \(2006, Chapter 4\)](#).

Formally we can express the trust region approach by the following formula:

$$\begin{aligned} \arg \min_{\Delta\theta} f(\theta) + \nabla f \Delta\theta + \frac{1}{2} \Delta\theta^T \mathbf{H} \Delta\theta \\ \text{s. t. } \|\Delta\theta\|_2^2 = (\Delta\theta)^T \Delta\theta \leq r \end{aligned} \tag{4.8}$$

We use ∇f to indicate the partial derivative of f with respect to θ . The constant r is a hyper-parameter that can change from one iteration to another of the algorithm and it represents the radius of the trust region. There is usually some freedom in choosing which norm of $\Delta\theta$ we use, though in Equation (4.8) we rely on the squared Euclidean norm for convenience. Also the Hessian can be replaced by some approximation \mathbf{B} that can make computations more tractable, as for example a diagonal approximation of the Hessian \mathbf{H} .

One easy way of understanding how to solve this problem, is to attempt to solve it via Lagrange multipliers (or specifically, since we have an inequality constraint, using the Kuhn-Tucker method). This will tell us that we need to solve:

$$\arg \min_{\Delta\theta} f(\theta) + \nabla f \Delta\theta + \frac{1}{2} \Delta\theta^T \mathbf{H} \Delta\theta + \alpha (\Delta\theta)^T \Delta\theta, \quad (4.9)$$

where α is a function of r and comes from additional constraints that we have to Equation (4.9). We can further massage this equation, noticing that we can insert the identity matrix in between $(\Delta\theta)^T \Delta\theta$. This leads to :

$$\arg \min_{\Delta\theta} f(\theta) + \nabla f \Delta\theta + \frac{1}{2} \Delta\theta^T (\mathbf{H} + 2\alpha \mathbf{I}) \Delta\theta \quad (4.10)$$

We notice that Equation (4.10) the standard formula for damping the Hessian matrix. We can fold the constant 2 into α which bears the name of the damping coefficient. Because r was a hyper-parameter that was set heuristically, one can instead set α heuristically, avoiding to solve for α from our constraints which could be difficult. We therefore are left with the solution:

$$\Delta\theta = -(\mathbf{H} + \alpha \mathbf{I})^{-1} \nabla f^T \quad (4.11)$$

A useful observation is that damping can also be seen as simply adding α to each eigenvalue of the matrix \mathbf{H} . We can show this by looking at the definition of the eigenvalue, eigenvector pairs. Namely if λ_i and \mathbf{x}_i is such a pair, we know that:

$$\mathbf{H} \mathbf{x}_i = \lambda_i \mathbf{x}_i$$

If now we look at what happens when we use the damped matrix, we get:

$$(\mathbf{H} + \alpha \mathbf{I}) \mathbf{x}_i = \mathbf{H} \mathbf{x}_i + \alpha \mathbf{I} \mathbf{x}_i = (\lambda_i + \alpha) \mathbf{x}_i$$

This is useful as it shows how damping can help with negative curvature (which is equivalent to a negative-definite matrix) or a singular Hessian matrix. Namely if α is large enough, it will make all 0 eigenvalues equal to α and all negative eigenvalues positive. From this formulation one can also see that when $\alpha \rightarrow 0$, we recover the standard Newton method (as the radius of the trust region goes to ∞). On the other hand, when $\alpha \rightarrow \infty$, the step becomes infinitesimal in size (as we

scale with roughly $1/\alpha$), but, also, the direction of the step becomes aligned with the direction of the gradient. That is, α not only controls the magnitude of the step, but also the orientation, smoothly interpolating between a scaled version of the first order gradient and the Newton step. Geometrically, this happens because, as the trust region radius becomes smaller, both the second and first order approximations of the function become more and more reliable, to the point that both approximate f equally well.

Finally we make a note on how to update the damping coefficient during learning. One basic heuristic is given by looking at how well our second order approximation of the function is at the step that we want to take. That is, we compute the following value, where the denominator is the change in f predicted by our quadratic approximation :

$$\rho = \frac{f(\theta + \Delta\theta) - f(\theta)}{\nabla f \Delta\theta + \frac{1}{2} \Delta\theta^T (\mathbf{H} + \alpha \mathbf{I}) \Delta\theta} \quad (4.12)$$

If $\rho > \rho_{max}$ then our approximation is pretty reliable and we can reduce the damping α by multiplying it with $1/\alpha_{scale}$. This will increase the radius of our trust region. If $\rho < \rho_{min}$ then our approximation is not reliable and we need to increase the damping by multiplying it with α_{scale} . The constants $\rho_{max}, \rho_{min}, \alpha_{scale}$ are hyper-parameters of the algorithm. This heuristic is sometimes called the Levenberg-Marquardt heuristic as it was introduced for the Levenberg-Marquardt algorithm (More, 1978).

4.1.2 Conjugate Gradient

Conjugate Gradient (CG) is one specific algorithm for solving the linear system:

$$\mathbf{Ax} = \mathbf{b}, \quad (4.13)$$

where \mathbf{A} is a square, positive-definite and symmetric matrix, $\mathbf{A} \in \mathbb{R}^{n \times n}$. For an indepth introduction please see Nocedal and Wright (2006, Chapter 5) or Shewchuk (1994). The algorithm starts from $\mathbf{x}_{[0]}$ (usually set to 0) and picks, at each step, a direction that is \mathbf{A} -orthogonal or *conjugate* to the previously chosen directions. Conjugacy is defined below:

$$\mathbf{x} \text{ is conjugate with } \mathbf{y} \text{ iff } \mathbf{x}^T \mathbf{A} \mathbf{y} = 0 \quad (4.14)$$

Algorithm 3 Conjugate Gradient applied for solving $\mathbf{A}\mathbf{x} = \mathbf{b}$

```

1:  $\mathbf{r}_{[0]} \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_{[0]}$ 
2:  $\mathbf{d}_{[0]} \leftarrow \mathbf{r}_{[0]}$ 
3:  $\delta \leftarrow \mathbf{r}_{[0]}^T \mathbf{r}_{[0]}$ 
4:  $k \leftarrow 0$ 
5: while  $\mathbf{r}_{[k]}^T \mathbf{r}_{[k]} > \epsilon^2 \delta$  and  $k \leq k_{max}$  do
6:    $\mathbf{q}_{[k]} \leftarrow \mathbf{A}\mathbf{d}_{[k]}$ 
7:    $\alpha_{[k]} \leftarrow \frac{\mathbf{r}_{[k]}^T \mathbf{r}_{[k]}}{\mathbf{d}_{[k]}^T \mathbf{q}_{[k]}}$ 
8:    $\mathbf{x}_{[k+1]} \leftarrow \mathbf{x}_{[k]} + \alpha_{[k]} \mathbf{d}_{[k]}$ 
9:    $\mathbf{r}_{[k+1]} \leftarrow \mathbf{r}_{[k]} - \alpha_{[k]} \mathbf{q}_{[k]}$ 
10:   $\beta_{[k]} \leftarrow \frac{\mathbf{r}_{[k+1]}^T \mathbf{r}_{[k+1]}}{\mathbf{r}_{[k]}^T \mathbf{r}_{[k]}}$ 
11:   $\mathbf{d}_{[k+1]} \leftarrow \mathbf{r}_{[k+1]} + \beta_{[k]} \mathbf{d}_{[k]}$ 
12:   $k \leftarrow k + 1$ 
13: end while

```

Pseudocode describing CG is provided as Algorithm 3. For each conjugate search direction $\mathbf{d}_{[k]}$, the algorithm computes the step size $\alpha_{[k]}$ needed to minimize the objective along that direction. Also it proposes a new conjugate direction $\mathbf{d}_{[k+1]}$ by adding to the remaining residual the old conjugate direction times β . The factor β is such that it imposes conjugacy between $\mathbf{d}_{[k+1]}$ and previous search directions. Finally, the matrix \mathbf{A} is only used to compute products with the different search directions explored by the algorithm. This means that you do not need to store or compute \mathbf{A} as long as you can provide efficient means for computing products between this matrix and some vector $\mathbf{d}_{[k]}$.

Linear Conjugate Gradient can be extended to the nonlinear case, obtaining in this way the Nonlinear Conjugate Gradient (NCG) algorithm. NCG however is a heuristic method that, as any method on a non-convex task, can only find a local minimum. CG, in contrast, is guaranteed to solve the linear system within n steps. The idea of NCG is to explore directions that are conjugate with respect to the Hessian of the function. When the function is actually quadratic the algorithm behaves as CG and the same convergence speed can be attained.

Unfortunately, in the general case, when we drop the quadratic assumption, the algorithm comes with out any guarantees. Because the Hessian is not constant

from one step to another, we can not enforce the search directions to be conjugate with each other anymore. The best we can do is to ask for local conjugacy, where only two consecutive search directions are conjugate, or a few consecutive ones if we make a smoothness assumption on the Hessian (the assumption says that $\mathbf{H}_{[k]}$ is about the same as $\mathbf{H}_{[k+1]}$). This leads to the common practice of resetting the search direction $\mathbf{d}_{[k]}$ to $\mathbf{r}_{[k]}$ after a predefined number of steps (so that your search directions do not become too biased by the form the Hessian had several steps before, a form that is not relevant anymore).

There are three main alterations done to the CG algorithm to obtain NCG. The first one is that the value \mathbf{r} is now given by the derivative of the function with respect to the parameters. Secondly, α is computed using a line-search to minimize the objective. Thirdly, computing the β term for NCG is also more complicated because the Hessian matrix can change from one step to another. There is no unique and universally accepted formula for β , but rather several proposals that can behave differently depending on the problem. All these proposals lead to the same solution in the quadratic case or when used for (linear) CG and hence distinguishing between them in these cases is not as important. Some of the popular choices for formulas to compute β are the Fletcher-Reeves, Polak-Ribiere or Hestenes-Stiefel formulations given below. Polak-Ribiere is the more widely used to compute a new conjugate direction as it behaves better than the original Fletcher-Reeves formulation.

$$\beta_{\text{Fletcher-Reeves}} = \frac{\langle \mathbf{r}_{[k+1]}, \mathbf{r}_{[k+1]} \rangle}{\langle \mathbf{r}_{[k]}, \mathbf{r}_{[k]} \rangle} \quad (4.15)$$

$$\beta_{\text{Polak-Ribiere}} = \frac{\langle \mathbf{r}_{[k+1]}, \mathbf{r}_{[k+1]} \rangle - \langle \mathbf{r}_{[k+1]}, \mathbf{r}_{[k]} \rangle}{\langle \mathbf{r}_{[k]}, \mathbf{r}_{[k]} \rangle} \quad (4.16)$$

$$\beta_{\text{Hestenes-Stiefel}} = \frac{\langle \mathbf{r}_{[k+1]}, \mathbf{r}_{[k+1]} \rangle - \langle \mathbf{r}_{[k+1]}, \mathbf{r}_{[k]} \rangle}{\langle \mathbf{d}_{[k]}, \mathbf{r}_{[k+1]} \rangle - \langle \mathbf{d}_{[k]}, \mathbf{r}_{[k]} \rangle} \quad (4.17)$$

Here we have used angular brackets (“ $\langle \rangle$ ”) to indicate inner products between vectors. Finally the convergence of CG (or NCG) can be accelerated by using *preconditioning*. Preconditioning can help improve the condition number of the matrix that needs to be inverted, making numerical computations more stable. This is usually done via a preconditioning matrix \mathbf{M}^{-1} that gets multiplied on both sides of the linear system. The art of properly preconditioning consists in choosing the right matrix \mathbf{M} which is usually a task dependent one. However, there

are standard preconditioners that improve the behaviour of CG/NCG in general. One such example is the Jacobi preconditioner which is a diagonal matrix whose diagonal elements correspond to the diagonal of \mathbf{A} (or the diagonal of the Hessian matrix in the nonlinear case).

Other improvements deal with the nature of \mathbf{A} . For example, MinResQLP (Choi et al., 2011) is such an improvement that relaxes the constraints on \mathbf{A} and allows singular or non-positive definite matrices \mathbf{A} . When the matrix is singular, MinResQLP will return a minimal residual solution.

4.2 Generalized trust region methods

In this section we propose the framework of *generalized trust region methods*. They are a straightforward extension of trust region methods, which we propose as a unifying framework for many of the existing optimization techniques recently proposed for deep learning.

Let $\mathcal{T}_k\{f, \theta, \Delta\theta\}$ denote the first k terms ($k \leq 2$) of the Taylor expansion of f around θ evaluated at $\Delta\theta$. When there is no confusion, we will sometimes abridge the notation to simply $\mathcal{T}_k\{f\}$. The following equation spells out the meaning of $\mathcal{T}_k\{f\}$:

$$\mathcal{T}_k\{f, \theta, \Delta\theta\} = f(\theta) + \frac{\partial f}{\partial \theta} \Delta\theta + \frac{1}{2} \Delta\theta^T \frac{\partial^2 f}{\partial \theta^2} \Delta\theta + \dots \quad (4.18)$$

We call a generalized trust region method any iterative optimization algorithm that, at each step, solves the following subproblem to obtain the (direction of the) step $\Delta\theta$:

$$\begin{aligned} \Delta\theta &= \arg \min_{\Delta\theta} \mathcal{T}_k\{\mathcal{L}, \theta, \Delta\theta\} \quad \text{with } k \in \{1, 2\} \\ \text{s. t. } &d(\theta, \theta + \Delta\theta) \leq r \end{aligned} \quad (4.19)$$

Where d is some relevant distance measure. There are two fundamental changes from the standard formulation of trust regions. The first change is that we allow to also minimize a first order Taylor expansion of our function \mathcal{L} , rather than only a second order approximation. The second change is that we replaced the norm of $\Delta\theta$ by some distance measure between θ and $\theta + \Delta\theta$.

4.3 Analysis of certain optimization techniques for deep learning

In this section we describe (or re-derive) some proposed variants of second order methods or natural gradient approaches for neural networks. We also describe the relationship between these methods and, where possible, we show how they can be understood as generalized trust region methods.

4.3.1 Natural gradient descent

Natural gradient descent can be traced back to [Amari \(1985\)](#) and the algorithm was analyzed in several publications since. To name just a few, please see [Amari et al. \(1992\)](#); [Amari \(1998\)](#); [Heskes \(2000\)](#); [Park et al. \(2000\)](#); [Kakade \(2001\)](#); [Peters and Schaal \(2008\)](#); [Le Roux et al. \(2008\)](#); [Sun et al. \(2009\)](#); [Arnold et al. \(2011\)](#); [Desjardins et al. \(2013\)](#). We provide here a derivation of the algorithm that is similar to the one proposed by [Heskes \(2000\)](#), a derivation that we also used in [Desjardins, Pascanu, Courville, and Bengio \(2013\)](#). For clarity, in our description of the algorithm we avoid relying on concepts from Riemannian geometry, instead we only make use of basic concepts from calculus and constrained optimization. This derivation is complementary to the more traditional description that we have provided in [Section 2.4.3](#).

Natural gradient can be understood as a generalized trust region method, [Equation \(4.18\)](#), where we use a first order Taylor expansion of the objective and d is given by the change in the model (in the KL sense), where the model is described by the probability density function p . Note that most models can be interpreted from a probabilistic perspective as some distribution, which we denote here with p . For example, a standard neural network is usually seen as a conditional distribution between input and target.

We are looking for $\Delta\theta$ that minimizes the first order Taylor expansion of \mathcal{L} when the second order Taylor approximation of the KL-divergence between p_θ and $p_{\theta+\Delta\theta}$ has to be constant:

$$\begin{aligned} & \arg \min_{\Delta\theta} \mathcal{L}(\mathbf{z}; \theta + \Delta\theta) \\ & \text{s. t. } \text{KL}(p_\theta(\mathbf{z}) || p_{\theta+\Delta\theta}(\mathbf{z})) = \text{const.} \end{aligned} \tag{4.20}$$

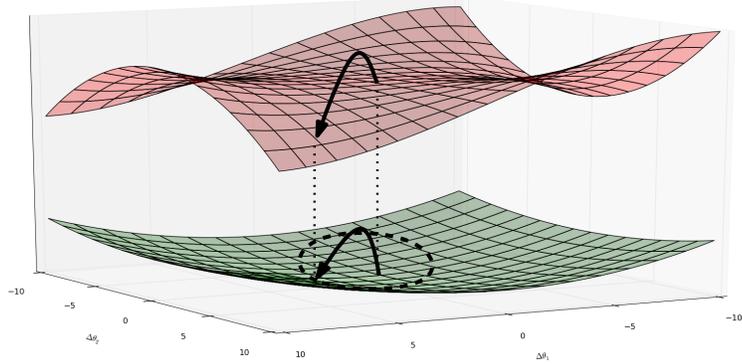


Figure 4.3: An illustration of natural gradient descent as a trust region. The upper surface (red) represents the error surface. The surface on the bottom (green) indicates the KL-divergence between p_θ and $p_{\theta+\Delta\theta}$ as a function of $\Delta\theta$. Note how each step goes to the boundary of the trust region.

We use an equality constraint instead of an inequality as Equation (4.18) dictates. However, because we rely on the first order Taylor expansion, the two are equivalent. The reason is that the first order equation has a minimum at either minus or plus infinity, and hence, even with an inequality constraint we will always jump to the border of our trust region. See Figure 4.3 for a visualization. Using an equality is however more intuitive as it offers a better understanding of what the algorithm is trying to do: we want to minimize our loss, but at the same time taking a step that induces a *fixed change* in p . That is we want to move with constant speed on the manifold, without being slowed down by its curvature.

This also makes learning locally *robust with respect to re-parametrizations of the model*, as the functional behaviour of p does not depend on how it is parametrized.

Assuming $\Delta\theta \rightarrow 0$, we can approximate the KL divergence by its second order Taylor series:

$$\begin{aligned}
 \text{KL}(p_\theta \parallel p_{\theta+\Delta\theta}) &\approx (\mathbf{E}_z [\log p_\theta] - \mathbf{E}_z [\log p_{\theta+\Delta\theta}]) \\
 &\quad - \mathbf{E}_z [\nabla \log p_\theta(\mathbf{z})] \Delta\theta - \frac{1}{2} \Delta\theta^T \mathbf{E}_z [\nabla^2 \log p_\theta] \Delta\theta \\
 &= \frac{1}{2} \Delta\theta^T \mathbf{E}_z [-\nabla^2 \log p_\theta(\mathbf{z})] \Delta\theta \\
 &= \frac{1}{2} \Delta\theta^T \mathbf{F} \Delta\theta
 \end{aligned} \tag{4.21}$$

The first term cancels out, and, because $\mathbf{E}_{\mathbf{z}} [\nabla \log p_{\theta}(\mathbf{z})] = 0^1$, we are left with only the last term. The Fisher Information Matrix form can be obtained from the expected value of the Hessian through algebraic manipulations as follows:

$$\begin{aligned}
\mathbf{E}_{\mathbf{z}} \left[-\frac{\partial^2 \log p_{\theta}}{\partial \theta} \right] &= \mathbf{E}_{\mathbf{z}} \left[-\frac{\partial \frac{1}{p_{\theta}} \frac{\partial p_{\theta}}{\partial \theta}}{\partial \theta} \right] = \mathbf{E}_{\mathbf{z}} \left[-\frac{1}{p_{\theta}(\mathbf{z})} \frac{\partial^2 p_{\theta}}{\partial \theta^2} + \left(\frac{1}{p_{\theta}} \frac{\partial p_{\theta}}{\partial \theta} \right)^T \left(\frac{1}{p_{\theta}} \frac{\partial p_{\theta}}{\partial \theta} \right) \right] \\
&= -\frac{\partial^2}{\partial \theta^2} \left(\sum_{\mathbf{z}} p_{\theta}(\mathbf{z}) \right) + \mathbf{E}_{\mathbf{z}} \left[\left(\frac{\partial \log p_{\theta}(\mathbf{z})}{\partial \theta} \right)^T \left(\frac{\partial \log p_{\theta}(\mathbf{z})}{\partial \theta} \right) \right] \\
&= \mathbf{E}_{\mathbf{z}} \left[\left(\frac{\partial \log p_{\theta}(\mathbf{z})}{\partial \theta} \right)^T \left(\frac{\partial \log p_{\theta}(\mathbf{z})}{\partial \theta} \right) \right] \tag{4.22}
\end{aligned}$$

We now express Equation (4.20) as a Lagrangian, where the KL divergence is approximated by Equation (4.21) and $\mathcal{L}(\theta + \Delta\theta)$ by its first order Taylor series $\mathcal{L}(\theta) + \nabla \mathcal{L}(\theta) \Delta\theta$:

$$\mathcal{L}(\theta) + \nabla \mathcal{L}(\theta) \Delta\theta + \frac{1}{2} \lambda \Delta\theta^T \mathbf{F} \Delta\theta \tag{4.23}$$

Solving Equation (4.23) for $\Delta\theta$ gives us the natural gradient decent formula:

$$\begin{aligned}
\nabla_N \mathcal{L}(\theta) &= \nabla \mathcal{L}(\theta) \mathbf{E}_{\mathbf{z}} \left[(\nabla \log p_{\theta}(\mathbf{z}))^T (\nabla \log p_{\theta}(\mathbf{z})) \right]^{-1} \\
&= \nabla \mathcal{L}(\theta) \mathbf{F}^{-1}. \tag{4.24}
\end{aligned}$$

We use ∇_N for the natural gradient to distinguish it from ∇ for gradients. Note that we get a scalar factor of $2\frac{1}{\lambda}$ times the natural gradient. We fold this scalar into the learning rate, which now also controls the weight we put on preserving the KL-distance between p_{θ} and $p_{\theta+\Delta\theta}$. The approximations we use in equation (4.21) are meaningful only around θ : in Schaul (2012) it is shown that taking large steps might harm convergence. We deal with such issues both by using damping (i.e. setting a trust region around θ by adding another constraint on the norm of $\Delta\theta$) and by properly selecting a learning rate.

1. Proof: $\mathbf{E}_{\mathbf{z}} [\nabla \log p_{\theta}(\mathbf{z})] = \sum_{\mathbf{z}} \left(p_{\theta}(\mathbf{z}) \frac{1}{p_{\theta}(\mathbf{z})} \frac{\partial p_{\theta}(\mathbf{z})}{\partial \theta} \right) = \frac{\partial}{\partial \theta} (\sum_{\mathbf{z}} p_{\theta}(\mathbf{z})) = \frac{\partial 1}{\partial \theta} = 0$. The proof holds for the continuous case as well, replacing sums for integrals.

Adapting natural gradient descent for neural networks

In order to use natural gradient descent for deterministic neural networks we rely on their probabilistic interpretation (see Section 2.1.2 of this work or Bishop (2006, Chapter 3.1.1 and Chapter 4.2) for linear regression or classification). For example, the output of an MLP with linear activation function can be interpreted as the mean of a conditional Gaussian distribution with a fixed variance, where we condition on the input. Minimizing the squared error, under this assumption, is equivalent to maximum likelihood.

For classification, depending on the activation function, we can define the output units as being the success probability of a Bernoulli distribution or the event probability of a multinoulli distribution conditioned on the input \mathbf{u} .

By an abuse of notation¹, we will use $p_\theta(\mathbf{t}|\mathbf{u})$ to define this conditional probability density function described above. Because it is a conditional probability function, the formulation of natural gradient descent, Equation (4.20), changes into the following equation:

$$\begin{aligned} & \arg \min_{\Delta\theta} \mathcal{L}(\theta + \Delta\theta) \\ \text{s. t. } & \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}(\mathbf{u})} [\text{KL}(p_\theta(\mathbf{t}|\mathbf{u}) || p_{\theta+\Delta\theta}(\mathbf{t}|\mathbf{u}))] = \text{const.} \end{aligned} \tag{4.25}$$

Each value of \mathbf{u} now defines a different family of density functions $p_\theta(\mathbf{t}|\mathbf{u})$, and hence a different manifold. In order to measure the functional behaviour of $p_\theta(\mathbf{t}|\mathbf{u})$ for different values of \mathbf{u} , we use the expected value (over \mathbf{u}) of the KL-divergence between $p_\theta(\mathbf{t}|\mathbf{u})$ and $p_{\theta+\Delta\theta}(\mathbf{t}|\mathbf{u})$.

In defining the constraint of equation 4.25, we have chosen to allow ourselves the freedom to compute the expectation over \mathbf{u} using some distribution $\tilde{\pi}$ instead of the empirical distribution π . Usually we want $\tilde{\pi}$ to be π as it is shown in Peters et al. (2003), though one can imagine situations when this would not be true. E.g. when we want our model to look more carefully at certain types of inputs, which we can do by biasing $\tilde{\pi}$ towards that type of inputs.

Applying the same steps as before we can recover the formula for natural gradient descent. This formula can be massaged further (similar to Park et al. (2000)) for specific activations and error functions.

1. E.g., for softmax output layer the random variable sampled from the multinoulli is a scalar not a vector

Linear activation function

In the case of linear outputs we assume that each entry of the vector \mathbf{t} , t_i comes from a Gaussian distribution centered around $\mathbf{y}(\mathbf{u})_i$ with some standard deviation β . From this it follows that:

$$p_\theta(\mathbf{t}|\mathbf{u}) = \prod_{i=1}^o \mathcal{N}(t_i|\mathbf{y}(\mathbf{u})_i, \beta^2) \quad (4.26)$$

$$\begin{aligned} \mathbf{F} &= \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} \left[\mathbf{E}_{\mathbf{t} \sim \mathcal{N}(\mathbf{t}|\mathbf{y}(\mathbf{u}, \theta), \beta^2 \mathbf{I})} \left[\sum_{i=1}^o \left(\frac{\partial \log p_\theta(t_i|\mathbf{y}(\mathbf{u})_i)}{\partial \theta} \right)^T \left(\frac{\partial \log p_\theta(t_i|\mathbf{y}(\mathbf{u})_i)}{\partial \theta} \right) \right] \right] \\ &= \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} \left[\sum_{i=1}^o \left[\mathbf{E}_{\mathbf{t} \sim \mathcal{N}(\mathbf{t}|\mathbf{y}(\mathbf{u}, \theta), \beta^2 \mathbf{I})} \left[\left(\frac{\partial(t_i - y_i)^2}{\partial \theta} \right)^T \left(\frac{\partial(t_i - y_i)^2}{\partial \theta} \right) \right] \right] \right] \\ &= \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} \left[\sum_{i=1}^o \left[\mathbf{E}_{\mathbf{t} \sim \mathcal{N}(\mathbf{t}|\mathbf{y}(\mathbf{u}, \theta), \beta^2 \mathbf{I})} \left[(t_i - y_i)^2 \left(\frac{\partial y_i}{\partial \theta} \right)^T \left(\frac{\partial y_i}{\partial \theta} \right) \right] \right] \right] \\ &= \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} \left[\sum_{i=1}^o \left[\mathbf{E}_{\mathbf{t} \sim \mathcal{N}(\mathbf{t}|\mathbf{y}(\mathbf{u}, \theta), \beta^2 \mathbf{I})} \left[(t_i - y_i)^2 \right] \left(\frac{\partial y_i}{\partial \theta} \right)^T \left(\frac{\partial y_i}{\partial \theta} \right) \right] \right] \\ &= \beta^2 \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} \left[\mathbf{J}_{\mathbf{y}}^T \mathbf{J}_{\mathbf{y}} \right] \end{aligned} \quad (4.27)$$

Here, \mathbf{J} stands for the Jacobian matrix $\frac{\partial \mathbf{y}}{\partial \theta}$. The subscript describes for which variable the quantity is computed over.

Sigmoid activation function

In the case of the sigmoid units, i.e. $\mathbf{y} = \text{sigmoid}(\mathbf{r})$, we assume a binomial distribution which gives us:

$$p(\mathbf{t}|\mathbf{u}) = \prod_i y_i^{t_i} (1 - y_i)^{1-t_i} \quad (4.28)$$

$\log p$ gives us the usual cross-entropy error used with sigmoid units. We can compute the Fisher information matrix as follows:

$$\begin{aligned} \mathbf{F} &= \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} \left[\mathbf{E}_{\mathbf{t} \sim p(\mathbf{t}|\mathbf{u})} \left[\sum_{i=1}^o \frac{(t_i - y_i)^2}{y_i^2(1-y_i)^2} \left(\frac{\partial y_i}{\partial \theta} \right)^T \frac{\partial y_i}{\partial \theta} \right] \right] \\ &= \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} \left[\sum_{i=1}^o \frac{1}{y_i(1-y_i)} \left(\frac{\partial y_i}{\partial \theta} \right)^T \frac{\partial y_i}{\partial \theta} \right] \\ &= \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} \left[\mathbf{J}_{\mathbf{y}}^T \text{diag} \left(\frac{1}{\mathbf{y}(1-\mathbf{y})} \right) \mathbf{J}_{\mathbf{y}} \right] \end{aligned} \quad (4.29)$$

Note that $\text{diag}(\mathbf{v})$ stands for the diagonal matrix constructed from the values of the vector \mathbf{v} and we make an abuse of notation, where by $\frac{1}{\mathbf{y}}$ we understand the

vector obtain by applying the division element-wise (the i -th element of $\frac{1}{\mathbf{y}}$ is $\frac{1}{y_i}$).

Softmax activation function

For the softmax activation function, $\mathbf{y} = \text{softmax}(\mathbf{r})$, $p(\mathbf{t}|\mathbf{u})$ takes the form of a multinoulli:

$$p(\mathbf{t}|\mathbf{u}) = \prod_{i=1}^o y_i^{t_i} \quad (4.30)$$

$$\mathbf{F} = \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} \left[\sum_{i=1}^o \frac{1}{y_i} \left(\frac{\partial y_i}{\partial \theta} \right)^T \frac{\partial y_i}{\partial \theta} \right] = \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} \left[\mathbf{J}_{\mathbf{y}}^T \text{diag} \left(\frac{1}{\mathbf{y}} \right) \mathbf{J}_{\mathbf{y}} \right] \quad (4.31)$$

4.3.2 Hessian-Free Optimization

Hessian-Free Optimization (HF) was recently introduced in [Martens \(2010\)](#); [Martens and Sutskever \(2011\)](#) for training deep and recurrent models. It represents one of the first works which show that one can achieve good results with deep models with out relying on pre-training, the technique of initializing the weights of the deep model by first performing, for each layer in turn, some unsupervised learning task.

The algorithm is a second order method that relies on a truncated Newton strategy for approximately inverting the Hessian. HF uses damping (and therefore it is a trust region method), where the damping factor is usually updated during learning by a Levenberg-Marquardt heuristic. What sets it apart from a standard truncated Newton method is the use of a specific approximation of the Hessian called the extended Gauss-Newton approximation of the Hessian introduced in [Schraudolph \(2001\)](#). Other particular changes of the pipeline are also important, like a modification of the stopping criterion for CG or backtracking, a strategy of going backwards through the step of CG until finding the best step-length for minimizing the function f . For HF, CG is also warm-started by using the previous descent direction as a starting point, speeding up convergence of the algorithm if the Hessian and gradient have not changed by much. One additional important achievement of these works is the successful application of this complicated algorithm to large scale problems.

We will focus our attention on the extended Gauss-Newton approximation,

which played a crucial role in the success of the algorithm. Let us decompose the error function into $\mathcal{L} \circ \sigma^{(out)} \circ \mathbf{r}$, where \mathbf{r} represents the function from the input of the model up to the output layer *before applying* the output activation function $\sigma^{(out)}$ and \mathcal{L} is the loss function (applied on the output of the model). We can now re-write the Hessian matrix as:

$$\mathbf{H} = \left(\frac{\partial \mathbf{r}}{\partial \theta} \right)^T \left(\frac{\partial^2 \mathcal{L}}{\partial \mathbf{r}^2} \right) \left(\frac{\partial \mathbf{r}}{\partial \theta} \right) + \sum_{i=0}^O \left(\frac{\partial \mathcal{L}}{\partial \mathbf{r}_i} \right) \left(\frac{\partial^2 \mathbf{r}_i}{\partial \theta^2} \right) \quad (4.32)$$

Based on this formulation, [Schraudolph \(2001\)](#) argues that the second term $\sum_{i=1}^O \left(\frac{\partial \mathcal{L}}{\partial \mathbf{r}_i} \right) \left(\frac{\partial^2 \mathbf{r}_i}{\partial \theta^2} \right)$ goes to 0 much faster when one is close to a minimum, so one can approximate the Hessian using only the first term of the sum. This term is named the extended Gauss-Newton approximation because in the linear case (when $\sigma^{(out)}$ is the identity function and \mathcal{L} is the square error) one recovers the Gauss-Newton approximation of the Hessian. We will denote it by \mathbf{G}_N defined below.

$$\mathbf{G}_N = \left(\frac{\partial \mathbf{r}}{\partial \theta} \right)^T \left(\frac{\partial^2 \mathcal{L}}{\partial \mathbf{r}^2} \right) \left(\frac{\partial \mathbf{r}}{\partial \theta} \right) \quad (4.33)$$

This new approximation is particularly useful for corresponding output activation functions and loss functions, such as identity function and square error, sigmoid and cross-entropy, softmax and negative log likelihood. In such a situation we say that the error function matches the output activation function. In these cases the Hessian $\frac{\partial^2 \mathcal{L}}{\partial \mathbf{r}^2}$ is diagonal and positive (it can be shown by algebraic manipulation for each case separately), making the whole approximation positive-definite. The products $\mathbf{G}_N \mathbf{x}$ can also be computed efficiently, similarly to $\mathbf{H} \mathbf{x}$, though one has to rely on both the R and the L operators. In the equation below we make the additional note that the Hessian $\frac{\partial^2 \mathcal{L}}{\partial \mathbf{r}^2}$ is diagonal and can easily be derived and evaluated if the error functions matches the output activation function:

$$\mathbf{G}_N \mathbf{x} = \mathbf{R} \left\{ \mathcal{L}, \theta, \frac{\partial^2 \mathcal{L}}{\partial \mathbf{r}^2} \mathbf{L} \{ \mathcal{L}, \theta, 1 \} \right\} \quad (4.34)$$

For training recurrent models, HF relies on an additional regularization term called *structural damping*. Structural damping asks that the hidden state of the recurrent model does not change by much from one step to another. Provided that a suitable distance measure is chosen that is paired with the hidden layer activation, its second order Taylor expansion reduces to the second order term. Additionally

we can take an extended Gauss-Newton approximation of this quantity, and, with the distance measure denoted by d and the pre-activation value of the hidden state by \mathbf{e} , we get the approximation:

$$\mathbf{G}_{sd} = \left(\frac{\partial \mathbf{e}}{\partial \theta} \right)^T \left(\frac{\partial^2 d}{\partial \mathbf{e}} \right) \left(\frac{\partial \mathbf{e}}{\partial \theta} \right) \quad (4.35)$$

Computing \mathbf{G}_{sd} defined by Equation (4.35) separately from \mathbf{G}_N would be impractical, but, as is apparent from Equations (4.33) (4.35), these matrices share many computations. Therefore the linear combination $\mathbf{G}_N + \alpha \mathbf{G}_{sd}$ is actually quite cheap to compute as shown in the equation:

$$\mathbf{G}_N + \alpha \mathbf{G}_{sd} = \left(\frac{\partial \mathbf{e}}{\partial \theta} \right)^T \left[\left(\frac{\partial \mathbf{r}}{\partial \theta} \right)^T \left(\frac{\partial^2 \mathcal{L}}{\partial \mathbf{r}^2} \right) \left(\frac{\partial \mathbf{r}}{\partial \theta} \right) + \alpha \left(\frac{\partial^2 d}{\partial \mathbf{e}^2} \right) \right] \left(\frac{\partial \mathbf{e}}{\partial \theta} \right) \quad (4.36)$$

Natural gradient descent and Hessian-Free Optimization

As already mentioned, Hessian-Free Optimization relies on the *extended Gauss-Newton approximation of the Hessian*, \mathbf{G}_N , which can be re-written as follows:

$$\begin{aligned} \mathbf{G}_N &= \frac{1}{n} \sum_{i=1}^n \left[\left(\frac{\partial \mathbf{r}}{\partial \theta} \right)^T \frac{\partial^2 \log p(\mathbf{t}^{(i)} | \mathbf{u}^{(i)})}{\partial \mathbf{r}^2} \left(\frac{\partial \mathbf{r}}{\partial \theta} \right) \right] \\ &= \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} \left[\mathbf{J}_{\mathbf{r}}^T \left(\mathbf{E}_{\mathbf{t} \sim \tilde{\pi}(\mathbf{t} | \mathbf{u})} [\mathbf{H}_{\mathcal{L}_{\text{or}}}] \right) \mathbf{J}_{\mathbf{r}} \right] \end{aligned} \quad (4.37)$$

The last step of Equation (4.37) assumes that $(\mathbf{u}^{(i)}, \mathbf{t}^{(i)})$ are i.i.d samples, and $\tilde{\pi}$ stands for the distribution represented by the mini-batch¹ over which the matrix is computed. A composition in the subscript, as in $\mathbf{H}_{\mathcal{L}_{\text{or}}}$, implies computing the Hessian of \mathcal{L} with respect to \mathbf{r} , with \mathbf{r} being the output layer before applying the activation function.

The reason for choosing this approximation over the Hessian is not computational, as computing both can be done equally fast. The extended Gauss-Newton approximation is better behaved² during learning. This is assumed to hold because

1. subset of training examples that is used to compute a descent direction by some iterative optimization technique

2. Such complex non-convex problems do not allow most of the time for any kind of convergence

\mathbf{G}_N is positive semi-definite by construction, so one does not need to worry about negative curvature.

It is known that the Gauss-Newton approximation (for linear activation function and square error) matches the Fisher Information matrix. In this section we show that *this identity holds also for other matching pairs like sigmoid and cross-entropy or softmax and negative log-likelihood for which the extended Gauss-Newton is defined*. By choosing this specific approximation, one can therefore view *Hessian-Free Optimization as being an implementation of natural gradient descent*. We make the additional note that [Heskes \(2000\)](#) makes similar algebraic manipulations as the ones provided in this section, however for different reasons, namely to provide a new justification of the algorithm that relies on distance measures. The original contribution of this section is in describing the relationship between Hessian-Free Optimization on the one hand and natural gradient descent on the other. This relation is not mentioned anywhere in the literature as far as we are aware of.

In the case of sigmoid units with cross-entropy objective, $\mathbf{H}_{\mathcal{L}_{\text{or}}}$ is

$$\begin{aligned} \mathbf{H}_{\mathcal{L}_{\text{or}}_{ij, i \neq j}} &= \frac{\partial^2 \sum_k (-t_k \log(\text{sigmoid}(r_k)) - (1-t_k) \log(1-\text{sigmoid}(r_k)))}{\partial r_i \partial r_j} \\ &= \frac{\partial \text{sigmoid}(r_i) - t_i}{\partial r_j} = 0 \\ \mathbf{H}_{\mathcal{L}_{\text{or}}_{ii}} &= \dots = \frac{\partial \text{sigmoid}(r_i) - t_i}{\partial r_i} = \text{sigmoid}(r_i)(1 - \text{sigmoid}(r_i)) \end{aligned} \quad (4.38)$$

If we insert this back into the Gauss-Newton approximation of the Hessian and re-write the equation in terms of \mathbf{J}_y instead of \mathbf{J}_r , we get the corresponding natural gradient metric, Equation (4.29):

$$\begin{aligned} \mathbf{G}_N &= \frac{1}{n} \sum_{\mathbf{u}^{(i)}, \mathbf{t}^{(i)}} \mathbf{J}_r^T \mathbf{H}_{\mathcal{L}_{\text{or}}} \mathbf{J}_r \\ &= \frac{1}{n} \sum_{\mathbf{x}^{(i)}} \mathbf{J}_r^T \text{diag}(\mathbf{y}(1-\mathbf{y})) \text{diag}\left(\frac{1}{\mathbf{y}(1-\mathbf{y})}\right) \text{diag}(\mathbf{y}(1-\mathbf{y})) \mathbf{J}_r . \\ &= \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} \left[\mathbf{J}_y^T \text{diag}\left(\frac{1}{\mathbf{y}(1-\mathbf{y})}\right) \mathbf{J}_y \right] \end{aligned} \quad (4.39)$$

The last matching activation and error function that we consider is the softmax with cross-entropy. The derivation of the Gauss-Newton approximation is given in Equation (4.40).

analysis; in general these properties are studied empirically by looking at the behaviour of the algorithm on some benchmark task.

$$\begin{aligned}
\mathbf{H}_{\mathcal{L}_{\text{or}}_{ij, i \neq j}} &= \frac{\partial^2 \sum_k (-t_k \log(\text{softmax}(r_k)))}{\partial r_i \partial r_j} = \frac{\partial \sum_k (t_k \text{softmax}(r_i) - t_i)}{\partial r_j} \\
&= -\text{softmax}(r_i) \text{softmax}(r_j) \\
\mathbf{H}_{\mathcal{L}_{\text{or}}_{ii}} &= \dots = \frac{\partial \text{softmax}(r_i) - t_i}{\partial r_i} = \text{softmax}(r_i) - \text{softmax}(r_i) \text{softmax}(r_i)
\end{aligned} \tag{4.40}$$

$$\begin{aligned}
\mathbf{F} &= \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} \left[\sum_{k=1}^o \frac{1}{y_k} \left(\frac{\partial y_k}{\partial \theta} \right)^T \frac{\partial y_k}{\partial \theta} \right] \\
&= \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} \left[\mathbf{J}_{\mathbf{r}}^T \left(\sum_{k=1}^o \frac{1}{y_k} \left(\frac{\partial y_k}{\partial \mathbf{r}} \right)^T \left(\frac{\partial y_k}{\partial \mathbf{r}} \right) \right) \mathbf{J}_{\mathbf{r}} \right] \\
&= \frac{1}{N} \sum_{\mathbf{u}^{(i)}} \left(\mathbf{J}_{\mathbf{r}}^T \mathbf{M} \mathbf{J}_{\mathbf{r}} \right)
\end{aligned} \tag{4.41}$$

$$\begin{aligned}
\mathbf{M}_{ij, i \neq j} &= \sum_{k=1}^o \frac{1}{y_k} \frac{\partial y_k}{\partial r_i} \frac{\partial y_k}{\partial r_j} = \sum_{k=1}^o (\delta_{ki} - y_i) y_k (\delta_{kj} - y_j) \\
&= y_i y_j - y_i y_j - y_i y_j = -\text{softmax}(r_i) \text{softmax}(r_j) \\
\mathbf{M}_{ii} &= \sum_{k=1}^o \frac{1}{y_k} \frac{\partial y_k}{\partial y_i} \frac{\partial y_k}{\partial r_j} = y_i^2 \left(\sum_{k=1}^o y_k \right) + y_i - 2y_i^2 \\
&= \text{softmax}(r_i) - \text{softmax}(r_i) \text{softmax}(r_i)
\end{aligned} \tag{4.42}$$

Equation (4.41) starts from the natural gradient metric and singles out a matrix \mathbf{M} in the formula such that the metric can be re-written as the product $\mathbf{J}_{\mathbf{r}}^T \mathbf{M} \mathbf{J}_{\mathbf{r}}$ (similar to the formula for the Gauss-Newton approximation). In Equation (4.42) we show that indeed \mathbf{M} equals $\mathbf{H}_{\mathcal{L}_{\text{or}}}$ and hence the natural gradient metric is the same as the extended Gauss-Newton matrix for this case as well. Note that δ is the Kronecker delta, where $\delta_{ij, i \neq j} = 0$ and $\delta_{ii} = 1$.

This identification between natural gradient and HF means that (in a non-trivial way) HF is also a generalized trust region method that relies on a first order Taylor expansion of the objective. There is also a one-to-one mapping between most of the other heuristics used by Hessian-Free Optimization.

Following the functional manifold interpretation of the algorithm, we can recover the Levenberg-Marquardt heuristic used in [Martens \(2010\)](#) if we consider a first order Taylor approximation on the manifold. For any function f , if $\Delta\theta$ depicts the picked descent direction and η the step size

$$f(\theta_t - \eta \Delta\theta) \approx f(\theta_t) - \eta \frac{\partial f(\theta_t)}{\partial \theta_t} \Delta\theta \tag{4.43}$$

This gives the reduction ratio given by Equation (4.44) which can, under the assumption that $p = \frac{\partial f(\theta_t)}{\partial \theta_t} \mathbf{F}^{-1}$, be shown to behave identically with the one in [Martens \(2010\)](#) (under the same assumption, namely that CG is close to conver-

gence).

$$\rho = \frac{f(\theta_t - \eta \Delta \theta) - f(\theta_t)}{-\eta \frac{\partial f(\theta_t)}{\partial \theta} \Delta \theta} \approx \frac{f\left(\theta - t - \eta \mathbf{F}^{-1} \frac{\partial f(\theta_t)}{\partial \theta}^T\right) - f(\theta_t)}{-\eta \frac{\partial f(\theta_t)}{\partial \theta} \mathbf{F}^{-1} \frac{\partial f(\theta_t)}{\partial \theta}^T} \quad (4.44)$$

Structural damping (Martens and Sutskever, 2011), a specific regularization term used to improve training of recurrent neural network, can also be explained from the natural gradient descent perspective. Roughly it implies using the joint probability density function $p(\mathbf{t}, \mathbf{h}|\mathbf{u})$, where \mathbf{h} is the hidden state, when writing the KL-constraint. The quantity $\log p(\mathbf{t}, \mathbf{h}|\mathbf{u})$ will break in the sum of two terms, one being the Fisher Information Matrix, the other measuring the change in \mathbf{h} and forms the structural damping term. While theoretically pleasing, however, this derivation results in a fixed coefficient of 1 for the regularization term.

We can be more flexible by using two constraints when deriving the natural gradient descent algorithm, namely:

$$\begin{aligned} & \arg \min_{\Delta \theta} \mathcal{L}(\theta + \Delta \theta) \\ \text{s. t. } & \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}(\mathbf{u})} [\text{KL}(p_{\theta}(\mathbf{t}|\mathbf{u}) || p_{\theta + \Delta \theta}(\mathbf{t}|\mathbf{u}))] = \text{const.} \\ & \text{and } \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}(\mathbf{u})} [\text{KL}(p_{\theta}(\mathbf{h}|\mathbf{u}) || p_{\theta + \Delta \theta}(\mathbf{h}|\mathbf{u}))] = \text{const.} \end{aligned} \quad (4.45)$$

If we apply the same steps as before for both constraints (i.e. replace them by a second order Taylor expansion), the second term will give rise to the structural damping term.

4.3.3 Natural gradient descent (TONGA)

In Le Roux et al. (2008) a new variant of natural gradient was introduced. The algorithm assumes that the stochastic gradients we get from MSGD, where each gradient is computed on a different mini-batch, are distributed according to a Gaussian centered around the true gradient with some covariance matrix \mathbf{C} .

One can, therefore, use the uncertainty given by \mathbf{C} to correct the step that we intend to take such that we maximize the probability of decreasing the generalization error (expected negative log-likelihood), resulting in a formula similar to that of natural gradient descent:

$$\Delta\theta = -\nabla\mathcal{L}\mathbf{C}^{-1} \tag{4.46}$$

While the probabilistic derivation requires the centered covariance \mathbf{C} , in [Le Roux et al. \(2008\)](#) it is argued that one can use the uncentered covariance \mathbf{U} resulting in a simplified formula:

$$\begin{aligned} \Delta\theta &\approx -\nabla\mathcal{L}\mathbf{E}_{(\mathbf{u},\mathbf{t})\sim\pi} \left[\left(\frac{\partial\log p(\mathbf{t}|\mathbf{u})}{\partial\theta} \right)^T \left(\frac{\partial\log p(\mathbf{t}|\mathbf{u})}{\partial\theta} \right) \right]^{-1} \\ &= -\nabla\mathcal{L}\mathbf{U}^{-1} \end{aligned}$$

In [Pascanu and Bengio \(2014\)](#) we argue that this algorithm is not identical with the natural gradient method proposed by [Amari \(1985\)](#) and, therefore, its name is a misnomer. The discrepancy comes from the fact that the equation is an expectation, though the expectation is *over the empirical distribution* $\pi(\mathbf{u}, \mathbf{t})$ as opposed to $\mathbf{u} \sim \pi(\mathbf{u})$ and having \mathbf{t} sampled from the model distribution $\mathbf{t} \sim p_\theta(\mathbf{t}|\mathbf{u})$ as is done in Amari’s work. It is therefore not clear if \mathbf{U} tells us how p_θ would change, whereas it is clear that Amari’s metric does.

One particular situation in which the matrices can be quite different is close to or at a critical point. In such a situation we know that \mathbf{U} will go to 0. The matrix \mathbf{U} is a sum of outer products of the gradients of the loss function \mathcal{L} . Because we are near a critical point of \mathcal{L} , its gradients must vanish, which will make their outer product to be close to 0 and hence \mathbf{U} will be close to singular. However, the Fisher Information matrix, \mathbf{F} , does not have to go to zero, as it is *not* the covariance of the gradients we follow to the local minima.

Another argument used to differentiate between TONGA and natural gradient is that the matrix \mathbf{U} tends to be more rank deficient than \mathbf{F} as it is composed by summing fewer outer products (when computed over the same mini-batch of examples). While [Martens \(2010\)](#); [Schraudolph \(2001\)](#) do not differentiate between TONGA and natural gradient descent, these papers, for example, do argue that \mathbf{U} is more rank deficient compared to the extended Gauss-Newton approximation of the Hessian (which we have previously shown to be identical to the Fisher Information matrix).

[Le Roux et al. \(2008\)](#) also introduce a specific implementation of their algo-

rithm (or more clearly a specific approximation of the inverse of \mathbf{U}) that they call TONGA. We will extend this name to refer to the whole algorithm within this manuscript, rather than this specific implementation. In other words, whenever we refer to TONGA we refer to the algorithm that uses the update step introduced in Equation (4.47). This will help us differentiate between this algorithm and Amari (1985).

Le Roux and Fitzgibbon (2010) argues that natural gradient descent (and specifically the TONGA variant) is a first order method, and, therefore, one can use second order information to improve the descent direction. This results in the following update rule:

$$\Delta\theta = -\nabla\mathcal{L} \left[\mathbf{I} + \frac{\mathbf{H}^{-1}\mathbf{C}\mathbf{H}^{-1}}{n\sigma} \right]^{-1} \mathbf{H}^{-1} \quad (4.47)$$

Finally, we note that TONGA can be as efficiently to implement as Hessian Free Optimization or truncated Newton, because the product between the uncentered covariance matrix \mathbf{U} and some vector \mathbf{x} can be computed as efficiently using the L and R operators:

$$\mathbf{U}\mathbf{x} = \mathbf{L}\{f, \theta, \mathbf{R}\{f, \theta, \mathbf{x}\}\}$$

Natural gradient descent (TONGA) as a generalized trust region method

While we argued that TONGA is not the same as Amari's natural gradient descent, the algorithm is, however, a generalized trust region method. To see this consider the constrained optimization where we look at the first order Taylor approximation of the loss, and where we also impose that the expected change of the loss is constant:

$$\begin{aligned} & \arg \min_{\Delta\theta} \mathcal{T}_1 \{ \mathcal{L}(\theta + \Delta\theta) \} \\ \text{s. t. } & \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}(\mathbf{u})} [\| \mathcal{L}(\theta + \Delta\theta) - \mathcal{L}(\theta) \|_2^2] = \text{const.} \end{aligned} \quad (4.48)$$

If we approximate $\mathcal{L}(\theta + \Delta\theta)$ in the constraint by its first order Taylor expansion, the constraint becomes:

$$\begin{aligned}
\mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} [\|\mathcal{L}(\theta + \Delta\theta) - \mathcal{L}(\theta)\|_2^2] &= \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} [\|\mathcal{L}(\theta) + \nabla\mathcal{L}\Delta\theta - \mathcal{L}(\theta)\|_2^2] \\
&= \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} [(\Delta\theta)^T [(\nabla\mathcal{L})^T(\nabla\mathcal{L})] \Delta\theta] \\
&= (\Delta\theta)^T \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}} [(\nabla\mathcal{L})^T(\nabla\mathcal{L})] \Delta\theta \\
&= (\Delta\theta)^T \mathbf{U} \Delta\theta
\end{aligned} \tag{4.49}$$

If we now use Lagrange multipliers to solve this constrained optimization, similarly to the steps taken for natural gradient in Section 4.3.1, we recover the formula proposed by TONGA.

4.3.4 Natural Conjugate Gradient

Natural gradient descent, being a first order method, can be extended to incorporate second order information, as also argued in [Le Roux and Fitzgibbon \(2010\)](#).

If we are to follow the Riemannian manifold understanding of natural gradient, one can easily implement a second order method on the manifold. See for example [Absil et al. \(2008\)](#) that describes how various standard optimization techniques can be extended to the Riemannian manifold case.

In [Honkela et al. \(2008, 2010\)](#) an extension of nonlinear conjugate gradient on the manifold is proposed in the context of variational inference and specific assumptions are made on the form of p_θ in order to make the algorithm tractable. [Gonzalez and Dorrnsoro \(2006\)](#) proposes a similar nonlinear conjugate gradient extension, but this time for MLPs.

In both papers, the algorithm makes use of a diagonal approximation of the matrix defining the metric and relies on the Polak–Ribiere formula for computing a new conjugate direction. We argue that this approach is problematic, as one needs to use the inner product of $\mathbf{r}_{[k]}$ and $\mathbf{r}_{[k+1]}$ to compute the factor β needed for finding the new conjugate direction, where β is given by Equation (4.16). But, since we are dealing with a Riemannian manifold, the two tangent vectors are likely to lie in different tangential planes. Specifically, the metric matrix describing the geometry around $\mathbf{r}_{[k]}$, $\mathbf{F}_{[k]}$, and the metric matrix describing the geometry around $\mathbf{r}_{[k+1]}$, $\mathbf{F}_{[k+1]}$ are different. Following [Absil et al. \(2008\)](#), we would need to map $\mathbf{r}_{[k]}$ into the space of $\mathbf{r}_{[k+1]}$, an expensive operation, before we can compute the inner

product and apply the Polak–Ribiere formula for β .

Gonzalez and Dorronsoro (2006); Honkela et al. (2010) address these issues by making the assumption that $\mathbf{F}_{[k]}$ and $\mathbf{F}_{[k+1]}$ are identical. This assumption is detrimental to the algorithm because it goes against what we want to achieve. By employing a conjugate gradient method we hope to make large steps, from which it follows that the metric is very likely to change. Hence the assumption can not hold.

A second approximation employed by both algorithms is to do a normal line search for finding α , instead of moving along geodesics which is a very expensive operation.

Being variants of natural gradient descent, the subproblem of finding the natural gradient direction is a generalized trust region method. However, natural conjugate gradient is not itself a generalized trust region method.

4.3.5 Krylov Subspace Descent

Vinyals and Povey (2012) introduces the Krylov Subspace Descent (KSD) algorithm for deep learning. The method takes a different approach to solving the linear system

$$\mathbf{G}_N \mathbf{x} = -\nabla \mathcal{L}^T.$$

We know that the solution $\Delta\theta$ lies in the Krylov subspace defined below:

$$\mathbf{S}_{n_\theta} = \text{span} \{ -\nabla \mathcal{L}^T, -\mathbf{G}_N \nabla \mathcal{L}^T, -\mathbf{G}_N^2 \nabla \mathcal{L}^T, \dots, -\mathbf{G}_N^{n_\theta} \nabla \mathcal{L}^T \} \quad (4.50)$$

Therefore, the algorithm constructs a restricted version of the subspace \mathbf{S}_k , with only k vectors, to gain tractability, and then searches within this restricted subspace for the best step using BFGS which is known to work well for lower dimensional problems. Formally we have:

$$\Delta\theta = \gamma \mathbf{S}_k = \begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \dots \\ \gamma_k \end{bmatrix} \begin{bmatrix} -\nabla \mathcal{L} \\ -\nabla \mathcal{L} \mathbf{G}_N \\ \dots \\ -\nabla \mathcal{L} \mathbf{G}_N^k \end{bmatrix} \quad (4.51)$$

$$\gamma = \arg \min_{\gamma \in \mathbb{R}^k} \mathcal{L}(\theta + \gamma \mathbf{S}_k)$$

A similar idea was explored previously in [Mizutani and Demmel \(2003\)](#). The main advantage of this algorithm is that damping becomes implicit (as BFGS optimizes directly \mathcal{L}) and one does not need to worry about setting the corresponding hyper-parameter. Also other details of the HF pipeline are not needed anymore, like backtracking, or the custom stopping criterion of CG. The algorithm is also reported to be somewhat faster than Hessian Free Optimization.

One additional detail of the algorithm proposed by [Vinyals and Povey \(2012\)](#), which we believe to be of some importance, is that they modify the Krylov subspace in order to mimic the warm start of CG used for HF. Namely, they add one additional vector to the subspace, vector given by the previous descent direction. The subspace becomes in this case:

$$\mathbf{S}_k = \text{span} \{ \mathbf{d}_{[t-1]}, -\nabla \mathcal{L}^T, -\mathbf{G}_N \nabla \mathcal{L}^T, -\mathbf{G}_N^2 \nabla \mathcal{L}^T, \dots, -\mathbf{G}_N^k \nabla \mathcal{L}^T \} \quad (4.52)$$

Natural conjugate gradient and Krylov Subspace Descent

Because Krylov Subspace Descent relies on the extended Gauss-Newton approximation of the Hessian, like Hessian-Free Optimization, KSD implements a variant of natural gradient descent. But there is an additional difference between KSD and HF that can be interpreted from the natural gradient descent perspective.

KSD adds to the Krylov subspace the previous search direction. We hypothesize that due to this change, KSD is more similar to natural conjugate gradient than natural gradient descent.

To show this we can rewrite the subproblem that KSD is solving as:

$$\arg \min_{\alpha, \beta, \gamma_1, \dots, \gamma_k} \mathcal{L} \left(\theta + \beta \mathbf{d}_{[t-1]} + \alpha \underbrace{\begin{bmatrix} \frac{\gamma_1}{\alpha} \\ \frac{\gamma_2}{\alpha} \\ \dots \\ \frac{\gamma_k}{\alpha} \end{bmatrix} \begin{bmatrix} \nabla \mathcal{L} \\ \nabla \mathcal{L} \mathbf{F} \\ \dots \\ \nabla \mathcal{L} \mathbf{F}^{k-1} \end{bmatrix}}_{\approx (\nabla \mathcal{L}) \mathbf{F}^{-1}} \right) \quad (4.53)$$

From this formulation one can see that the previous direction plays a different role than the one played when doing a warm restart for CG. The algorithm is remi-

niscient of the nonlinear conjugate gradient. The descent directions we pick, besides incorporating the natural gradient direction, also tend to be locally conjugate to the Hessian of the error with respect to the functional behaviour of the model. Additionally, compared to nonlinear conjugate gradient BFGS is used to compute β rather than some known formula like equations (4.15), (4.16) and (4.17).

4.3.6 Improved natural conjugate gradient

Computing the correct conjugate direction for natural conjugate gradient, introduced in Section 4.3.4, is difficult. Mapping one gradient to the space in which another gradient lies is expensive to compute in a generic manner, with out enforcing strict constraints on the form of p_θ . In this section we take inspiration from the new interpretation of the KSD algorithm, Section 4.3.5, and show the utility of such reinterpretations. In equation (4.53) we can see that β is computed by minimizing the cost \mathcal{L} with respect to both β and α . However the algorithm also requires computing and storing a large Krylov subspace and re-parametrizing the problem in this subspace (hence we also have to solve for $\gamma_1, \dots, \gamma_k$). We propose an algorithm that requires solving two subproblems for finding the new descent direction. The first subproblem is the same as the one solved by a truncated Newton approach. Namely we use linear CG to find the natural gradient. Once we have the natural gradient we can solve a 2-dimensional subproblem in α and β that finds a new conjugate direction (based on the previous one) by minimizing the loss \mathcal{L} :

$$\min_{\alpha, \beta} \mathcal{L} \left(\theta_{[t-1]} + \begin{bmatrix} \alpha_{[t]} \\ \beta_{[t]} \end{bmatrix} \begin{bmatrix} \nabla_{N_{[t]}} \\ d_{[t-1]} \end{bmatrix} \right) \quad (4.54)$$

The new direction is:

$$d_{[t]} = \nabla_{N_{[t]}} + \frac{\beta_{[t]}}{\alpha_{[t]}} d_{[t-1]} \quad (4.55)$$

The resulting algorithm looks like a truncated Newton implementation of the previously described natural conjugate gradient. The main difference is that we do not rely on a formula such as Polak–Ribiere to compute the new conjugate direction, but rather use an off-the-shelf solver to find it by minimizing the loss. The new conjugate direction might still not be the right direction, since, for example, the off-the-shelf optimizer ignores the manifold structure. However, we are always guaranteed to follow a descent direction, and, arguably, the best descent direction

that we can find within this re-parametrized 2-D problem.

We can show that in the Euclidean space, given a second order Taylor approximation of $\mathcal{L}(\theta_{[t-1]})$, this approach will result in following conjugate directions.

Let $\mathbf{d}_{[t-1]}$ be the previous direction and $\gamma_{[t-1]}$ the step size such that $\mathcal{L}(\theta_{[t-1]} + \gamma_{[t-1]}\mathbf{d}_{[t-1]})$ is minimal for fixed $\mathbf{d}_{[t-1]}$. If we approximate \mathcal{L} by its second order Taylor expansion and compute the derivative with respect to the step size $\gamma_{[t-1]}$ we have:

$$\frac{\partial \mathcal{L}}{\partial \theta} \mathbf{d}_{[t-1]} + \gamma_{[t-1]} \mathbf{d}_{[t-1]}^T \mathbf{H} \mathbf{d}_{[t-1]} = 0 \quad (4.56)$$

Suppose now that we take the next step which is defined implicitly by

$$\begin{aligned} & \mathcal{L}(\theta_{[t]} + \beta_{[t]}\mathbf{d}_{[t-1]} + \alpha_{[t]}\nabla_{N_{[t]}}^T) \\ &= \mathcal{L}(\theta_{[t-1]} + \gamma_{[t-1]}\mathbf{d}_{[t-1]} + \beta_{[t]}\mathbf{d}_{[t-1]} + \alpha_{[t]}\nabla_{N_{[t]}}^T), \end{aligned}$$

where we minimize for $\alpha_{[t]}$ and $\beta_{[t]}$. If we replace \mathcal{L} by its second order Taylor series around $\theta_{[t-1]}$, compute the derivative with respect to $\beta_{[t]}$ and use the fact that $\mathbf{H}_{[t-1]}$ (where we drop the subscript) is symmetric, we get:

$$\frac{\partial \mathcal{L}}{\partial \theta} \mathbf{d}_{[t-1]} + \alpha_{[t]} \nabla_{N_{[t]}} \mathbf{H} \mathbf{d}_{[t-1]} + (\gamma_{[t-1]} + \beta_{[t]}) \mathbf{d}_{[t-1]}^T \mathbf{H} \mathbf{d}_{[t-1]} = 0.$$

Using the previous relation 4.56 this implies that

$(\alpha_{[t]} \nabla_{N_{[t]}} \mathcal{L}^T + \beta_{[t]} \mathbf{d}_{[t-1]})^T \mathbf{H} \mathbf{d}_{[t-1]} = 0$, i.e. that the new direction is conjugate to the last one.

4.3.7 Adding second order information – using the curvature of the error

Using natural conjugate gradient is one way of introducing second order information of the error in the equation. Another possible direction to achieve this is the one proposed in [Le Roux and Fitzgibbon \(2010\)](#). In this section we will point to yet another approach.

One can easily see that in the generalized trust region framework, used to derive TONGA or natural gradient, one can rely on a second order Taylor expansion of the loss, instead of a first order expansion.

Let us consider TONGA first. We can rewrite the constrained optimization that we are attempting to solve as:

$$\begin{aligned} & \arg \min_{\Delta\theta} \mathcal{T}_2 \{ \mathcal{L}(\theta + \Delta\theta) \} \\ \text{s. t. } & \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}(\mathbf{u})} [\| \mathcal{L}(\theta + \Delta\theta) - \mathcal{L}(\theta) \|_2^2] = \text{const.} \end{aligned} \quad (4.57)$$

We can now take the same approximations as before of the expected change in the loss, which gives us the simplified constrained optimization:

$$\begin{aligned} & \arg \min_{\Delta\theta} \mathcal{L}(\theta) + \nabla \mathcal{L} \Delta\theta + \frac{1}{2} (\Delta\theta)^T \mathbf{H} \Delta\theta \\ \text{s. t. } & (\Delta\theta)^T \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}(\mathbf{u})} [(\nabla \mathcal{L})^T \nabla \mathcal{L}] \Delta\theta = \text{const.} \end{aligned} \quad (4.58)$$

If we apply Lagrange multipliers again, we get that the form of the solution for $\Delta\theta$ is the following:

$$\Delta\theta = -\nabla \mathcal{L} \left(\frac{1}{2} \mathbf{H} + \alpha \mathbf{U} \right)^{-1} \quad (4.59)$$

where $\alpha \in \mathbb{R}$ is a real number that comes from applying the Lagrange multipliers (and can be solved from additional constraints that result from this method). If we use an inequality constraint we get the same form for the solution (with some extra constraints on α). Instead of solving for α from these constraints, which can be complicated, we opt, as before, to consider α to be a hyper-parameter and update it heuristically.

The same derivation can be applied to Amari’s natural gradient, where we will get the Fisher Information Matrix \mathbf{F} instead of the uncentered covariance matrix \mathbf{U} . One can view these derivations as saying that both the covariance of gradients \mathbf{U} and the Fisher Information Matrix \mathbf{F} can serve as meaningful “structured” damping additive terms for the Hessian.

This method is different from [Le Roux and Fitzgibbon \(2010\)](#) which is attempting to use the Hessian of the true gradients, or from the natural conjugate gradient where we are attempting to use the Hessian of the loss with respect to the functional manifold.

Also, this formulation can be efficiently implemented in a truncated Newton framework. Any weighted sum between the Hessian and \mathbf{U} or \mathbf{F} can be efficiently computed by using the L and R operators. One simply sums the result of $\mathbf{U}\mathbf{x}$ or $\mathbf{F}\mathbf{x}$ times α with the result of $\mathbf{H}\mathbf{x}$, each computed as described in previous sections.

4.3.8 Mixing constraints

Finally we make the observation that within this trust region approach we can always add multiple constraints. By doing so, and solving the constraint optimization, we get the same generic form for the descent direction: the gradient times the inverse of some matrix. This matrix, however, will be a weighted sum of the different matrices produced by the different constraints.

In particular, if we are to combine Amari’s natural gradient descent with TONGA, we get:

$$\begin{aligned} & \arg \min_{\Delta\theta} \mathcal{T}_1 \{ \mathcal{L}(\theta + \Delta\theta) \} \\ \text{s. t. } & \mathbf{E}_{\mathbf{u} \sim \tilde{\pi}(\mathbf{u})} [\| \mathcal{L}(\theta + \Delta\theta) - \mathcal{L}(\theta) \|_2^2] = \text{const.} \\ & \text{and } \text{KL}(p_\theta \| p_{\theta + \Delta\theta}) = \text{const.} \end{aligned} \tag{4.60}$$

This gives the following step:

$$\Delta\theta = -\nabla\mathcal{L}(\mathbf{F} + \alpha\mathbf{U})^{-1} \tag{4.61}$$

We propose, as before, to heuristically take $\alpha \in \mathbb{R}$ and treat it as a hyper-parameter of the algorithm. Note that from the Lagrange multiplier method we get a scalar factor in front of both \mathbf{U} and \mathbf{F} . We can fold one of these two factors in to the learning rate. The parameter α in Equation (4.61) refers to the ratio of the factor in front of \mathbf{U} divided by the factor in front of \mathbf{F} .

We can also minimize a second order Taylor expansion of the loss. This results in a step of the following form:

$$\Delta\theta = -\nabla\mathcal{L} \left(\frac{1}{2}\mathbf{H} + \alpha_1\mathbf{F} + \alpha_2\mathbf{U} \right)^{-1} \tag{4.62}$$

where, again, $\alpha_1, \alpha_2 \in \mathbb{R}$ can be considered hyper-parameters. It is easy to see the combinatorial growth of the possible algorithms with the number of different possible constraints that one can use. All these algorithms can be implemented efficiently using the same pipeline as the Hessian-Free Optimization algorithm.

Exploring these alternatives amongst themselves is a hard task. It is to be expected that a single benchmark (or even multiple benchmarks) might not be

sufficient, as each method might have different weaknesses and strengths. A more solid approach might be to understand the weaknesses and strengths of each constraint and assume they compose in some simple way when used together. This would offer some intuitions for deciding the right subset of constraints one might need for a given task.

An even more proper avenue is to consider testing these algorithms on different “unit tests” that check their robustness with respect to particular structures in the error surface, as proposed in [Schaul et al. \(2014\)](#). Such a bottom up approach can prove to be extremely efficient in quickly determining which combinations make sense and when.

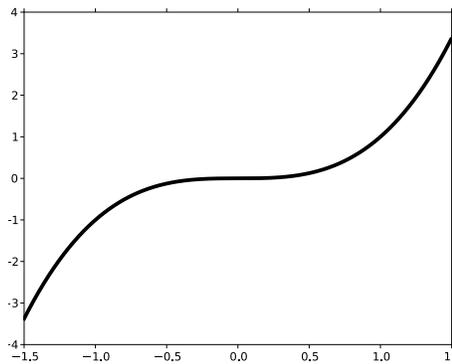
4.4 Saddle-points

The bottom up approach of “unit-testing” proposed in [Schaul et al. \(2014\)](#) can also be used to theoretically analyse these algorithms. In this section we will do this by asking how different algorithms behave for a specific such unit test, the saddle point. It represents a specific structure in the error surface. We start by explaining the properties of this structure as well as providing some motivation for why saddle points are very important when optimizing non-convex high-dimensional functions.

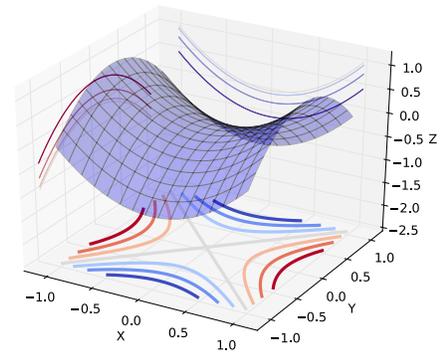
Understanding what happens around saddle points is useful as they represent one failure mode of “blindly” porting optimization techniques meant for the convex domain to the non-convex one. A descent direction for the second order approximation of a function (near a saddle point) is not necessarily a descent direction of the function itself. This is due to negative curvature, that makes the model move in an ascent direction. However, ignoring the negative direction (or damping it) also leads to a suboptimal step.

4.4.1 Understanding saddle points

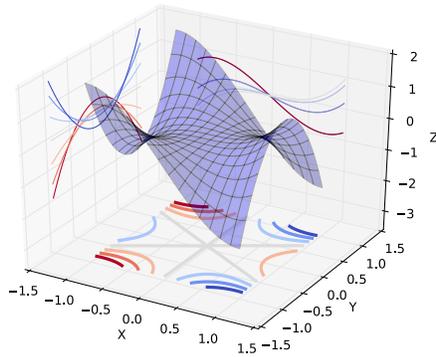
Saddle points are critical points of a function \mathcal{L} which are neither minima nor maxima. See, for example, the few illustrations provided in [Figure 4.4](#). Note that the Hessian is a symmetric matrix, and therefore its eigenvalues are real numbers.



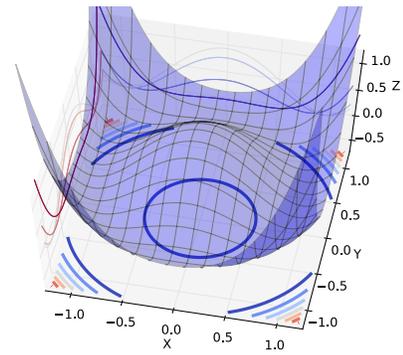
(a) Saddle point in 1-D given by x^3



(b) Classical saddle point in 2-D given by $x^2 - y^2$



(c) Monkey saddle given by $x^3 - 3xy^2$



(d) gutter structure given by $(x^2 + y^2 - 1)^2$

Figure 4.4: Illustrations of three different types of saddle points (a-c) plus a gutter structure (d). Note that for the gutter structure, any point from the circle $x^2 + y^2 = 1$ is a minimum. The shape of the function is like the bottom of a bottle of wine. This means that the minimum is now a ring instead of a single point. The Hessian is singular at any of these points.

The critical points can be categorized or identified based on the signs of these eigenvalues. Specifically, we know that :

1. If all eigenvalues of the Hessian are non-zero and positive, then the critical point is a local minimum. The Hessian is positive definite.
2. If all eigenvalues are non-zero and negative, then the critical point is a local maximum. The Hessian is negative definite.
3. If the eigenvalues are non-zero and we have both positive and negative eigenvalues, then the critical point is a saddle point with a *min-max* structure (see

Figure 4.4 (b)). That is, if we restrict the function \mathcal{L} to the subspace given by the eigenvectors corresponding to positive eigenvalues, then the saddle point is a minimum of this restriction. On the other hand, if we restrict \mathcal{L} to the subspace of the eigenvectors corresponding to negative eigenvalues, then the saddle point becomes a maximum of this restriction.

4. If the Hessian matrix is singular, then the *degenerate* critical point can be a saddle point, as it is, for example, for $x^3, x \in \mathbb{R}$ or for the monkey saddle (Figure 4.4 (a) and (c)). If it is a saddle, then, if we restrict θ to only change along the direction of singularity, the restricted function does not exhibit a minimum nor a maximum. We would only have a plateau. When moving from one side to other of the plateau, the eigenvalue corresponding to this picked direction changes sign, being exactly zero at the critical point. Note that an eigenvalue of zero can also indicate the presence of a gutter structure. A gutter is a connected set of points that are all either minima, maxima or saddle (depending on the rest of the eigenvalues). In the direction of the gutter, the function is constant. This structure can have the shape of a line or subspace if, for example, one or more coordinates do not affect the function at all. It can also have more interesting shapes.

In the enumeration above we make a distinction between gutters and plateaus. In this text, a plateau is an almost flat region in some direction. This structure is given by having the eigenvalues (which describe the curvature) corresponding to the directions of the plateau be *close to 0*, but *not exactly 0*. Or, additionally, by having a large discrepancy between the norm of the eigenvalues. This large difference would make the direction of “relatively” small eigenvalues look like flat compared to the direction corresponding to large eigenvalues. A gutter is the extreme case when the surface is perfectly flat and the eigenvalue is 0.

One simple way of analyzing (and understanding) nondegenerate critical points is by relying on Morse’s lemma (see, for example, chapter 7.3, Theorem 7.16 in Callahan (2010)). It states that locally to such a critical point there exists a change of coordinates such that the function can be rewritten as a sum of squares:

$$\mathcal{L}(\theta^* + \Delta\theta) = \mathcal{L}(\theta^*) - (\Delta\mathbf{v}_1)^2 - \dots - (\Delta\mathbf{v}_r)^2 + (\Delta\mathbf{v}_{r+1})^2 + \dots + (\Delta\mathbf{v}_{n_\theta})^2 \quad (4.63)$$

Note that $\Delta \mathbf{v}$ are the new coordinates and that we subtract r squares and add $n_\theta - r$ squares. r is the *index* (or *index of inertia*) of the nondegenerate critical point and is equal to the number of negative eigenvalues of the Hessian. By abuse of notation we will refer to the index of inertia also as the fraction of negative eigenvalues (and denote it by r as well).

This reparametrization provides a clear geometrical understanding of the landscape around the critical point. We can explore each coordinate of the reparametrized space in parallel. Along each dimension, the function has the shape of a bowl, where the critical point is either a minimum (if the bowl is concave up) or a maximum (if the bowl is concave down). It also shows that if all eigenvalues have the same sign that the critical point becomes a local minimum or local maximum of the function. Otherwise, it has the min-max structure and it is a saddle point. For a more in depth understanding of saddle points from a geometrical/mathematical perspective we recommend Chapter 7 of Callahan (2010).

Before going further in our analysis, one question we need to answer is: *Why should we look at saddle points? How common are they?* Some results on these questions come from statistical physics where the nature of critical points for random Gaussian error functions on high dimensional continuous domains are studied. See the seminal work of Bray and Dean (2007); Fyodorov and Williams (2007). The results presented in these works rely on the *replica theory*, a mathematical technique for analysing large dimensional systems with *quenched disorder* like spin glasses. A recent description of this technique is given in Parisi (2007).

Recall that the index of a critical point r is the fraction of negative eigenvalues of the Hessian, and let us denote the error obtained at the critical point by the error that we obtain \mathcal{L} . Any function will have a global minimum with $\mathcal{L} = \mathcal{L}_{\min}$ and $r = 0$ and a global maximum with $\mathcal{L} = \mathcal{L}_{\max}$ and $r = 1$. Bray and Dean (2007) counted the number of critical points of a random function in a finite volume of N dimensions within a range of error \mathcal{L} and index r . The authors found that any such function with large enough N , has an exponential number of critical points. If we project these points in the plane whose axes are given by \mathcal{L} (i.e. the amount of error that we have) and r , they are overwhelmingly likely to be located on a monotonically increasing curve $\mathcal{L}^*(r)$ that rises from \mathcal{L}_{\min} to \mathcal{L}_{\max} as r goes from 0 to 1. The probability of a critical point to be $O(1)$ away from the curve is exponentially small in the dimensionality N of the space.

This result states that most critical points that correspond to an error larger than \mathcal{L}_{\min} are highly likely to be saddle points and the larger the error \mathcal{L} is, the larger r becomes (we have more and more negative eigenvalues as the error at the critical point increases). This means that the values of all local minima are concentrated close to the value of the global minimum of the function.

Another way of understanding these findings is via random matrix theory. [Bray and Dean \(2007\)](#) states that the eigenvalue distribution of the Hessian follows a semi-circular law described by [Wigner \(1958\)](#), except that the semi-circle is shifted according to \mathcal{L} . In particular, for $\mathcal{L} = \mathcal{L}_{\min}$, the semi-circle is shifted so far to the right that all eigenvalues are positive. This means that, besides having most critical points be saddle points, for any such saddle point that corresponds to a reasonable large error there are sufficiently many directions of low curvature (many eigenvalues are very close to 0). This indicates the presence of a plateau like structure around the saddle point, plateau that can affect considerably stochastic gradient descent. [Fyodorov and Williams \(2007\)](#) findings are very similar.

In [Baldi and Hornik \(1989\)](#) the error surface of a single hidden layer MLP with linear units is analysed. The number of hidden units is assumed to be less than the number of inputs units. Such an error surface shows only saddle-points and *no* local minimum or local maximum. This result agrees with the observation made by [Bray and Dean \(2007\)](#). In fact, as long as we do not *get stuck* in the plateaus surrounding these saddle points, for such a model we are guaranteed to obtain the global minimum of the error. A similar observation is also made in [Saxe et al. \(2014\)](#), where the existence of symmetries in the weights of a deep linear feedforward models leads to saddle structures.

In [Saad and Solla \(1995\)](#) the dynamics of stochastic gradient descent are analysed for soft committee machines. A soft committee machine is a single nonlinear hidden layer MLP, whose output weights are all equal to 1 and there is no output activation function. The paper explores how well a student model can learn to imitate a teacher model which was randomly sampled. The approach taken is analytical, where differential equations are provided that describe the evolution of the learning dynamics. An important observation of this work is showing that learning goes through an initial phase of *being trapped in the symmetric subspace*. In other words, due to symmetries in the randomly initialized weights, the network has to traverse one or more plateaus that are caused by units with similar be-

haviour. Rattray et al. (1998); Inoue et al. (2003) provides further analysis, stating that the initial phase of learning is plagued with saddle point structures caused by symmetries in the weights.

Mizutani and Dreyfus (2010) looks at the effect of negative curvature for learning and implicitly at the effect of saddle point structures in the error surface. Their findings are similar. A proof is given where the error surface of a single layer MLP is shown to have saddle points (where the Hessian matrix is indefinite). Other small scale problems are also discussed such as the XOR problem.

4.4.2 Optimization algorithm and saddle points

One way of understanding Morse's lemma is to look at a second order Taylor expansion of the function \mathcal{L} around a critical point. If we assume that the Hessian is not singular, then there is a neighbourhood around this critical point where this approximation is reliable.

In the case of a singular Hessian, if the function contains higher order terms, then there is a direction in which the second order approximation is not sufficient. Namely, consider a restriction of θ to the direction given by the eigenvector corresponding to the zero eigenvalue. Being a critical point the first order derivative vanishes. By our assumption so does the second order derivative. Therefore the behaviour of the function \mathcal{L} in this projected subspace is described only by the higher order derivatives, and hence a second order approximation (which would predict a constant function) is not reliable for describing the dynamics of learning.

If the Hessian is not singular, given that the first derivative vanishes, the approximation of the function becomes:

$$\mathcal{L}(\theta^* + \Delta\theta) = \mathcal{L}(\theta^*) + \frac{1}{2}(\Delta\theta)^T \mathbf{H} \Delta\theta \quad (4.64)$$

Let us denote by $\mathbf{e}_{[1]}, \dots, \mathbf{e}_{[n_\theta]}$ the eigenvectors of \mathbf{H} and by $\lambda_{[1]}, \dots, \lambda_{[n_\theta]}$ the corresponding eigenvalues. We can now make a change of coordinates, by projecting $\Delta\theta$ on these eigenvectors, making them the new basis of our coordinate system.

$$\Delta\mathbf{v} = \frac{1}{2} \begin{bmatrix} \mathbf{e}_{[1]}^T \\ \dots \\ \mathbf{e}_{[n_\theta]}^T \end{bmatrix} \Delta\theta \quad (4.65)$$

$$\mathcal{L}(\theta^* + \Delta\theta) = \mathcal{L}(\theta^*) + \frac{1}{2} \sum_{i=1}^{n_\theta} \lambda_{[i]} (\mathbf{e}_{[i]}^T \Delta\theta)^2 = \mathcal{L}(\theta^*) + \sum_{i=1}^{n_\theta} \lambda_{[i]} \Delta\mathbf{v}_i^2 \quad (4.66)$$

This reparametrization looks similar to Morse’s lemma, with the exception that we leave the basis vectors to have norm $1/2$ for convenience and do not rescale them by the absolute value of the eigenvalues. Leaving the eigenvalues outside allows for a better understanding of the dynamics of learning around this nondegenerate critical point by different optimization techniques. Now the different coordinates (or tunable parameters) are independent of each other and can be analysed in parallel.

Note that looking at the behaviour around these critical points is useful, as the behaviour is not only relevant close to convergence. Due to the presence of saddle points, the behaviour around critical points is relevant in all stages of learning.

For *gradient descent* we can see that the gradient points in the right direction. Namely, if some eigenvalue $\lambda_{[i]}$ is positive, then we move towards θ^* in that direction because the restriction of \mathcal{L} to the corresponding eigenvector is $\mathcal{L}(\theta^*) + \lambda_{[i]} \Delta\mathbf{v}_i^2$, which has a minimum when $\mathbf{v}_i = 0$. On the other hand, if the eigenvalue is negative, then the gradient descent will move away from θ^* which is a maximum when restricting the loss function to the corresponding eigenvector of said eigenvalue.

The downside of gradient descent is the *scale* of the step along each eigenvector. The step we will take, for any direction $\mathbf{e}_{[i]}$, is given by $-2\lambda_{[i]} \Delta\mathbf{v}_i$. Because the gradients are proportional to the corresponding eigenvalues of the Hessian, the eigenvalue dictates how fast we move in each direction. Note that also, to avoid divergence, the learning rate has to be at most $1/|\lambda_{[max]}|$. Therefore, if there is a large discrepancy between eigenvalues, then gradient descent will have to take very small steps in some directions. This means that it might takes a very long time to move away from the critical point, if the critical point is a saddle point, or to the critical point if it is a minimum.

The *Newton method*, in its original form, solves the slowness problem by properly rescaling the gradients in each direction with the inverse of the corresponding eigenvalue. The step we take now is given by $-\Delta\mathbf{v}_i$. However, this approach can result in moving in the wrong direction. Specifically, if an eigenvalue is negative, then by multiplying with its inverse, the Newton method would change the sign of

the gradient along this eigenvector. Instead of taking the step away from θ^* in the direction of negative curvature (where θ^* is a maximum), Newton method moves towards θ^* . This effectively makes θ^* an *attractor* for the dynamics of the Newton method, making this algorithm converge to this unstable critical point. Therefore, while gradient descent might still escape saddle points in finite time, the Newton method might not be able to.

Trust regions are one standard approach for using a second order method on a non-convex problem. It relies on damping the Hessian in order to remove the negative curvature. As discussed before, damping the Hessian by adding the identity matrix times some constant α is equivalent to adding α to each of the eigenvalues of the Hessian. That is, we now rescale the gradients by multiplying them with $1/\lambda_{[i]}+\alpha$, resulting in the step $-\left(\lambda_{[i]}/\lambda_{[i]}+\alpha\right)\Delta\mathbf{v}_i$. In particular, to deal with negative curvature, one usually increases the damping coefficient α enough such that even for the most negative eigenvalue $\lambda_{[min]}$ we have $\lambda_{[min]}+\alpha>0$. We can see now that because the denominators are all positive, the trust region method does not change the sign in any of the possible directions of the gradient, and, hence, it will move in a descent direction.

The downside is, again, the size of step along each eigenvector is not optimal. In principle, if $\lambda_{[min]}+\alpha\approx 0$, then, in this direction, the gradient will end up being rescaled to a very large value. By the same argument as before, to avoid divergence, the learning rate must now be very small and the algorithm might take a long time to escape the saddle point. The same is true even if we make $\lambda_{[min]}+\alpha$ comfortably large (say equal to $|\lambda_{[min]}|$). We will have, in this case, $\lambda_{[max]}+\alpha\gg\lambda_{[max]}$, meaning that we will move very slow in the direction given by the largest positive eigenvalue. This mismatch of the step with the optimum step size becomes very significant when the difference between the most positive and most negative eigenvalue is large. One way of understanding this behaviour is by realizing that, in the presence of negative curvature, the second order approximation of the function becomes unreliable. To deal with this issue, a trust region method has to drastically decrease the radius of the trust region (increasing the damping factor α) and effectively throwing away any second order information. In such an extreme case this second order method reduces to a first order one.

Besides damping, another approach to deal with negative curvature is to ignore them. For example, we could stop the linear solver as soon as it starts incorporat-

ing directions of negative curvature when using a truncated Newton method (by checking if the step we would take still results in reducing the error). BFGS, which is a popular choice for scaling up the Newton method, deals with negative curvature by damping the Hessian, but also by ignoring directions of negative curvature (see (Nocedal and Wright, 2006, chapter 6.1)). Ignoring these directions means that we will only approach the saddle point from the direction of positive curvature, without being able to escape it.

Another approach for speeding convergence is natural gradient descent, which relies on the Fisher Information Matrix instead of the Hessian. The Fisher Information Matrix is positive definite by construction, therefore one does not need to deal with negative curvature explicitly. It is argued in Rattray et al. (1998); Inoue et al. (2003) that natural gradient descent can address certain saddle point structures effectively. Specifically, if we have different units behaving in a very similar manner, then the Fisher matrix will be close to singular. By taking a natural gradient step, we move much further in the directions in which the matrix is almost singular. This will force the model to differentiate the behaviour of those units much faster than gradient descent. If the model is initialized such that the Fisher matrix is not singular, the algorithm will also not approach regions of singularity, and therefore it will avoid the saddle points hidden in these regions. We argue that this might be another argument in favor of the recent success of the Hessian Free Optimization algorithm. As it was shown in Section 4.3.2, we know that HF corresponds to a natural gradient algorithm.

Mizutani and Dreyfus (2010) argues that natural gradient (or more exactly the Gauss-Newton method) also suffers from negative curvature. In this work they look at least square problems, for which the well studied Gauss-Newton method is defined, and where natural gradient descent coincides with this method. One particular known issue of the Gauss-Newton algorithm happens in the over-realizable regime, when the model is over complete (i.e. the model has a lot more parameters than it actually needs to model the task at hand). In this situations, while there exists a stationary solution θ^* , the Fisher matrix around θ^* is rank deficient. Numerically, this means that the Gauss Newton direction can be (close to) orthogonal to the gradient at some distant point from θ^* (Mizutani and Dreyfus, 2010). A line search in this direction could fail and it might lead to the algorithm converging to some non-stationary point.

Another weakness of natural gradient is that the *residual* \mathbf{S} defined as the difference between the Hessian and the Fisher Information Matrix can be large close to a saddle point when the function is highly nonlinear and its Hessian exhibits large negative eigenvalues. In this case, the landscape close to the critical point can be dominated by \mathbf{S} . This implies a large discrepancy between the Hessian and the Fisher Information Matrix, and therefore the rescaling provided by the Fisher Matrix has to be suboptimal as it will not match the eigenvalues of the Hessian.

In the case of TONGA (Section 4.3.3) a similar observation holds as for natural gradient. Because TONGA uses the gradients of the loss \mathcal{L} , which vanishes at a critical point, the matrix is more likely to end up being close to singular when we approach any such critical point, even if the Hessian might not be. In such a case the rescaling provided the TONGA is not the right one. We can directly use the reformulation of our function close to the critical point to show this. Note that by TONGA we refer to the algorithm that uses \mathbf{U} , the uncentered covariance of gradients, as the matrix that needs to be inverted. In particular we are not talking about the approximation used for computing the inverse of \mathbf{U} proposed in Le Roux et al. (2008).

Let us look at some neighbourhood around a critical point θ^* . We can now rely on the second order approximation of the function, and re-write this approximation in the coordinate system given by the eigenvectors of the Hessian, see Equation (4.66). In this system of coordinates each coordinate is independent.

If we assume further that the different examples in the mini-batch, when rewritten in the coordinate system given by $\Delta\mathbf{v}$, fall according to an isotropic Gaussian distribution of standard deviation β^2 , and also each of these examples are within the neighbourhood where our second order approximation makes sense then

$$\mathbf{E}_{\Delta\mathbf{v}_i \sim \mathcal{N}(0, \beta^2)} [(\lambda_{[i]} \Delta\mathbf{v}_i)^2] = \beta^2 \lambda_{[i]}^2.$$

Therefore by using this matrix, we get that we rescale the gradients, in each direction, with $1/\beta^2 \lambda_{[i]}^2$. This means that the rescaled gradient will still be a function of $\lambda_{[i]}$, and hence non-optimal. In practice, our assumption that $\Delta\mathbf{v}_i \sim \mathcal{N}(0, \beta^2)$ will not hold and the distance from the critical point will play a role in the makeup of \mathbf{U} .

4.5 Addressing the saddle point problem

In order to address the saddle point problem let us look for a solution within the family of generalized trust region methods. We know that the sign of the eigenvalues of the Hessian can result in flipping the direction of the gradient along the corresponding eigenvector. Therefore it is safe to assume that we want to consider a generalized trust region method that minimizes a first order approximation of the loss under some constraint. This allows for a formula that is not in terms of the Hessian.

Indeed we want the curvature information to come from the constraint that we use, but we do want the algorithm to be such that we only use the curvature to decide on the step length (not the direction).

4.5.1 Constraining the change in the gradient – squared Newton method

The Hessian is a measure of change in the gradient. We argue that, within a generalized trust region approach, the trust region should indicate how much we can trust the first order approximation of the cost \mathcal{L} . That is, when the first order approximation deteriorates in some direction, we want to shrink the trust region in said direction, and if the first order approximation is very reliable in some other direction, we want to increase the trust region in that direction.

Given that the first order Taylor approximation of \mathcal{L} is given by projecting the step $\Delta\theta$ on the gradient, one can argue that as long as the gradient does not change when we add to the parameters some value $\Delta\theta$, the first order approximation is reliable. That is, if $\left.\frac{\partial\mathcal{L}}{\partial\theta}\right|_{\theta}$ is about the same as $\left.\frac{\partial\mathcal{L}}{\partial\theta}\right|_{\theta+\Delta\theta}$, then the first order approximation of \mathcal{L} at θ is about the same as the first order approximation of \mathcal{L} at $\theta + \Delta\theta$.

Assume now that we are to take a small steps when we are at θ . Because the gradient does not change, the subsequent step, after this first initial small step, will point in the same direction as the previous one. Therefore we are better off to move further when the gradient does not change. We fix a maximal change in the gradient that we allow, and define the trust region based on this maximal change. Formally this translates into:

$$\begin{aligned} & \arg \min_{\Delta\theta} \mathcal{T}_1 \{ \mathcal{L}(\theta + \Delta\theta) \} \\ \text{s. t. } & \| \nabla \mathcal{L}(\theta + \Delta\theta) - \nabla \mathcal{L}(\theta) \|_2^2 = \text{const.} \end{aligned} \quad (4.67)$$

We can expand our constraint by taking a first order approximation of $\nabla \mathcal{L}(\theta + \Delta\theta)$ which gives:

$$\nabla \mathcal{L}(\theta + \Delta\theta) - \nabla \mathcal{L}(\theta) \approx \nabla \mathcal{L}(\theta) + \mathbf{H}\Delta\theta - \nabla \mathcal{L}(\theta) = \mathbf{H}\Delta\theta \quad (4.68)$$

This means that our constraint translates into:

$$\| \nabla \mathcal{L}(\theta + \Delta\theta) - \nabla \mathcal{L}(\theta) \|_2^2 = (\Delta\theta)^T \mathbf{H}^T \mathbf{H} \Delta\theta = \text{const} \quad (4.69)$$

From this we get a solution of the form:

$$\Delta\theta = -\nabla \mathcal{L} (\mathbf{H}^T \mathbf{H})^{-1} \quad (4.70)$$

One intuitive way of understanding this approach, is that we consider a ball, whose radii are scaled according to the eigenvalues of $\mathbf{H}^T \mathbf{H}$, which correspond to the squared eigenvalues of \mathbf{H} . The radii of our trust region reflect, therefore, the local curvature of the function.

The disadvantage of this formulation is that we rely on the squared eigenvalues of the Hessian when defining the radii of the trust region. It is easy to show that close to a critical point, once we do a change of coordinates based on the eigenvectors of the Hessian, this method will use the square of the eigenvalues to scale the gradient in each direction. That means that in each direction will have the gradient of form $-\left(\frac{1}{\lambda_{[i]}}\right) \Delta \mathbf{v}_i$, which is inverse proportional to the norm of the eigenvalues (though it preserves the right sign). Because of this inverse proportionality, when we have a big discrepancy between the largest and smallest eigenvalue, the model will be forced to move too slowly in some directions, resulting in a suboptimal step.

One advantage of this approach is that, as with the previous algorithms, it can be efficiently implemented in a truncated Newton approach. To compute the

product $\mathbf{H}^T \mathbf{H} \mathbf{x}$, we just apply twice the formula for computing $\mathbf{H} \mathbf{x}$.

4.5.2 Limit the influence of second order terms – Saddle-Free Newton Method

The observations made for different optimization techniques states that, close to nondegenerate critical points, what we want to do is to rescale the gradient in each direction $\mathbf{e}_{[i]}$ by $1/|\lambda_{[i]}|$. This achieves the same rescaling as the Newton method, while preserving the sign of the gradient. This means that if the gradient says that we should move away from θ^* , the rescaled step will still move away. Saddle points are not attractors of the dynamics of this approach, as they are to the dynamics of the Newton method.

The idea of taking the absolute value of the eigenvalue was briefly suggested before, see for example in [Nocedal and Wright \(2006, Chapter 3.4\)](#) or in [Murray \(2010, Chapter 4.1\)](#). However, we *are not aware* of any proper justification of this algorithm or even a proper detailed exploration (empirical or otherwise) of this idea.

The problem is that one cannot simply replace \mathbf{H} by $|\mathbf{H}|$, where $|\mathbf{H}|$ is the matrix obtained by taking the absolute value of each eigenvalue of \mathbf{H} , with out proper justification. For example, one obvious question is: are we still optimizing the same function? While we might be able to argue that we do the right thing close to critical points, can we do the same far away from these critical points? In what follows we will provide such a justification for replacing \mathbf{H} with $|\mathbf{H}|$ by employing the generalized trust region framework. Namely we want to define k and d in the following equation, such that, when solving this constrained optimization using Lagrange multipliers, we get back $\Delta\theta = -\nabla\mathcal{L}|\mathbf{H}|^{-1}$:

$$\begin{aligned} \Delta\theta &= \arg \min_{\Delta\theta} \mathcal{T}_k\{\mathcal{L}, \theta, \Delta\theta\} \quad \text{with } k \in \{1, 2\} \\ \text{s. t. } &d(\theta, \theta + \Delta\theta) \leq r \end{aligned} \tag{4.71}$$

We first note that k must be 1. If k is 2, then the step will be a function of \mathbf{H} rather than $|\mathbf{H}|$. Having $k = 1$ also makes sense because we know that a second order approximation is *not reliable* when we have negative curvature. Next, we need to design a distance measure d such that it will produce $|\mathbf{H}|$.

The matrix $|\mathbf{H}|$ is a measure of the local curvature of the loss \mathcal{L} . We therefore

want, similar to the Squared Newton method proposed in Section 4.5.1, to define the radius of the trust region according to the curvature of the function.

To achieve this, the Squared Newton method looked at the change in the gradient of the loss \mathcal{L} . The gradient is a ratio of the change in \mathcal{L} divided by the change in the parameter, when $\Delta\theta \rightarrow 0$. The nature of this ratio is different from that of the loss function. One can become aware of this by assigning units to the loss function¹. If so, the constraints end up being expressed in different units compared to the first order Taylor approximation of the loss. This is a sign that there is a rescaling term missing that is a function of $\Delta\theta$ between the constraint and the function we need to minimize. We end up ignoring this rescaling factor (by assuming that it is constant with respect to $\Delta\theta$) when we apply the Lagrange multipliers method.

In other words, the change in the gradients does not tell us how far from θ we can assume \mathcal{L} to have the about the same first order approximation, but rather how fast the first order of \mathcal{L} would change per ϵ change in the parameter, which is not a reliable measure of trustworthiness for the trust region.

The proper question to ask is how far from θ can we trust our first order approximation of \mathcal{L} . One measure of this trustfulness is given by how much the second order term of the Taylor expansion of \mathcal{L} influences the value of the function at some point $\theta + \Delta\theta$. That is we want the following constraint to hold:

$$\begin{aligned}
 d(\theta, \theta + \Delta\theta) &= |\mathcal{T}_2\{\mathcal{L}, \theta, \Delta\theta\} - \mathcal{T}_1\{\mathcal{L}, \theta, \Delta\theta\}| \\
 &= |\mathcal{L}(\theta) + \nabla\mathcal{L}\Delta\theta + \frac{1}{2}\Delta\theta^T\mathbf{H}\Delta\theta - \mathcal{L}(\theta) - \nabla\mathcal{L}\Delta\theta| \\
 &= \frac{1}{2}|\Delta\theta^T\mathbf{H}\Delta\theta| \leq r
 \end{aligned} \tag{4.72}$$

where $\nabla\mathcal{L}$ is the partial derivative of \mathcal{L} with respect to θ and $r \in \mathbb{R}$ is some small value that indicates how much discrepancy we are willing to accept between our first order approximation of \mathcal{L} and the second order approximation of \mathcal{L} .

Note that the distance measure d takes into account the curvature of the function. It uses the curvature to decide how far from θ we have that its first order approximation is still reliable. If we have high curvature in some direction, we expect the corresponding radius of the trust region to be small and if have low curvature, the radius will be larger.

1. This suggestion was made by Surya Ganguli.

The proposed distance, however, does not easily allow to solve for $\Delta\theta$ in more than one dimension. If we take the square of the norm to remove the absolute value, we get a function that is quartic in $\Delta\theta$ (the term is raised to the power 4). To address this problem we rely on the following Lemma:

Lemma 5. *Let \mathbf{A} be a nonsingular square matrix in $\mathbb{R}^n \times \mathbb{R}^n$, and $\mathbf{x} \in \mathbb{R}^n$ be some vector. Then it holds that $|\mathbf{x}^T \mathbf{A} \mathbf{x}| \leq \mathbf{x}^T |\mathbf{A}| \mathbf{x}$, where $|\mathbf{A}|$ is the matrix obtained by taking the absolute value of each of the eigenvalues of \mathbf{A} .*

Proof. Let $\mathbf{e}_{[1]}, \dots, \mathbf{e}_{[n]}$ be the different eigenvectors of \mathbf{A} and $\lambda_{[1]}, \dots, \lambda_{[n]}$ the corresponding eigenvalues. We now re-write the identity by expressing the vector \mathbf{x} in terms of these eigenvalues:

$$\begin{aligned} |\mathbf{x}^T \mathbf{A} \mathbf{x}| &= \left| \sum_i (\mathbf{x}^T \mathbf{e}_{[i]}) \mathbf{e}_{[i]}^T \mathbf{A} \mathbf{x} \right| \\ &= \left| \sum_i (\mathbf{x}^T \mathbf{e}_{[i]}) \lambda_{[i]} (\mathbf{e}_{[i]}^T \mathbf{x}) \right| \\ &= \left| \sum_i \lambda_{[i]} (\mathbf{x}^T \mathbf{e}_{[i]})^2 \right| \end{aligned}$$

We can now use the triangle inequality $|\sum_i x_i| \leq \sum_i |x_i|$ and get that

$$\begin{aligned} |\mathbf{x}^T \mathbf{A} \mathbf{x}| &\leq \sum_i |(\mathbf{x}^T \mathbf{e}_{[i]})^2 \lambda_{[i]}| \\ &= \sum_i (\mathbf{x}^T \mathbf{e}_{[i]}) |\lambda_{[i]}| (\mathbf{e}_{[i]}^T \mathbf{x}) \\ &= \mathbf{x}^T |\mathbf{A}| \mathbf{x} \end{aligned}$$

□

Lemma 5 shows that

$$d(\theta, \theta + \Delta\theta) = |\Delta\theta^T \mathbf{H} \Delta\theta| \leq \Delta\theta^T |\mathbf{H}| \Delta\theta$$

so we enforce our constraint on this upper bound of the distance, instead of the distance directly, resulting in the following generalized trust region method:

$$\begin{aligned} \Delta\theta &= \arg \min_{\Delta\theta} \mathcal{L}(\theta) + \nabla\mathcal{L}\Delta\theta \\ \text{s. t. } \Delta\theta^T|\mathbf{H}|\Delta\theta &\leq r \end{aligned} \tag{4.73}$$

Note that, as was discussed before when we introduced natural gradient, the inequality constraint can be turned into an equality one as the first order approximation of \mathcal{L} has a minimum at infinity, which means that the step will always be on the boundary of the trust region. We can use the Lagrange multipliers method, which gives us:

$$\Delta\theta = -\nabla\mathcal{L}|\mathbf{H}|^{-1} \tag{4.74}$$

As before, we do not solve for the Lagrange coefficient in terms of r , but rather fold it into the learning rate for which we carry out a line search. The resulting algorithm has the desired behaviour around critical points, where it uses the right step size (as predicted by the Newton method) while also being able to escape saddle points. That is, if we go back to the approximation of the function near a critical point proposed in Equation (4.66), this method will move on each coordinate by $-\mathbf{v}_i$, which is the optimal speed according to the curvature of the function.

Far away from a critical point, the method also moves in the right direction because of its justification as a generalized trust region method. Namely, far away from the critical point, the method defines a neighbourhood in which the first order approximation of \mathcal{L} is reliable and minimizes this approximation within this neighbourhood. This means that we always follow a descent direction of \mathcal{L} . We call this algorithm *Saddle-Free Newton method*.

The description of the algorithm suggest that it should behave very well in practice. In fact, if we do not have negative curvature, the algorithm converts into the Newton method. This makes the algorithm ideal for compact models, where we can get close to fully compute the whole Hessian and do an eigen decomposition of this matrix. For example, recurrent networks discussed in the next chapters are compact models.

In general, however, the difficulty of this proposed approach is in scaling it up. Specifically, the standard pipeline employed by HF can not be directly applied because there is no efficient way of computing $|\mathbf{H}|\mathbf{x}$. The R and L operators can not yield this computation.

One approach of approximating this method is to rely on the Squared Newton method introduced previously. If we assume that all eigenvalues of the Hessian are clustered around the same value, then we can use the following identity:

$$|\mathbf{H}|_{\mathbf{x}} \approx \frac{1}{|\lambda_{[max]}|} \mathbf{H}^T \mathbf{H} \mathbf{x} \quad (4.75)$$

The value of $\lambda_{[max]}$ can easily be approximating using the Power method. One can also view this approach as using a specific per iteration scaling of the matrix $\mathbf{H}^T \mathbf{H}$. Normally this scaling would fold back into the learning rate, but if we use damping for this matrix by adding some other matrix after we rescaled $\mathbf{H}^T \mathbf{H}$, then the rescaling becomes important.

In particular, the Fisher Information Matrix is believed to approximate well the Hessian while being positive definite. By relying on the justification in Section 4.3.8, we could use the Fisher Information Matrix to damp the squared Newton (where we rescale $\mathbf{H}^T \mathbf{H}$). This could lead to minimizing even more the difference between this computed matrix and the matrix $|\mathbf{H}|$ corresponding to taking the absolute value of the eigenvalues of the Hessian. The additive term from the Squared Newton should help natural gradient descent when the FIM matrix becomes singular (due to negative curvature) in some direction while the Hessian is not. The advantage of such an approach is that it is efficient to compute in the framework introduced for the Hessian-Free Optimization.

More principled approaches might also be possible. We regard the problem of scaling up Saddle-Free Newton as a future research direction. At this point, this thesis will only introduce the algorithm from a theoretical point of view, and argue that this algorithm takes an optimal step near a critical point. In the next section we will also demonstrate the effectiveness of the algorithm on a small scale experiment where we can afford to compute the full Hessian.

4.5.3 Benchmark

Small scale experiments

In this section we run experiments on a scaled down version of MNIST, where each input image is rescaled to be of size 10×10 . This rescaling allows us to construct models that are small enough such that we can implement the exact

Newton method and Saddle-Free Newton method, with out relying on any kind of approximations.

As a baseline we also consider mini-batch stochastic gradient descent, the *de facto* optimization algorithm for most neural networks. We additionally use momentum to improve the convergence speed of this algorithm. The hyper-parameters of mini-batch SGD – the learning rate, mini-batch size and the momentum constant – are chosen using random search (Bergstra and Bengio, 2012). We pick the best configurations from approximately 80 different choices. The learning rate and momentum constant are sampled on a logarithmic scale, while the mini-batch size is sampled from the set $\{1, 10, 100\}$. The best performing hyper-parameter values for SGD are provided in Table 4.1.

The Damped Newton method is a trust region method where we damp the Hessian \mathbf{H} by adding the identity times the damping factor. No approximations are used in computing the Hessian or its inverse (beside numerical inaccuracy due to machine precision). For the Saddle-Free Newton we also damp the matrix $|\mathbf{H}|$, obtained by taking the absolute value of the eigenvalues of the Hessian. At each step, for both methods, we pick the best damping coefficient from the following values: $\{10^0, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$. We do not perform an additional line search for the step size, but rather consider a fixed learning rate of 1. Note that by searching over the damping coefficient we implicitly search for the optimum step size as well. These two methods are run in batch mode. The results of these experiments are shown in Figure 4.5. Figure 4.5 (a) shows the minimum training error reached by different algorithms as a function of the model size. The plot provides empirical evidence that, as the dimensionality of the problem increases, the number of saddle points also increases (exponentially so). We argue that for the larger model (50 hidden units), the likelihood of an algorithms such as SGD or Newton method to stop near a saddle point becomes higher (as the number of saddle points is much larger) and therefore we should see these algorithms perform worse in this regime. The plot confirms our intuition. We see that for the 50 hidden units case, Saddle-Free outperforms the other two methods considerably.

Figure 4.5 (b) depicts the training error versus the number of epochs¹ that the model already went through. This plot suggests that Saddle-Free behaves well not

1. An epoch measures the number of steps it takes for the model to see all the examples in the training set once.

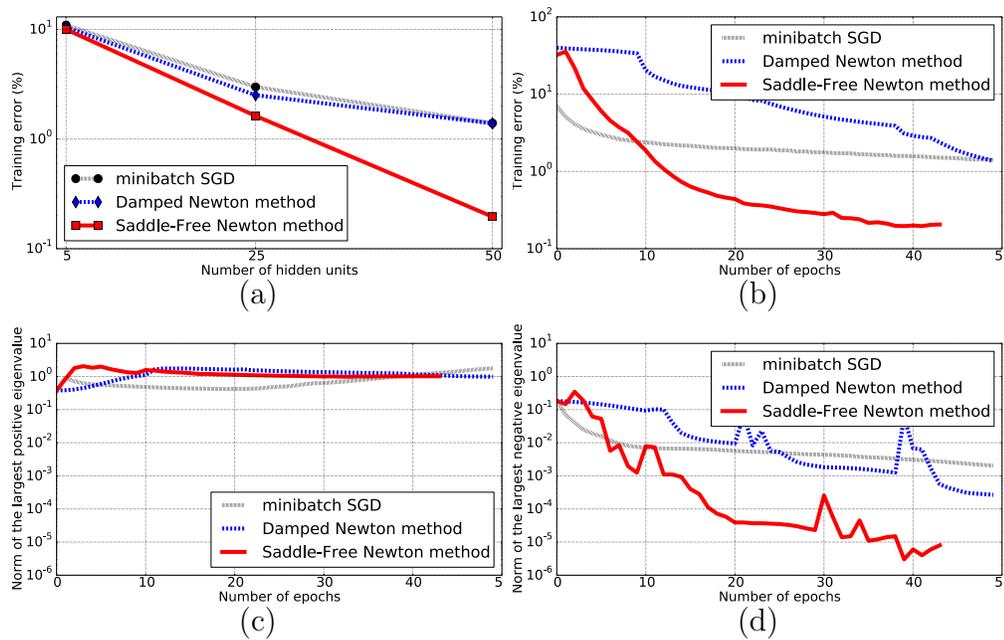


Figure 4.5: Empirical evaluation of different optimization algorithms for a single layer MLP trained on the rescaled MNIST dataset. In (a) we look at the minimum error obtained by the different algorithms considered as a function of the model size. (b) shows the optimum training curve for the three algorithms. The error is plotted as a function of the number of epochs (passes through the entire dataset). (c) looks at the evolution of the norm of the largest positive eigenvalue of the Hessian and (d) at the norm of the largest negative eigenvalue of the Hessian.

Model size	learning rate	momentum constant	mini-batch size
5 units	0.074	0.031	10
25 units	0.040	0.017	10
50 units	0.015	0.254	1

Table 4.1: Best performing hyper-parameters for stochastic gradient descent.

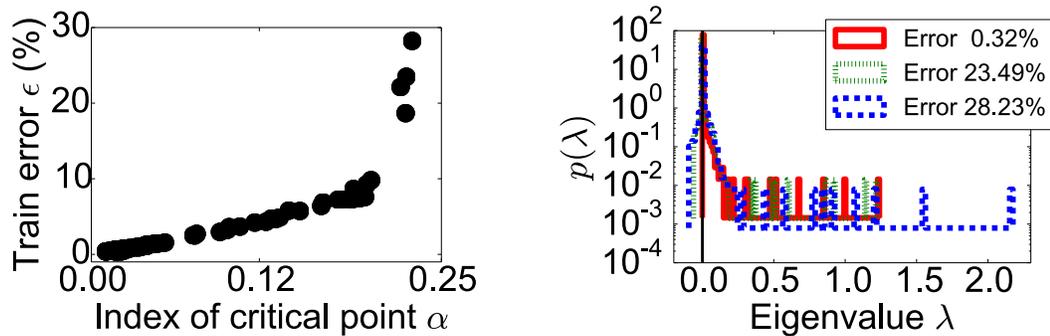


Figure 4.6: The plot on the left depict the training error versus the index (fraction of negative eigenvalues) for different critical points found nearby the path taken by different runs of the Saddle-Free Newton method. The critical points are discovered using the Newton method. The plot on the right shows the distribution of eigenvalues of the Hessian for three different critical points selected based on their error. Note that the y-axis is on a log scale.

only near a critical point but also far from them, taking reasonable large steps.

In Figure 4.5 (c) we look at the norm of the largest positive eigenvalue of the Hessian as a function of the number of training epochs for different optimization algorithms. Figure 4.5 (d) looks in a similar fashion at the largest negative eigenvalues of the Hessian. Both these quantities are approximated using the Power method. The plot clearly shows that initially there is a direction of negative curvature (and therefore we are bound to go near saddle points). The norm of the largest negative eigenvalue is close to that of the largest positive eigenvalue initially. As learning progresses, the norm of the negative eigenvalue decreases. For stochastic gradient descent and Damped Newton method, however, even at convergence we still have reasonably large negative eigenvalues, suggesting that we have actually stalling to a saddle point rather than a local minimum. For Saddle-Free Newton method the value of the most negative eigenvalue decreases considerably, suggesting that we are more likely to have converged to an actual local minimum.

Figure 4.6 is an empirical evaluation of whether the properties predicted by Bray and Dean (2007) for random Gaussian error functions hold for neural networks.

To obtain this plot we used the Newton method to discover nearby critical points along the path taken by the Saddle-Free Newton algorithm. We consider 20 different runs of the Saddle-Free algorithm, each using a different random seed. We then run 200 jobs. The first 100 jobs are looking for critical points near the value of the parameters obtained after some random number of epochs (between 0 and 20) of a randomly selected run (among the 20 different runs) of Saddle-Free Newton method. To this starting position uniform noise is added of small amplitude (the amplitude is randomly picked between the different values $\{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}\}$). The last 100 jobs look for critical points near uniformly sampled weights (the range of the weights is given by the unit cube). The model used is the 50 hidden units MLP, and the dataset is the same rescaled version of MNIST.

In Figure 4.6, the plot on the left shows the index of the critical point (fraction of negative eigenvalues of the Hessian at the critical point) versus the training error that it obtains. This plot shows that all critical points, projected in this plane, align in a monotonically decreasing curve, as predicted by the theoretical results on random Gaussian error functions (Bray and Dean, 2007). Empirically, it also seems that there are many more critical points for small index values, compared to intermediate values of the index.

The plot on the right looks at the distribution of the eigenvalues of the Hessian at 3 different critical points picked according to the error that they realise. Note that the plot is on a log scale on the y-axis. These distributions *do not follow* the semicircle rule, as predicted by the theory of random matrices. This is probably caused by the structure of the neural network (and of the task). However, the generic observation of (Bray and Dean, 2007), that as the error decreases the distribution shifts to the right (except the peak around 0) seems to hold. The fact that we have a large number of eigenvalues at 0, and a few eigenvalues that are sufficiently large suggest that any of these saddle-points are surrounded by plateaus, in which the different algorithms might end up taking a suboptimal step.

Larger scale experiments

Next, we look at a larger scale problem. Our goal is to provide some empirical evidence for the intuitions we put forward for each of the algorithms discussed in

previous sections. In particular, we are interested in comparing natural gradient descent with natural conjugate gradient (as formulated by [Honkela et al. \(2008\)](#); [Gonzalez and Dorrnsoro \(2006\)](#)) where we implement both algorithms with the same pipeline. We also want to show that our proposed improvement of natural conjugate gradient in Section 4.3.6 does better than the standard natural conjugate gradient algorithm that relies on the Polak–Riebiere formula to compute the new conjugate direction.

In our analysis we also consider TONGA, the Squared Newton method introduced in Section 4.5.1 and the effect of considering a second order approximation of the loss, Section 4.3.7, or mixing multiple constraints in a generalized trust region method, Section 4.3.8.

We carry out our benchmark on the Curves dataset, using the 6 layer deep auto-encoder from [Martens \(2010\)](#). The dataset is still somewhat small, it has only 20K training examples of 784 dimensions each. This allows to run all algorithms in batch mode with out running into memory issues. However the model that we use is of reasonable size.

All methods except SGD use the truncated Newton pipeline proposed for Hessian-Free Optimization [Martens \(2010\)](#). In particular, we use warm restart (where the previous solution is scaled by .8) for linear CG. We use the same stopping criterion and a Jacobi preconditioner ([Chapelle and Erhan, 2011](#)). To find the optimum step size we do a line search, and the matrix that has to be inverted is damped. The damping coefficient is adapted based on the Levenberg-Marquardt heuristic. Its initial value is set to 3 for all algorithms.

The benchmark is run on Tesla M2070 Nvidia cards, using Theano ([Bergstra et al., 2010](#)) for cuda kernels. We rely on `scipy.optimize.fmin_cobyla` as the off-the-shelf optimizer used for natural conjugate gradient to simultaneously find the right step size and new conjugate direction (see Section 4.3.6). For both implementation of natural conjugate gradient (using the off-the-shelf optimizer or the Polak–Riebiere formula) we forcefully reset the conjugate direction to the natural gradient after three steps (value obtained based on cross validation).

For SGD we use a batch size of 100 examples. The optimum learning rate, obtained by a grid search, is 0.01. We do not anneal the learning rate and we do not use any additional enhancements like momentum.

Note that while we show squared error (to be consistent with [Martens \(2010\)](#))

we are actually minimizing the cross-entropy during training (i.e. the gradients are computed based on the cross-entropy cost, square error is only computed for visualization).

The first observation that we can make is that natural conjugate gradient, which adds second order information to natural gradient, does perform better than natural gradient (and it seems to outperform, in terms of time, SGD as well). In particular, NatCG-L is doing particularly well. This algorithm relies on an off-the-shelf solver (in this case COBYLA) to find both the right step size and the next conjugate direction as we proposed in Section 4.3.6. This provides evidence supporting our hypothesis that relying on the Polak–Ribiere formula when implementing natural conjugate gradient (even if we reset the direction often) is harmful in practice. To apply any of these formula one needs to compute the inner products between vectors belonging to different tangential spaces. If we ignore this fact and assume the metric does not change from one step to another (as NatCG-F does), the assumption will hurt learning. The metric matrix stays about the same only if one takes smallish steps.

The second observation is that natural gradient descent shows flat regions (albeit small in terms of steps). For example look at the region around the training error of 16, or even around the training error of 11. We hypothesize that these flat regions might be evidence that the algorithm is taking a suboptimal step close to some saddle point that forces the Fisher Information matrix to be close to singular.

The third observation regards the Squared Newton method we proposed in Section 4.5.1. As discussed in our analysis, because it rescales the gradient in each direction by one over the squared eigenvalues of the Hessian, this algorithm always takes a suboptimal step. This is reflected in practice, where the algorithm underperforms. However, the curve for this algorithm also shows the least amount of flat regions (at least in the runs examined for constructing this plot). Flat regions of the training curve can be caused by the algorithm being trapped close to some saddle point. This can indicate that while there might be a saddle points close to which the Fisher Information matrix is close to singular (forcing the algorithm to take smaller steps), the Hessian might be well behaved in terms of the norm of the eigenvalues. Therefore the matrix $\mathbf{H}^T\mathbf{H}$ never gets close to singular and we do not end up stalling in the plateau around the saddle point.

We can couple this observation with the behaviour of mixing the Squared New-

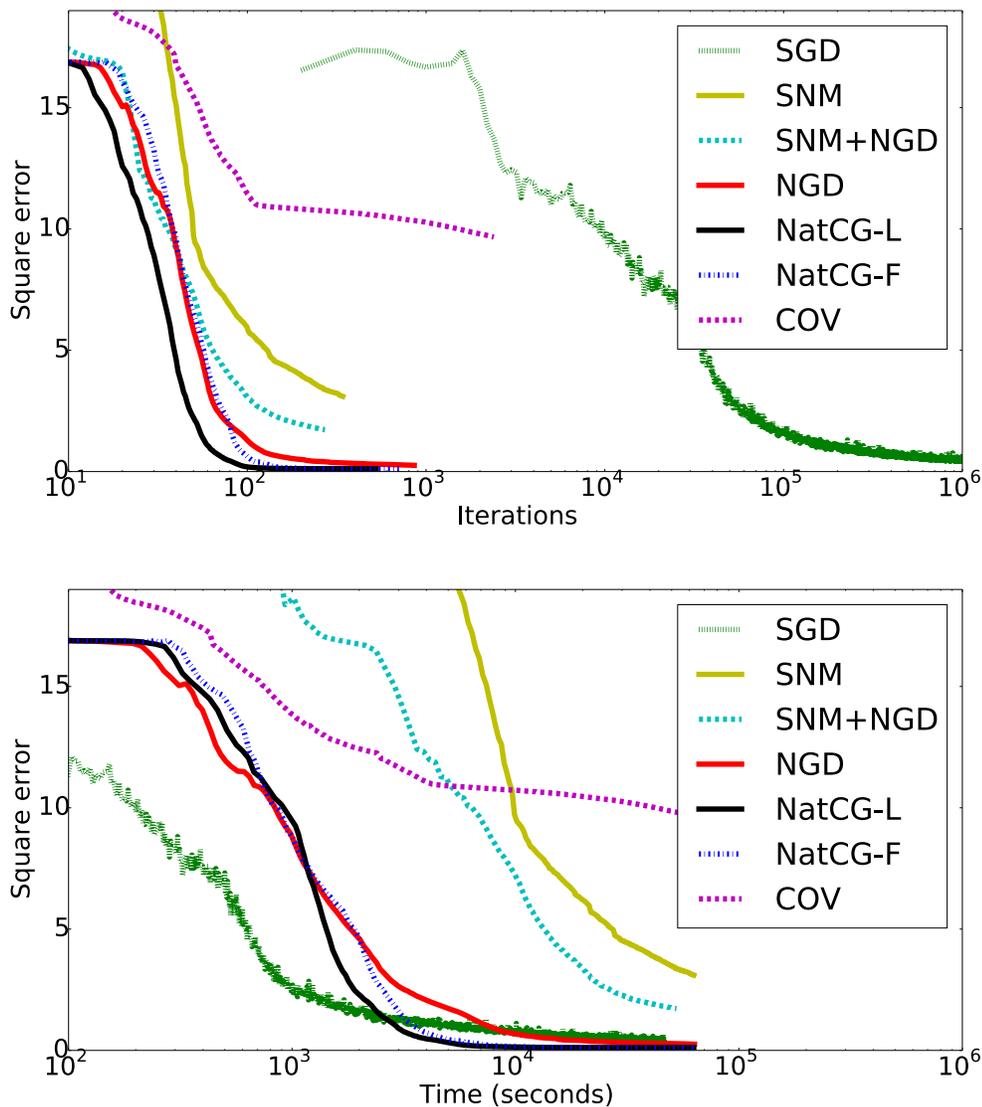


Figure 4.7: The plots depict the training curves on a log scale (x-axis) for different learning algorithms on the Curves dataset. On the y-axis we have training error on the whole dataset. Top plot compares (in terms of iterations or steps) stochastic gradient descent (SGD), Squared Newton method (SNM), a mixture of Squared Newton method (rescaled by the largest eigenvalue of the Hessian) plus natural gradient (SNM+NGD), natural gradient descent (NGD), the enhanced natural conjugate gradient, where we solve for both the step size and new conjugate direction as described in Section 4.3.6 (NatCG-L), natural conjugate gradient based on the Polak–Ribiere formula (NatCG-F) and TONGA (COV). Bottom plot shows the same algorithms in terms of clock time (seconds).

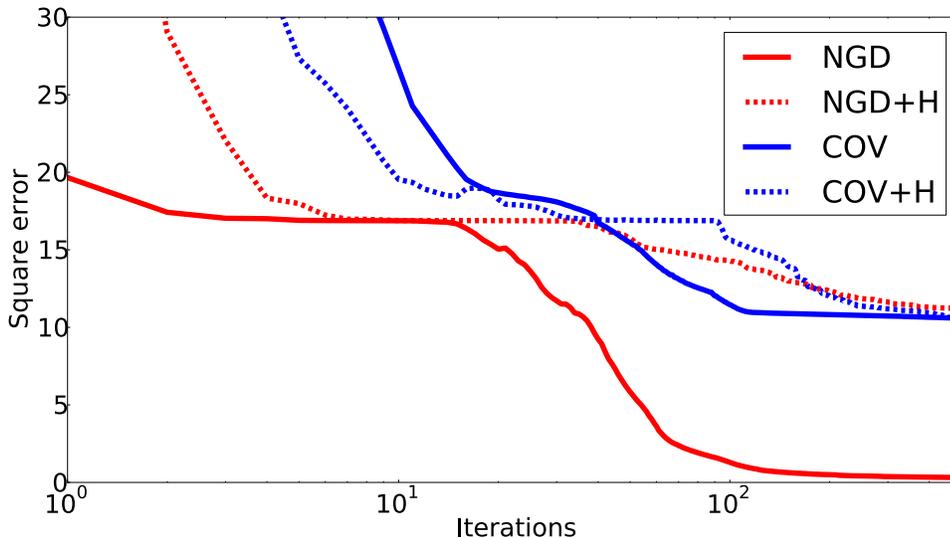


Figure 4.8: The plots depict the training curves on a log scale (x-axis) for different learning algorithms on the Curves dataset. On the y-axis we have training error on the whole dataset. It shows the effect of adding second order information (by considering a second order approximation of the loss) to natural gradient descent (NGD+H) and to TONGA (COV+H) compared to either just natural gradient descent (NGD) or TONGA (COV).

ton method with natural gradient descent. For SNM+NGD, we rescaled the matrix $\mathbf{H}^T \mathbf{H}$ by $\frac{1}{|\lambda_{[max]}|}$, factor computed at each step by running the Power method. Additionally the weight for $\mathbf{H}^T \mathbf{H}$ is of 5., while that for \mathbf{F} is just 1. These hyper-parameters had been cross-validated.

We can assume that NGD is quite reliable everywhere, except possibly near saddle points with strong negative curvature. Therefore the Fisher is used as a damping scheme, that helps reducing, in general, the skewness introduced by taking the square of the eigenvalues of the Hessian.

We believe that as long as the eigenvalues cluster together, by relying on Equation (4.75), this method roughly approximates the Saddle-Free Newton method. In terms of iterations, we can see this mixture of the two algorithm performing decently well. However, as learning progresses, the eigenvalue distribution of the Hessian becomes more skewed, making the algorithm slow down fast. We argue that this plot makes a speculative prediction that the Saddle-Free Newton method would do well when is scaled up to larger problems. By inspecting the largest eigenvalues that we use to rescale the $\mathbf{H}^T \mathbf{H}$ matrix at each step, and under the

assumption that we also have eigenvalues close to 0, we make the following observation. Initially, the difference between the largest and smallest eigenvalue of $\mathbf{H}^T\mathbf{H}$ is on the order of 10^2 . However, as learning progresses the largest eigenvalue grows very fast, getting to be a few orders of magnitude larger. This makes the distribution of the eigenvalues more spread out and results in making the algorithm fail.

In this work we do not provide a way of scaling up the Saddle-Free Newton method to large problems. We leave this as future work.

TONGA seems to also underperform compared to all other algorithms. We argue that there are two reasons for this. One reason is that the matrix \mathbf{U} , for the same amount of data, is approximated by summing fewer outer products compared to natural gradient. The metric \mathbf{U} is therefore more rank deficient. This affects the behaviour of linear CG, and makes the method more prone to noise.

The second reason is that near any critical point, by virtue of the fact that the gradients go to 0, \mathbf{U} becomes close to singular. When the metric is close to singular while the Hessian is not, the method can result in taking very suboptimal steps. This results in the algorithm getting trapped in the plateaus around these critical points. We believe such behaviour happens around the value of 11. Natural gradient descent also gets “stuck” for a very brief moment in this plateau. It might be that around this stage in training there is some saddle point structure, where \mathbf{U} becomes close to singular, while \mathbf{F} does not (or at least not by the same degree). This pushes TONGA to almost converge to said critical point, while natural gradient manages to escape it in a few steps.

We believe that the strength of TONGA is hidden in its original formulation, where a crude approximation of \mathbf{U} is used (such as a diagonal approximation). Because we know \mathbf{U} is in general not well behaved, it does not make sense to waste a lot of time on approximating it well. The key insight of this algorithm is that one can re-use the gradients computed by a first order method to find a cheap yet meaningful rescaling of each coordinate of the parameters. If this rescaling is crudely approximated, making the overhead of each step insignificant compared to stochastic gradient descent, then the algorithm becomes efficient in practice. However, if we waste lots of computations to get a very reliable approximation of these values, then the algorithm under performs.

In Figure 4.8 we see the effect of incorporating second order information by con-

sidering a second order approximation of the loss \mathcal{L} when deriving either TONGA or natural gradient descent as generalized trust region methods. An alternative view of these approach is that we use the Fisher Information matrix (for natural gradient) or the uncentered covariance of the gradients (for TONGA) to damp the Hessian.

As expected, if there is some saddle point on the pathway of these algorithm, the damping term will be used to reverse the negative direction by adding some positive value to the eigenvalue. This results in suboptimal steps. We do not use an adaptive scheme for the damping coefficient, which might result in better behaviour. For a fixed damping coefficient, we do not seem to be able to find a value which does not lead to a slow down near said critical points.

Experiments with MinResQLP

We also provide some experiments where instead of linear CG we rely on MinResQLP as the linear solver. In particular, MinResQLP is more reliable when the matrix to be inverted is close to singular or is not positive definite. For these experiments, we also explore the effect of using smaller mini-batches for natural gradient.

We provide two runs, one in which we use small mini-batches of 5000 examples, and one in which we run in batch mode. In both cases all other hyper-parameters were manually tuned. The learning rate is fixed at 1.0 for the model using a mini-batch size of 5000 and a line search was used when running in batch mode.

For comparison, we also show the behaviour of natural conjugate gradient. As before we provide the two variants, one based on the Polak-Riebiere formula (Gonzalez and Dorrnsoro, 2006) and the enhancement proposed in Section 4.3.6, implemented using `scipy.optimize.fmin_coby1a` to solve the two dimensional line search.

The first observation that we can make is that natural gradient descent can run reliably with smaller mini-batches, as long as the gradient and the metric are computed on different samples and the learning rate used is sufficiently small to account for the noise on the natural direction (or the damping coefficient is sufficiently large). This can be seen from comparing the two runs of natural gradient descent. Our result is in the spirit of the work of Kiros (2013), though the exact approach of dealing with small mini-batches is slightly different.

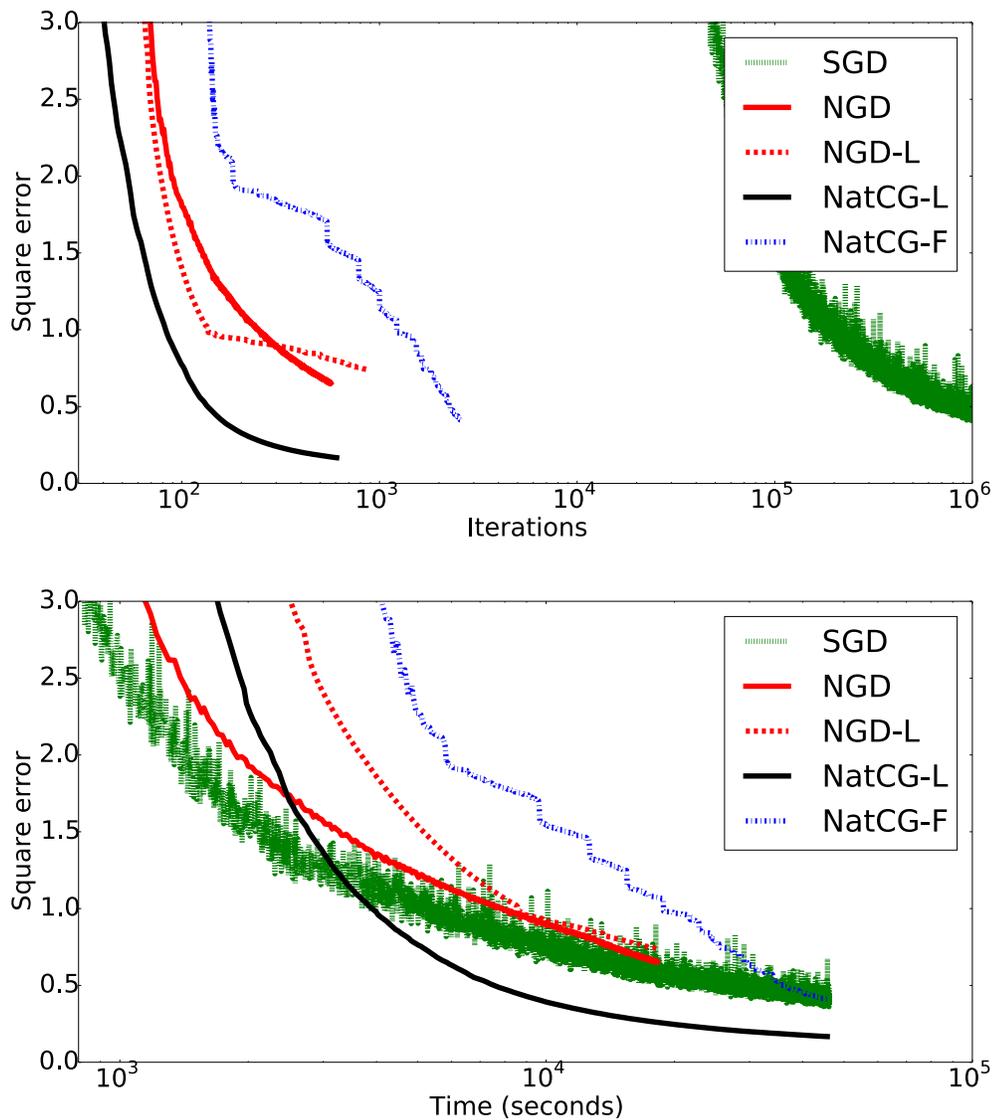


Figure 4.9: The plots depict the training curves on a log scale (x-axis) for different learning algorithms on the Curves dataset. On the y-axis we have training error on the whole dataset. The left plot shows the curves in terms of iterations, while the right plot show the curves as a function of clock time (seconds). NGD stands for natural gradient descent descent with a mini-batch of 5000 and a fixed learning rate. NGD-L is natural gradient descent in batch mode where we use a line search for the learning rate. NatCG-L uses an `scipy.optimize.fmin_coby1a` for finding both learning rate and the new conjugate direction. NatCG-F employs the Polak-Ribiere formula. SGD stands for stochastic gradient descent.

4.6 Properties of natural gradient descent

In this section we turn our attention back to natural gradient descent and discuss some specific properties of this approach. The algorithm is phrased as a learning problem. It makes the assumption that there is a model which can be understood as a distribution p_θ over the data.

At each iteration of natural gradient we minimize some loss function \mathcal{L} under the constraint of a fixed change in the KL-sense in the model p_θ . By this it means that the algorithm itself can not be applied to any function f unless we can identify in this function some meaningful model p_θ .

It is natural to ask whether there is some advantage for explicitly taking into account the model or not. We argue that there are a few such advantages.

One of them is that *natural gradient descent can be used in the online regime*. For this to hold, we assume that even though we only have a single example available at each time step, in this “online regime”, we also have access to a sufficiently large set of held out examples. If we are to apply a second order method in this regime, computing the gradient on the single example and the Hessian on the held out set would be conceptually problematic. The gradient and Hessian will be incompatible as they do not refer to the same function. However, for natural gradient descent, the metric comes from evaluating an independent expectation that is not related to the prediction error. It measures an intrinsic property of the model. It is therefore easier to motivate using a held-out set (which can even be unlabelled data as discussed in Section 4.6.1).

In [Desjardins, Pascanu, Courville, and Bengio \(2013\)](#), we show a straightforward application of natural gradient to Deep Boltzmann Machines. It is not obvious how the same can be done for a standard second order method. Probabilistic models like RBMs and DBMs do not have direct access to the cost they are minimizing. One usually can only approximate the necessary gradients in some efficient way. Fortunately, measuring the change in the KL of the model distribution does not require one to evaluate (or define) the loss we want to minimize, we just need to be able to sample from the model distribution.

Natural gradient descent is robust to local re-parametrization of the model. This comes from the constraint that we use. The KL-divergence is a measure of how the probability density function changes, regardless on how it was parametrized.

Sohl-Dickstein (2012) explores this idea, defining natural gradient descent as doing whitening in the parameter space.

The metric \mathbf{F} has two different forms, as it can be seen in equation (4.21), that of the expected Hessian of $\log p$ and that of the covariance matrix of $\nabla \log p$. Note, however, that *it is not the Hessian of the cost, nor the covariance of the gradients we follow to a local minimum*. The gradients are of $\log p_\theta$ which acts as a proxy for the cost \mathcal{L} . The KL-surface considered at any point θ during training always has its minimum at θ , and the metric we obtain is always positive semi-definite by construction which is not true for the Hessian of \mathcal{L} .

If we look at the KL constraint that we enforce at each time step, it does not only ensures that we induce at least some ϵ change in the model, but also that the model does not change by more than ϵ . We argue that this provides some kind of robustness to over-fitting. The model is not allowed to move *too far* in some direction \mathbf{d} if moving along \mathbf{d} changes the density computed by the model substantially.

4.6.1 Using unlabelled data

When computing the metric of natural gradient descent, the expectation over the target \mathbf{t} is computed where \mathbf{t} is taken from the model distribution for some given \mathbf{u} : $\mathbf{t} \sim p(\mathbf{t}|\mathbf{u})$. For the standard neural network models this expectation can be evaluated analytically (given the form of $p(\mathbf{t}|\mathbf{u})$). This means that we do not need target values to compute the metric of natural gradient descent.

Furthermore, to compute the natural gradient descent direction we need to evaluate two different expectations over \mathbf{u} . The first one is when we evaluate the expected (Euclidean) gradient, while the second is when we evaluate the metric. In this section we explore the effect of re-using the same samples in computing these two expectations as well as the effect of improving the accuracy of the metric \mathbf{F} by employing unlabelled data.

Statistically, if both expectations over \mathbf{u} are computed from the same samples, the two estimations are correlated to each other. If we use different examples, then the two estimations are less correlated. We hypothesize that using the same examples can lead to an increase in the over-fitting of the current mini-batch. Figure 4.10 provides empirical evidence that our hypothesis is correct. Vinyals and

Povey (2012) make a similar empirical observation.

As discussed in Section 4.6, enforcing a constant change in the model distribution helps ensure stable progress but also protects from large changes in the model which can be detrimental (could result in over-fitting a subset of the data). We get this effect as long as the metric provides a good measure of how much the model distribution changes. Unfortunately the metric is computed over training examples, and hence it will focus on how much p changes at these points. When learning over-fits the training set we usually observe reduction in the training error that result in larger increases of the generalization error. This behaviour can be avoided to some extent by natural gradient descent if we can measure how much p changes far away from the training set. This will allow us to slow down in the region of over-fitting, and either by switching to different examples, or by early-stopping, we can avoid over-fitting those specific examples. To explore this idea we propose to increase the accuracy of the metric \mathbf{F} by using unlabelled data, helping us to measure how p changes far away from the training set.

We explore empirically these two hypotheses on the Toronto Face Dataset (TFD) (Susskind et al., 2010) which has a small training set and a large pool of unlabelled data. Note that in the large data regime, the mini-batch over which we compute the metric might be large enough to contain all the necessary statistics. That is, the claims made here are especially important for smaller datasets. Figure 4.10 shows the training and test error of a model trained on fold 4 of TFD, though similar results are obtained for the other folds.

We used a two layer model, where the first layer is convolutional. It uses 512 filters of 14x14, and applies a sigmoid activation function. The next layer is a dense sigmoidal one with 1024 hidden units. The output layer uses sigmoids as well instead of softmax. The data is pre-processed by using local contrast normalization.

Hyper-parameters such as learning rate, starting damping coefficient have been selected using a grid search, based on the validation cost obtained for each configuration.

We used a fixed learning rate of 0.2 (with no line search) and adapting the damping coefficient using the Levenberg-Marquardt heuristic.

When using the same samples for evaluating the metric and gradient we used mini-batches of 256 samples, otherwise we used 384 other samples randomly picked from either the training set or the unlabelled set. We use MinResQLP as a linear

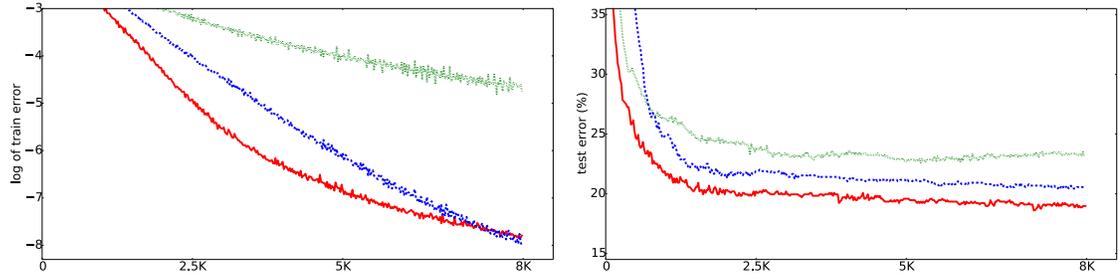


Figure 4.10: The plot depicts the training error (left) on a log scale and test error (right) as percentage of misclassified examples for fold 4 of TFD. On the x-axis we have the number of updates. Dotted green line (top, worst) stands for using the same mini-batch (of 256 examples) to compute the gradient and evaluate the metric. Dashed blue line (middle) uses a different mini-batch of 384 examples from the training set to evaluate the metric. The solid red line (bottom, best) relies on a randomly sampled batch of unlabelled data to estimate the metric.

solver, the picked initial damping factor is 5., and we allow a maximum of 50 iterations to the linear solver.

We notice that *re-using the same samples for the metric and gradient results in worse global training error* (training error over the entire train set) *and worse test error*.

Our intuition is that we are seeing the model over-fitting, at each step, the current training mini-batch. At each step we compute the gradient and the metric on the same examples. There can be, within this mini-batch, some direction in which we could over-fit some specific property of these examples. Because we only look at how the model changes at examples in the mini-batch, when computing the natural gradient step, the metric will not oppose this over-fitting behaviour. However the step is not useful for generalization, nor is it useful for the other training examples (e.g. if the particular deformation is not actually correlated with the class it needs to predict). This means that on subsequent training examples the model will underperform, resulting in a worse overall error.

On the other hand, if we use a different mini-batch for the gradient and for the metric, it is less likely that the same particular feature to be present in both the set of examples used for the gradient and those used for the metric. So either the gradient will not point in said direction (as the feature is not present in the gradient), or, if the feature is present in the gradient, it may not be present in the examples used for computing the metric. That would lead to a larger variance, and hence the model is less likely to take a large step in this direction.

Using unlabelled data results in better test error for worse training error, which means that this way of using the unlabelled data acts like a regularizer.

4.6.2 Robustness to reorderings of the train set

We want to test the hypothesis that, by forcing a constant change in the model, in the KL sense, the model avoids to take large steps towards specific patterns, being more robust to the reordering of the train set.

We repeat the experiment from [Erhan et al. \(2010\)](#), using the NISTP dataset introduced in [Bengio et al. \(2011\)](#) (which is just the NIST dataset plus deformations) and use 32.7M samples of this data. The original experiment attempts to measure the importance of the early examples in the learnt model. To achieve this we respect the same protocol as the original paper described below:

Algorithm 4 Protocol for the robustness to the order of training examples

- 1: Split the training data into two large chunks of 16.4M data points
 - 2: Split again the first chunk into 10 equal size segments
 - 3: **for** i between 1 and 10 **do**
 - 4: **for** steps between 1 and 5 **do**
 - 5: Replace segment i by new randomly sampled examples that have not been used before
 - 6: Train the model from scratch
 - 7: Evaluate the model on 10K heldout examples
 - 8: **end for**
 - 9: Compute the segment i mean variance, among the 5 runs in the output of the trained model (the variance in the activations of the output layer)
 - 10: **end for**
 - 11: Plot the mean variance as a function of which segment was resampled
-

Figure 4.11 shows these curves for mini-batch stochastic gradient descent and natural gradient descent.

Note that the variance at each point on the curve depends on the speed with which we move in function space. For a fixed number of examples one can artificially tweak the curves by decreasing the learning rate. With a smaller learning rate we move slower, and hence the model, from a function point of view, does not change by much, resulting in low variance. In the limit, with a learning rate of 0, the model always stays the same. In order to be fair to the two algorithms compared in the plot, natural gradient descent and stochastic gradient descent, we use the error

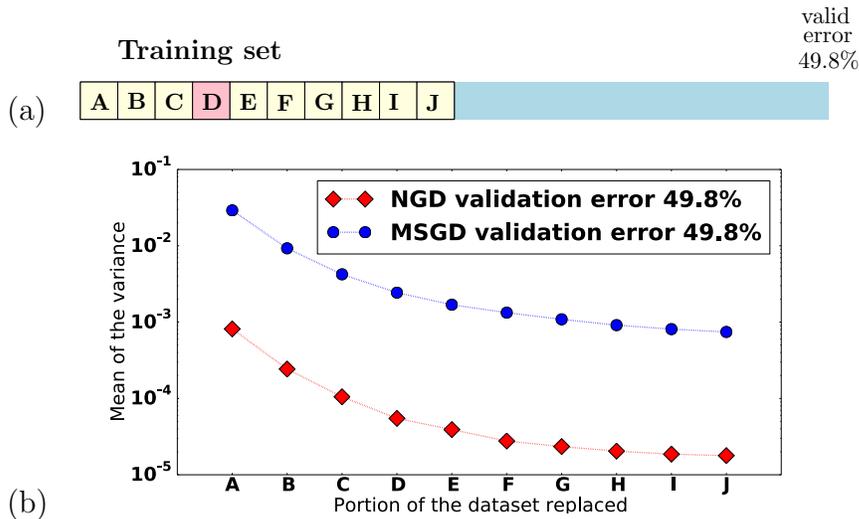


Figure 4.11: The plot describes how much the model is influenced by different parts of an online training set. (a) exemplifies how the training dataset was split into two, and then the first half into ten segments A, B, C, D, E, F, G, H, I, J. (b) shows the variance induced by re-shuffling of data examples in one of these segments, when everything else was kept the same. Note that natural gradient descent shows order of magnitude lower variance than SGD. See text for more information.

on a different validation set as a measure of how much we moved in the function space. This helps us to choose hyperparameters such that after 32.7M samples both methods achieve about the same validation error of 49.8%. Both models are run on mini-batches of the same size, 512 samples. We use the same mini-batch to compute the metric as we do to compute the gradient for natural gradient descent, to avoid favoring NGD over MSGD.

The model we experimented with was an MLP of only 500 hidden units. We use a constant damping factor of 3 to account for the noise in the metric (and ill-conditioning since we only use batches of 512 samples). The learning rates were kept constant, and we use 0.2 for the natural gradient descent and 0.1 for MSGD.

The results are consistent with our hypothesis that natural gradient descent avoids making large steps in function space during training, staying on the path that induces least variance. Note that relative variance might not be affected when switching to natural gradient descent. That is to be expected, as, in the initial stages of learning, any gradient descent algorithm needs to choose a basin of attraction (a local minimum) while subsequent steps make progress towards

this minimum. That is, the first examples must have, relatively speaking, a big impact on the learnt model. However what we are trying to investigate is the overall variance. Natural gradient descent (in global terms) has lower variance, resulting in more consistent functional behaviour regardless of the order of the input examples.

MSGD might move in a direction early on that could possibly yield over-fitting (e.g. getting forced into some quadrant of parameter space based only on a few examples) resulting in different models at the end. This suggests that natural gradient descent can deal better with nonstationary data and can be less sensitive to the particular examples selected early on during training.

4.7 Summary and Outlook

In this chapter we looked at different optimization techniques for learning. We proposed the framework of *generalized trust region methods*, a straightforward extension of traditional trust region approaches, which can be summarized by two modifications. The first one is that we allow minimization of a first order Taylor expansion of the function \mathcal{L} . In contrast, trust region methods are typically defined based on a second order approximation of \mathcal{L} . The second change is that we use a constraint on the distance between θ and $\theta + \Delta\theta$, as given by some distance measure d , instead of just looking at the norm of $\Delta\theta$.

We have discussed that many of the recently proposed algorithms for training deep models can be cast in this framework. Specifically we looked at natural gradient descent, Hessian-Free Optimization, TONGA, natural conjugate gradient and Krylov Subspace Descent. In our analysis we found a few relationships between these algorithms. Due to the use of the extended Gauss-Newton approximation of the Hessian, Hessian-Free Optimization and Krylov Subspace Descent can both be understood as natural gradient descent approaches. In contrast, TONGA is not the same as the original natural gradient algorithm proposed by Amari. It is, though, still a generalized trust region method. TONGA can be understood as the algorithm that minimizes a first order approximation of the loss restricted to the region in which the expected change in the loss is bounded by some constant.

Natural conjugate gradient is an extension of the natural gradient descent. It incorporates second order information by moving in directions conjugate to the

Hessian of the loss with respect to the functional manifold. In our work, we highlighted one particular weakness of a previously proposed variant of the algorithm and specifically to the set of approximations it suggested. Based on our interpretation of Krylov Subspace Descent as a variant of natural conjugate gradient, we proposed an alternative way of computing a new conjugate direction which behaves better in practice.

These algorithms can all be efficiently implemented using a truncated Newton approach where the products between the Hessian or the Fisher Information matrix or the covariance of gradients and some vector can be efficiently computed using the R and L operators (described in Section 4.1.1).

In Section 4.4 we looked at the saddle point problem. Based on a change of coordinates (similar to Morse’s lemma) we rephrased the function \mathcal{L} around a nondegenerate critical point as a weighted sum of squares, where the weights are given by the eigenvalues of the Hessian. This formulation allows for an intuitive description of the behaviour of different algorithms in the neighbourhood of the critical point. The analysis reveals that most algorithms take a suboptimal step. The derivation also indicates what an ideal step would be around this critical point. We provide a proper justification for an algorithm that realizes exactly this step, an algorithm that we called Saddle-Free Newton method.

This analysis is important as there is evidence that there exists many saddle points for large dimensional non-convex problems. Around any of these critical points the first order gradients will vanish, making SGD slow down considerably along some directions. This suggests that using the curvature (be it of the underlying manifold or of the function) is important not only close to the local minima, but in the early stages of learning as well (where we are close to saddle points).

In Section 4.6 we introduced two properties of the natural gradient method. The first property is that the algorithm can use unlabeled data to enhance the precision of the considered metric, which results in a regularization effect. This is particularly useful if we want to use smaller mini-batches for the algorithm. We validate that natural gradient (where the metric is damped accordingly) can run on medium sized mini-batches of 256 examples. It is however necessary to use different samples to approximate the metric versus the samples used to approximate the gradient. Otherwise the algorithm over-fits the current mini-batch, resulting in both worse training and test error.

For a second order method, it might be harder to justify computing the Hessian on different examples than the gradient in order to avoid over-fitting the current mini-batch. If did so, the Hessian would not be the Hessian of the function we are computing the gradients of.

If we use held out data to compute the metric, we get a sense of variability in the model far away from the training set. This can make the algorithm more cautious resulting in a regularization effect.

The second property that we empirically evaluated is the robustness of natural gradient to the order of the training set (compared to stochastic gradient descent). This robustness can help (to some degree) when working with non-stationary datasets.

We regard our work as preliminary, and there are many gaps left to fill. A more thorough comparison of the different possible algorithms is still lacking. In particular we believe that any such benchmark should also be accompanied by some form of unit-testing that checks the performance of the algorithms on particular patterns in the error surface. From our results it seems that natural gradient, an algorithm also validated in the literature, behaves quite well in general. The extension of natural gradient descent, natural conjugate gradient seems to do pretty well as well.

We regard the study of the Saddle-Free Newton method as a promising future research direction. In this work we only provided a theoretical description of the algorithm and small scale experiments. Scaling it up is still an open question that we intend to study. Other possible research directions include studying the properties of saddle points for large neural networks, and exploring the relationship between these saddle points of the training error and expected generalization error. For example, is the index of a critical point correlated with how likely we are to over-fit the training set? Also our analysis focuses on nondegenerate critical points. Because of the recurrent weight matrix, one can argue that the gradient of a recurrent network contains the value of some parameter raised to odd powers, which could induce degenerate critical points. Can our analysis be extended to such critical points?

Interesting questions are left unanswered for natural gradient as well. Empirical evaluation of this algorithm in the “online” regime or on non-stationary data, for example, is lacking. A thorough benchmarking of the algorithm on many task, or

even scaling the algorithm to very large tasks are other possible future research directions.

We conclude this chapter with two remarks. Optimization for learning right now has a “hammer” widely used by everyone in the community. This “hammer” is stochastic gradient descent. It is still not clear why this algorithm manages to be so resilient in all benchmarks, even when compared with theoretically much more powerful algorithms. We believe that these more powerful algorithms have “failure modes” that make them underperform, and properly understanding these weaknesses can help figure out when the “hammer” might not be sufficient anymore. Also the field has a bias towards models that work well with this “hammer”. Is there something to be lost from this bias ?

One other interesting research direction that we want to consider is parallelization. SGD is inherently a sequential algorithm. It relies on taking many small and computationally cheap steps. When one tries to parallelize such an algorithm it becomes somewhat difficult. Each step is already relatively cheap, and after each such step is taken one has to, in principle, communicate between all nodes the new value of the parameters. This communication between nodes usually becomes the bottleneck of any such parallelization scheme. In situations like these people rely on asynchronous approaches, see, for example, [Paine et al. \(2014\)](#). These approaches end up not making proper use of all the nodes involved.

When one increases the mini-batch size for SGD, in order to gain more computations to be parallelized at each step, the gain is minimal. The reason is that SGD is a fairly inefficient way of exploiting a large mini-batch of data. Instead a second order method or natural gradient approach is much more efficient, as it does not simply take the mean of the different gradients.

For second order methods, each step is expensive, which means that one has to communicate and update less often the parameters of the model. How to parallelize each iteration of a second order method is still an open question. For example CG does not necessarily seem to be easy to parallelize, though a different approach can be taken to compute the descent direction. For example, if we are to restrict our problem to a Krylov subspace, computing this subspace can easily be done in parallel, where different nodes look at different rows of the matrix that need to be inverted (parallelization along the model).

We believe that parallelization of higher order methods is an interesting and

attractive future direction which unfortunately is somewhat unexplored at this time.

5

Training recurrent neural models and memory

In this chapter we attempt to focus on recurrent neural networks and ask the question: Can we learn recurrent models that exhibit stable memory that can store some information for a possibly unbounded period of time? This is a restriction of the generic optimization problem discussed in the previous chapter to a particular challenging issue that arises in optimizing recurrent networks.

In this chapter we will rely on some of the concepts introduced in Section 2.3, which offers a refresher in recurrent models. In Section 5.1 we provide some motivation for why it is important to study and understand memory for recurrent models. Section 5.2 introduces some basic concepts of dynamical system theory, that we will use later. Section 5.3 provides some intuitions on how memory might be achieved in a recurrent model and what we mean by memory in this context. Section 5.4 provides a specific recurrent structure that results in models that can exhibit memory. To achieve this we *cheat* by providing hints during learning about how the memory should behave. In Section 5.5 we describe in detail two problems of training RNNs that affect its ability to exhibit long memory traces, the exploding and vanishing gradients problem. Sections 5.7 and 5.8 provide two heuristic solutions for these problems. We show, in Section 5.9 that by employing these heuristic solutions we are able to train an RNN that exhibit long memory traces with out using hints. Finally we provide some conclusions in Section 5.10.

The content of this chapter overlaps with the two papers I published together with my co-authors, Pascanu and Jaeger (2011); Pascanu, Mikolov, and Bengio (2013). I *borrowed* figures and entire paragraphs from these works when writing the chapter. The first paper describes the concept of working memory and provides a model that can exhibit memory when trained with hints. The second paper focuses on the exploding and vanishing gradients problem for training recurrent neural networks. Please see Section 1.1 for a detailed description of my personal contribution to these papers.

5.1 Motivation

As argued in Section 2.3, recurrent neural models have the same power as Turing machines, and can, therefore, simulate any algorithm. This makes them more powerful than feedforward networks, and, from a pure theoretical perspective, it is to be expected that in order to achieve artificial intelligence we would need recurrent connections¹.

In particular, the power of recurrent neural networks comes from the fact that they can exhibit memory. Memory is the crucial component that is needed to be able to simulate a Turing machine. We use these memory traces to remember where we are inside a program and to know what is the next step that needs to be taken.

Leaving aside the ultimate goal, artificial intelligence, even for tasks at hand memory seems to be necessary to obtain good results. For example, let us assume we are to work on a language modelling task, where we are asked to predict which is the next word in a sequence of words. By remembering the last seen words we have a better chance of predicting what comes next. We can rely on short term information to make a good guess, namely on the last few words in the sequence. In this case we would basically want to add the missing word from the current phrase. But long term information can also help. It can define the topic of this stream of words. This topic can influence the distribution of words. Or it might define the stylistic characteristics of the writer (or current speaker), which might put more probability on some words or phrases over others.

For speech or video the same principle holds. For example, in a noisy audio signal we can use the previously uttered words to disambiguate the current word or even phoneme, which due to the noise might not be recognizable. It is hard to tell how far in the past we need to go to do a good job at this disambiguation. For example the word might be a new name, which we can disambiguate by remembering the initial phase of the conversation, when the name was first introduced. In the case of video, understanding the current action of an actor highly depends on the motion it did before.

We as humans rely heavily on context (memory and history) to disambiguate the world around us and to define our intentions or the intentions of others. That

1. Arguably, artificial intelligence could also be based on some hybrid model, that only uses neural networks to map complicated patterns to simpler ones and relies on something else, like symbolic logic, for the control and decision making modules.

is to say there is a lot of information in the way events are ordered in time, and if we want to get better at solving some tasks we *have* to exploit this information.

One option is to imitate the computer architecture, and have the processing neural network connected to stacks or memory buffers in the same way the CPU is connected to RAM. These buffers or stacks do not themselves need to be neural, but actual memory buffers made readily available in some programming language.

We argue, however, that there is something to be gained from organically integrating memory into the neural network, as it is done, for example, in a recurrent network. By doing so we allow the memory to influence continuously the ongoing processing of the model at every level, in the sense of providing context to the processed information. Getting these interactions in the classical framework can be cumbersome. In the recurrent network the stored information *directly and implicitly interacts* with the computations carried out by the model.

The advantage of complex RNN based signal processing can be *richness and dynamical adaptivity* of internal representations. The interaction between memory and the ongoing processing of the model can result in high-dimensional, self-organizing (in interesting and complex ways), dynamically evolving memory traces that can be continuously adapted through learning.

Another way of phrasing these properties is the following. A shielded memory buffer will do exactly what it is programmed to do. Store the information inside the buffer until there is a signal that says this information needs to be read by the model or replaced or deleted. Each bit of information is stored separately in a stable manner and can not get lost or destroyed by accident (unless the memory gets corrupted which is not to be expected with modern memory buffers). This shielding suggests *a priori* knowledge of what and why we want to memorize. Given that the memory is finite, the system needs to immediately identify the bits that need to be stored, and when some new important information comes along, it needs to know what stored bits can be sacrificed and completely removed.

When going “neural” the hope is that such hard decision do not need to be made. There is no barrier between the stored information and the neural network. This implies, on one hand side, that it is easy for the information to be corrupted or lost. In fact, the first step that needs to be taken for RNN research is to show that they actually can store some information with out *accidentally* corrupting it.

On the other hand some subset of these *corruptions* might be useful and can

result in a form of clustering of the stored information. The model can end up *vaguely* remembering events, or similar events can be confused or replaced by some nominal category. Such kind of confusions can lead to interesting generalizations, helping the model do well on inputs it never seen before, for which it is not clear what needs to be remembered.

Obtaining the same in the classical framework involves a lot of hand engineering and understanding how these generalization events could happen.

These intuitions can be used to motivate this research direction, though systematic proof that such adaptive neural memory provides an advantage over the alternatives is still missing. In this chapter we are attempting to take some small steps in this direction, and hope that future research will be able to elucidate this question further.

5.2 Brief introduction to Dynamical Systems for Recurrent Networks

For a detailed and complete introduction of Dynamical Systems theory we invite the reader to look at [Strogatz \(1994\)](#) which provides an intuitive treatment of the topic. In here we will only provide an informal treatment of the subject, limited to the concepts that we will use later on.

Dynamical Systems theory tries to *analyse the evolution* of some predefined system *in time* under some fixed evolution rule that describes how the model changes from one time instant to the next. Historically, these kind of approaches date back to the invention of calculus and classical mechanics by Sir Isaac Newton and they have seen drastic improvements by the work of researchers as Poincaré, Lorenz and Mandelbrot to name just a few. Today it forms an important field on its own with applications in economics, analysis of fluid motion, quantum field theory, general relativity and as we will briefly discuss here for recurrent neural networks.

Formally a dynamical system is defined by a *phase space* or *state space* \mathcal{S} , whose coordinates describe the state of the system, a set of times \mathcal{T} , and a dynamical rule $\mathcal{R} : \mathcal{S} \times \mathcal{T} \rightarrow \mathcal{S}$ that gives the consequent state $s \in \mathcal{S}$.

The dynamical system can be continuous, if the set \mathcal{T} is continuous, or discrete. Depending on the nature of \mathcal{R} , the dynamical system can be linear or nonlinear, deterministic or stochastic. If additionally to the previous state $s \in \mathcal{S}$ and the current time step $t \in \mathcal{T}$ the dynamical rule takes some additional input \mathbf{u} , then it is not an autonomous dynamical system, otherwise it is.

In our work we will focus on deterministic, discrete and nonlinear dynamical systems, where the rule is given by a map $\rho : \mathcal{S} \rightarrow \mathcal{S}$. Most of the concepts we will use refer to autonomous dynamical systems. See [Pascanu and Jaeger \(2011\)](#) for a description of such concepts for an input-driven model. Some of these concepts for input-driven models are also analyzed in [Manjunath and Jaeger \(2013\)](#).

5.2.1 Phase portraits and bifurcation diagrams

The evolution of a dynamical system in time forms a *trajectory* or *orbit* that describes the sequence of states the system goes through when started from some initial state $s_{[0]}$. The set of all possible trajectories that a system can go through represents a phase portrait. A visualization of these phase portraits usually depicts only a few important trajectories.

One approach to analysing dynamical systems is to analyse their behaviour once the bias induced by the initial state disappears. In such an analysis we usually look at *attractors*. Loosely speaking, an attractor is a set of points to which all neighbouring trajectories converge. More strictly, an attractor \mathcal{A} satisfies the following properties:

- any trajectory that starts in \mathcal{A} stays in \mathcal{A} (\mathcal{A} is an invariant set)
- There exists an open set \mathcal{U} such that any trajectory that starts in \mathcal{U} will converge to \mathcal{A} . Formally the distance between $s_{[t]}$, where $s_{[t]}$ describes the system at time t , and \mathcal{A} goes to 0 as t goes to infinity. The set \mathcal{U} is called the *basin of attraction* of \mathcal{A} .
- \mathcal{A} is minimal, that is there is no subset of \mathcal{A} that satisfies the properties mentioned above

Attractors can take different shapes, from a single point (fixed point attractors), to limit cycles or chaotic attractors. Additionally to attractors, which are sometimes called *sinks*, one can also observe, in the phase portrait of a dynamical system, *repellers* or *sources*. A repeller is the set of points \mathcal{A} such that any trajec-

tory starting in \mathcal{A} ends in \mathcal{A} , but any trajectory starting in the neighbourhood of \mathcal{A} diverges, moving away from \mathcal{A} . For example, local maxima are sinks or attractors of the associated gradient system of some function f , that is the continuous system whose update rule is a differentiable equation following the gradient of f , $\dot{s} = \frac{\partial f(s)}{\partial s}$. Local minima are repellers of this system as they are unstable fixed points. The list of invariant sets also includes saddle structures, where one has repelling directions and attracting ones (see, for example, the detailed discussion in Section 4.4). Figure 5.1 depicts some text book examples of attractors and repellers.

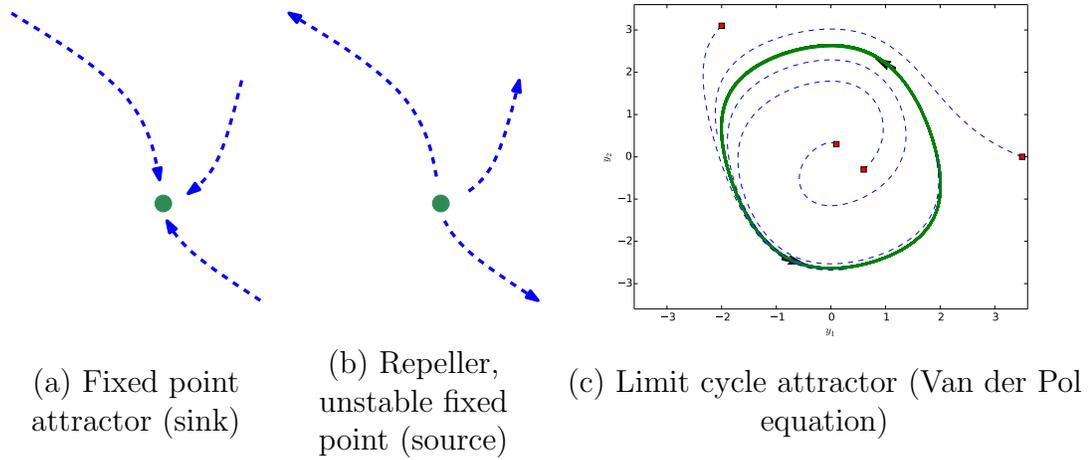
The set of all attractors, repellers or saddle structures (described by their nature and relative position) that some dynamical system exhibits provides a *qualitative description of the global behaviour* of the system. One approach to reasoning about these dynamical system is by analysing or describing these invariant sets of points.

Additionally, information such as the basin of attraction associated with certain attractors as well as the position of the boundaries between such basins of attraction can provide information about the expected trajectory that a system can take given a certain position in the state space.

Autonomous dynamical systems can find themselves also in a chaotic regime. Some prerequisites for a system to be completely chaotic are:

- the system has to be sensitive to initial conditions; this property is sometimes referred to as the *butterfly effect*, when small changes in the initial conditions results in completely different states as t goes to infinity.
- the system has to show *topological mixing*; a system exhibits topological mixing if there is a point $s_{[0]} \in \mathcal{S}$ such that the trajectory starting in s_0 is dense. Additionally, the same property is described by saying that for any two sets $\mathcal{U}, \mathcal{V} \subset \mathcal{S}$ there exists a $t \in \mathcal{T}$ such that $\mathcal{R}(\mathcal{U}, t) \cap \mathcal{V} \neq \emptyset$. The two definitions are not equivalent. Informally, topological mixing ensures that starting at most points $s_{[0]}$ we explore the whole phase space.

Going one step further, one typical assumption that can be made for dynamical system is that the evolution rule \mathcal{R} is parametrized by some vector of values $\theta \in \Theta$. This is specifically relevant for recurrent networks where we will identify the state space with the possible set of values of the hidden state, and the evolution rule (or discrete map) with the update rules of the recurrent network. As such, the dynamical system that a recurrent network could represent is always parametrized by the weights of the model.



Lorenz Attractor

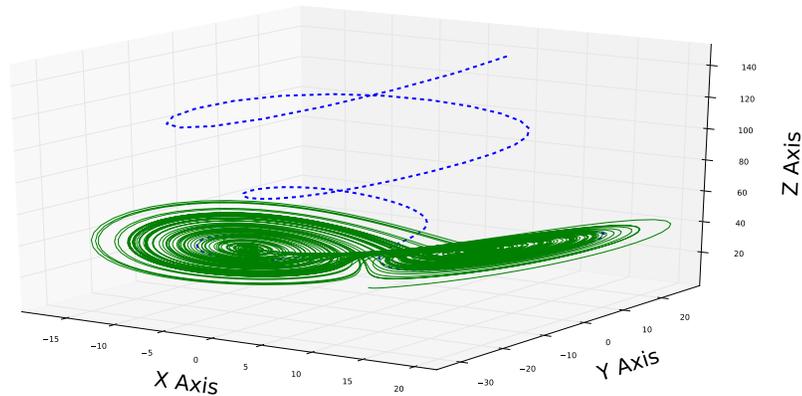


Figure 5.1: Illustration of a fixed point attractor in (a), a repeller in (b), a limit cycle attractor (given by the Van der Pol equation) in (c) and a strange or chaotic attractor in (d), named the Lorenz attractor. In green we show the attracting set, while with dashed blue lines we show trajectories that point towards the attracting set in (a), (c) and (d) or away from the repelling set in (c).

For such systems one can rely on a bifurcation diagram to qualitatively understand how changing the parameter vector θ affects the behaviour of the model. Let us consider a classical example, the logistic map, to describe a bifurcation diagram.

The logistic map is a dynamical system whose state space and parameter space are both given by the set of real numbers \mathbb{R} . The evolution rule is given by:

$$s_{[t]} = \rho(s_{[t-1]}) = \theta s_{[t-1]}(1 - s_{[t-1]}) \quad (5.1)$$

This system was popularized by biologist Robert May (May, 1976) as a discrete system to describe demographic growth. Figure 5.2 shows the bifurcation map corresponding to this system. Of particular interest are the values of the parameter θ for which the types (or number) of attractors (or other invariant set) of the system changes as we move on one side of this value.

Specifically, we can see in the plot that before the value $\theta \approx 2.9$ the system exhibits a single point attractor. As we change the value of θ the position of this single point attractor changes smoothly, however, topologically, the phase portrait of the system remains the same. After some critical value of θ close to 2.9 we suddenly have a 2-state periodic attractor instead of our original single point attractor. This is a topological different phase portrait. The critical value of θ that delimits these two different phase portraits is called a bifurcation boundary.

Bifurcation boundaries describe qualitative changes in the behaviour of the system. A particular such boundary is usually referred to as the *edge of chaos*. The edge of chaos refers to that bifurcation boundary that leads into a chaotic regime of the model.

5.3 Working memory

The concept of *working memory* (WM) refers to, following the definition from Durstewitz et al. (2000), “the ability to transiently hold and manipulate goal-related information to guide forthcoming actions.” To these properties we believe an important addition is stability. Or at least the information can be stabilizable on demand, e.g. by rehearsal.

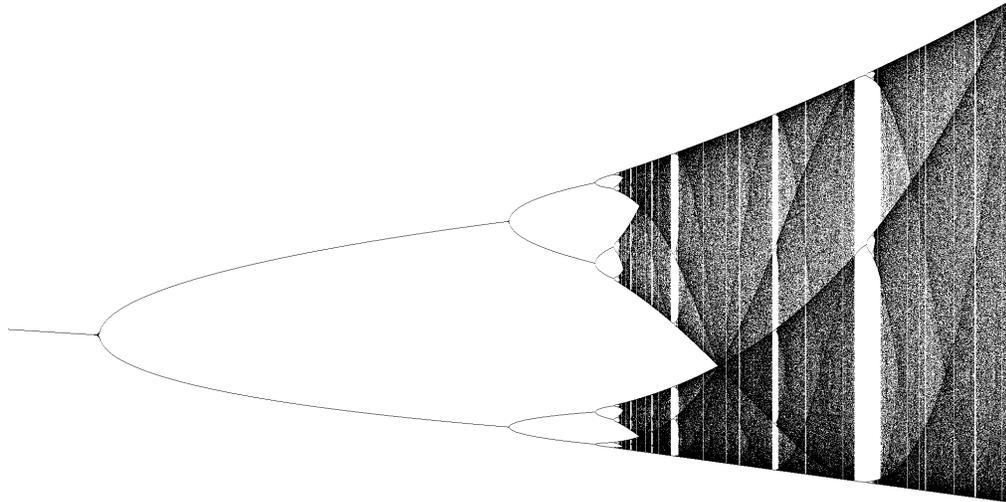


Figure 5.2: Bifurcation map for the logistic map. Each vertical line of the figure corresponds to the phase portrait for a certain value of the parameter θ . We only plot the different attractors set of the phase portrait. Note how for $\theta < 2.9$ the system has a single point attractor. For $\theta > 2.9$ the single point attractor converges into a 2-state periodic attractor, which will end up into a chaotic regime when $\theta > 3.569$.

Let us rephrase this definition in specific terms related to recurrent models. Working memory describes the ability of the model to remember information not by storing it in its weights, but rather in its hidden state. That is the model should be able to memorize sequences that it has not seen during training if there are some cue signals stating that such information will be useful further on. The memory trace should only exist during the evaluation of the model on a particular input sequence and not be part of the model.

The information stored has to be useful for the task to be solved and should continuously influence or define the behaviour of the model on incoming inputs. In principle, this information should be stored as long as needed, potentially for the entire length of the sequence.

A large portion of the RNN oriented literature on working memory is concerned with models that explain behavioural and neural observations from cognitive processing experiments (reviews [Durstewitz et al. \(2000\)](#); [Howard \(2009\)](#)).

A common belief is that stable short-term memory for RNNs is realized through *attractors*. Following this intuition a large number of attractors had been explored in various contexts.

The biologically oriented literature relies on point attractors (cell assemblies and bistable neurons) and traveling waves (synfire chains) as discussed in [Durstewitz et al. \(2000\)](#). In theoretical physics, spatiotemporal attractors (or pattern formations in excitable media) are explored, with connections into computational neuroscience and robotics via neural field theories of cortical representation ([Schöner et al., 1995](#); [Freeman, 2007a,b](#)). Coupled oscillators, that can be connected to attractors in spiking neural networks, are also thoroughly studied, see, for example, [Radicchi and Meyer-Ortmanns \(2006\)](#).

Chaotic attractors have been investigated as information representing neural mechanism in [Yao and Freeman \(1990\)](#) and [Babloyantz and Lourenço \(1994\)](#). Such models of memory offer a rich structure (due to the complexity of the chaotic attractor) and the possibility to stabilize or address sub-lobes as representational units ([Stollenwerk and Pasemann, 1996](#); [Tsuda, 2001](#)).

One fundamental flaw of the attractor view of working memory is that, by definition, attractors keep the system trajectory confined in their support (that is the model can not leave the set of points defining the attracting set). Cognitive dynamics do not seem to have this property. We can forget information, for example, which would be equivalent with leaving this support set of the attractor representing the stored information.

An important challenge for this view, is therefore, explaining how these attractors can be left. Many possible answers had been proposed. One provided solution is that of neural noise which can kick the trajectory out of some attractor. This solution is, however, unsatisfactory. Noise can not be specific, and forgetting information does not seem a random process. We need a more controlled (and input driven) mechanism to leave these attractors.

If we allow ourselves to move away from the standard definition of an attractor, attractor-like phenomena have been considered to realise memory. Such phenomena usually emerges in high-dimensional nonlinear dynamics: saddle point dynamics ([Rabinovich et al., 2008](#); [Sussillo and Barak, 2013](#)); *attractor relics* (or *attractor ruins*) where classical attractors in a fast-timescale subsystem are destroyed by a slow-timescale saturation dynamics ([Gros, 2009](#)); *transient attractors* defined by transient volume contractions of a flow ([Jaeger, 1995](#)); *unstable attractors*, which are classical attractors that appear in certain spiking neural networks and can be left under the impact of arbitrarily small noise because they are surrounded arbi-

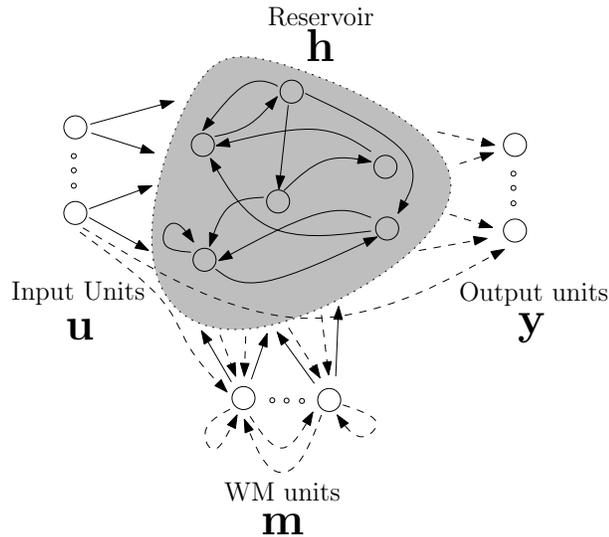


Figure 5.3: Diagram of the WM model. Dashed connections are trained, the others are left untouched. Note that the main difference to a standard echo state network is the presence of memory units. These units differentiate themselves from output units by having trainable connections among themselves. Also in our setup the output units do not have feedback connections.

trarily closely by basins of other attractors (Timme et al., 2002); *high-dimensional attractors* (initially named *partial attractors*) which govern only a subset of the dimensions of a high-dimensional phase space (Maass et al., 2007); *attractor landscapes* shaped by a control parameter (input) describe dynamics of a system which lead to the appearance and disappearance of attractors due to incessant bifurcations (Negrello and Pasemann, 2008).

This is only a subset of all the different proposed mechanism for working memory. Biological brains might also end up using several of these phenomena simultaneously.

None of the above mentioned methods seem to address all the properties that we would expect from working memory. In what follows we will start by first providing a specific structure and learning rule that results in echo state models that can exhibit memory. This will serve as a proof of concept that while the mechanism behind this behaviour is not clear, the desired behaviour can be obtained.

5.4 Echo state networks with working memory

In this section we present a working memory model based on an ESN which uses a dynamical mechanism related to point attractors for storing information. It is able to simultaneously store information, use the stored information to modulate further storing, detect patterns in the input that trigger memory content switches, and perform input classification. For a refresher on the echo state network model, please see Section 2.3.6. The ESN refers to a recurrent network for which we learn only the output weights and carefully sample the input and recurrent weights.

5.4.1 Model

Our model is obtained by adding a set of special output units to an otherwise standard ESN. We called these units *WM-units*, and they differ from normal output units by having trainable connections from one to another or to themselves, as in Figure 5.3. In our setup we only allow feedback connections from the WM-units to the reservoir¹ but not from the other, regular output units. Another difference between WM-units and output units is that the former are binary-state neurons which can store memory bits. To achieve this behaviour we use a sharp threshold function $\sigma^{(mem)}$ as the activation function of these units:

$$\sigma^{(mem)}(x) = \begin{cases} -0.5 & x \leq 0. \\ +0.5 & x > 0. \end{cases} \quad (5.2)$$

The network is described by Equations (5.3), (5.4) and (5.5). Note the addition of the WM-units \mathbf{m} , as well as the feedback connections from the memory units $\mathbf{W}^{(feedback)}$. Compared to a standard RNN, we also have direct connections from the input to the output units or memory units and connections between the memory units. For clarity in notation these connections were folded in to the matrix $\mathbf{W}^{(out)}$ and $\mathbf{W}^{(mem)}$ respectively. We also do not use biases (we do, however, force one of the input units to have a constant negative value), and rely on the tanh activation function. The only learnt weights of the model are $\mathbf{W}^{(mem)}$ and $\mathbf{W}^{(out)}$.

$$\mathbf{h}_{[n+1]} = \sigma(\mathbf{W}^{(in)}\mathbf{u}_{[n+1]} + \mathbf{W}^{(rec)}\mathbf{h}_{[n]} + \mathbf{W}^{(feedback)}\mathbf{m}_{[n]}) \quad (5.3)$$

1. In the Reservoir Computing literature, the hidden state of a recurrent network is usually referred to as a reservoir

$$\mathbf{y}_{[n+1]} = \sigma^{(out)} \left(\mathbf{W}^{(out)} \begin{bmatrix} \mathbf{u}_{[n+1]} \\ \mathbf{h}_{[n+1]} \end{bmatrix} \right) \quad (5.4)$$

$$\mathbf{m}_{[n+1]} = \sigma^{(mem)} \left(\mathbf{W}^{(mem)} \begin{bmatrix} \mathbf{u}_{[n+1]} \\ \mathbf{h}_{[n+1]} \\ \mathbf{m}_{[n]} \end{bmatrix} \right). \quad (5.5)$$

Training this model is similar to the standard training of ESNs. Because the WM-units feed back into the reservoir, we need the value of the WM-units at time $t - 1$ in order to compute the state of the reservoir $\mathbf{h}_{[t]}$. We use a *teacher-forcing* approach of relying on the true targets on the WM-units to get the activations of the reservoir for some input sequence.

There is one important observation to be made here. First of all, in order to train the model we need targets for the WM-units. This means that their meaning and behaviour is predefined and not learnt. We use *hints* that tell us how the memory units should behave for some sequence. A similar approach is used in [Gulcehre and Bengio \(2013\)](#). This model, therefore, can not be used on some dataset for which we do not know *what, when* and *for how long* we need to memorize some information. As such, the utility of the model is limited. However, the model itself is interesting, because, putting aside the learning problem, it offers evidence that stable working memory can be obtained in recurrent networks. These memory traces can, as we will show, continuously influence the ongoing processing, keep the information stored for unbounded periods of time and the model can learn to replace stored information when it receives different input cues. As such, our model, exhibits all the pre-requisites we have enumerated in [Section 5.3](#). An analysis of the model provides intuitions about the internal mechanism that result in this behaviour, a particular type of attractor-like phenomena to which we refer as *input-induced attractor*.

Because of the sharp activation of the memory units we do not need to add noise to the teacher signal¹ when we use it to compute the activation of the reservoir. However, if we forego the sharp activation function, this form of regularization is vital to get the memory units to be stable and not quickly diverge in the presence

1. When relying on a teacher-forcing strategy to train a model, we feedback through the recurrent connections of a model the target behaviour of a unit instead of its actual value. This target value is usually called the teacher signal.

of noise or new input sequences.

To compute the output weights $\mathbf{W}^{(out)}$, the reservoir state vectors, together with the activations of the input units are stored row-wise in a data collection matrix \mathbf{G} . If $\mathcal{Y}^{(target)}$ is the target signal, then the output weights are computed using linear regression as shown in Equation (5.6), where \dagger stands for pseudo-inverse. A similar process is done for learning the memory units $\mathbf{W}^{(mem)}$, where the inverse of the activation function is taken as being the identity.

$$\mathbf{W}^{(out)} = (\mathbf{G}^\dagger \cdot \sigma^{(out)^{-1}}(\mathcal{Y}_{(target)}))^T. \quad (5.6)$$

An important condition to make the learning of output weights by regression a well-defined procedure is the *echo state property* (ESP). This is a property of the reservoir and the admitted input. Roughly stated, a reservoir has the ESP with respect to a given admitted input range if for any infinite input sequence the network states $\mathbf{h}_{[n]}$ asymptotically forget the (arbitrary) initial state $\mathbf{h}_{[0]}$ used at startup time. Formal definitions of the ESP are given in Jaeger (2001), and refined algebraic conditions are in Buehner and Young (2006); Manjunath and Jaeger (2013). In practice, the ESP is usually ensured when the spectral radius of the reservoir weight matrix $\mathbf{W}^{(rec)}$ is set to a value below unity, but we emphasize that this is neither a necessary nor a sufficient criterion (Jaeger, 2007b), in spite of a folklore belief in the field that it is both. The value of 1 for the spectral radius is sometimes also referred to as the edge of chaos, though going over a spectral radius of 1 does not imply a chaotic regime.

Depending on the task that needs to be solved, there are a few global parameters that need to be tuned for optimal learning, namely, global scalings of input weights, reservoir weights, and output feedback weights. In the reservoir computing field, the global scaling of the reservoir weights $\mathbf{W}^{(rec)}$ is typically specified through the spectral radius of this matrix. All these tunable parameters are explained in more detail in Jaeger (2001).

At first sight, the strong couplings between WM-units through the trained $\mathbf{W}^{(mem)}$ might appear problematic for a clean¹ storing of memory items, because in technical storage devices one does not usually desire dynamical interaction between stored items. However, we will demonstrate that such interactions can be

1. By clean we mean that there is no interference with the stored information due to the ongoing processing of information of the model.



Figure 5.4: A fragment example of the rich graphic script used as input. The image was scaled for better visualization.

harnessed for realizing desirable processing functionalities which go beyond pure storage and retrieval.

5.4.2 Experiments

Task. The task is to keep track of the number of opened curly brackets as the system reads a rich graphic script input, one vertical pixel line per timestep. An input sample is shown in Figure 5.4. The system is required to maintain a counter. Any time a opened curly bracket appears at the input, the counter has to increment by 1. When a closed curly bracket appears the counter needs to decrement by 1. The architecture is required to be able to count up to 6. The input data are generated such that no overflow or underflow occurs. This task requires a persistent memory, as the network must remember the number of brackets seen for unbounded periods of time, but it also requires the ability to do the basic arithmetic operations of adding and subtracting 1.

In addition to this, as a computational payload the network also has to predict the next character. This functionality is trained into the normal output units. We use the same number of output units as the number of possible characters (excepting the curly bracket characters). Each output unit predicts how probable it is that the corresponding character will follow in the input stream. In doing this, the network will benefit from taking into account the current bracket level (the number of unclosed brackets) since the conditional distribution of the next character given the previous differs across the bracketing levels. This additional task is meant to demonstrate that the network is able to use the information stored in the WM-units.

Data. We train the network in two stages. During the first stage only $\mathbf{W}^{(mem)}$ is computed. In this stage training sequences of only 10000 symbols are used. In the second stage the weights $\mathbf{W}^{(out)}$ to the output units are computed. For the second stage we use sequences of 49000 characters. In both cases the input sequences are generated in exactly the same way.

For generating the sequences, characters are chosen randomly with a probability of 70% from a set of 65 different ASCII symbols (letters in lower case, numbers and other symbols typically used in text including other types of brackets), with a probability of 15% an open curly bracket and with 15% a closed curly bracket. The opening and closing curly brackets are inserted such that they form matching pairs with a nesting level of up to 6. According to the nesting level i (which can be between 0 and 6), a different Markov chain is used to sample from the other 65 characters. The Markov chain is defined such that if the current character has the index j (a number between 1 and 65) then the next character will be $j + i + 1$ modulo 65 with probability 80% and with equal probability (0.3125%) any of the other 64 characters.

The testing sequence is generated similarly. It has 35000 characters, picked now with a probability of 94% from the same set of 65 ASCII characters, while curly brackets are picked with a probability of only 6%. The same 7 Markov chains are used to sample characters in the periods corresponding to the 7 bracket levels.

The symbols are transformed afterwards to images by printing them with a randomly selected font from four different font sets (FreeMono, FreeMono Bold, FreeMono Italic and FreeMono Bold Italic of Gimp 2.3.6). All fonts have a width of 7 pixels and height of 12 pixels, where each pixel is a grayscale value between 0 and 1. Before printing, the character images are stretched randomly to a width of 6, 7 or 8 pixels. Salt-and-pepper noise with an amplitude of 0.1 is added. The final image-per-symbol has a fixed height of 12 pixels and a varying width. The resulting script video sequence is fed to the network one vertical line at a time step through 12 input units. The testing data are more challenging than the training data in the sense that switches between curly bracket levels occur more rarely, which means that the WM must maintain the current bracket level for longer periods. More precisely, in our training sequence this period ranged between 0 to 248 cycle updates, with a mean of 17.7, while in testing data the period ranges between 0 to 691 cycles with a mean of 96.8. Note that while the average amount of time spent in a state during training is well within the reach of the innate, fading short term memory of the reservoir (Jaeger, 2002), this is not the case for the testing data.

Architecture detail. The model used has 13 input units, 12 representing a line of the data, while the last unit feeds a constant bias of -0.5. The input to reservoir connections are sparse, 80% of them being 0. The rest are randomly chosen to be

either -0.5 or 0.5 with equal probability. The reservoir has 1200 units with only 12000 non-zero connections. The non-zero weights are either -0.1540 or +0.1540 with equal probability. The reservoir weight matrix has a spectral radius of 0.5. The activation function of the reservoir's units is \tanh . The number of units as well as the spectral radius was chosen so as to maximize the performance of the model. We remark that these values are a compromise between the requirements of the different subprocesses that go on simultaneously in the network. The task that the network is asked to solve requires the network to recognize characters, to memorize the number of unclosed curly brackets and to be able to do basic arithmetic operations.

Six WM-units are connected to the reservoir. The feedback weight matrix from the WM-units units to the reservoir is dense, all weights being randomly picked to be either -.4 or +.4. The value k of the counter is represented in the WM-units by having the first k units at +0.5 while the rest are at -0.5. The value 0 means that all WM-units are at -0.5. The network is able to represent the numbers 0, 1, 2, 3, 4, 5 and 6. The WM-units have the activation function described in equation 5.2. In addition to this, 65 ordinary output units are connected to the reservoir, with no feedback to the reservoir. Their value (as well as the target) is only defined at the time step when the input switches between characters, at which points the target is 0 everywhere except for the value corresponding to the next character which is 1.

Training. A teacher signal (target) for the 6 WM-units is generated for the training sequence, which represents the prescribed -0.5/0.5 switching as curly brackets appear in the input. The target switches occur in the middle of the presentation of a bracket character. The training is done by computing the output weights such that the distance between the output and the target is minimized in the mean square error sense. The algorithm used is the Wiener-Hopf method.

The same approach is used to train the output units. The only difference is that we only use the input and reservoir activations for the time steps where the input switches from one character to another.

Results. We ran the experiment 30 times, each time with different randomly generated training and testing sequences, and freshly randomly generated reservoir, input and feedback weights.

We started by inspecting the performance of the WM-units. An appropriate

measure of the performance of the memory performance is the number of mistakes made. As a mistake we consider events where the WM-unit state is different from the target. We do not check for mistakes during the presentation of a curly bracket (i.e., we do not evaluate transient effects within the timespan of a curly bracket). Once a mistake is present, we count it and then correct the state of the network. To do so, we only correct the WM-units state by externally forcing them for one time step to the desired configuration; the feedback connections then will also correct the reservoir’s state. We sorted the errors in false positives (when the network detects a bracket character even though none is present in the input) and false negatives (when the network fails to detect a bracket). Table 5.1 lists the error counts.

Type of error	Number of errors	Percentage of curly brackets	Percentage of characters	Percentage of time steps
false negatives	7.2 ± 6.5	$0.34 \pm 0.30\%$	$0.02 \pm 0.018\%$	$0.003 \pm 0.002\%$
false positives	59.8 ± 21.6	$2.84 \pm 1.02\%$	$0.17 \pm 0.061\%$	$0.024 \pm 0.008\%$
total	67.0 ± 22.9	$3.18 \pm 1.09\%$	$0.19 \pm 0.065\%$	$0.027 \pm 0.009\%$

Table 5.1: Number of erroneous WM states obtained by the ESN, averaged over 30 runs

At a closer inspection of the errors produced in the 30 runs, we found that the network never changed the WM-units to an invalid (non-coding) state or by increasing or decreasing the counter by more than one. This suggests that any error is actually the result of misclassification of a character, and not by the other subprocesses of WM-unit state management (adding/subtracting one or keeping a certain value stable).

Following up on this observation, we further differentiated the number of false positives according to which character triggered the error. Table 5.2 shows these results, which coincide with our intuition of when the recognition subtask might fail.

Another question one might raise is if correcting only the WM-units state is sufficient for correcting the state of the network. If this were not the case we would expect several errors to occur in rapid succession in a short timespan (during the same character presentation or over two consecutive characters). Such errors would also suggest instability of WM-unit locking. But such scenarios never happened in any of the 30 runs.

Character (number of characters in the testing sequences)	Number of times the counter increased	Number of times the counter decreased
“(” (499.5 ± 22.3)	21.5 ± 10.1	0 ± 0
“)” (502.4 ± 18.6)	0 ± 0	0.5 ± 0.2
“[” (496.2 ± 22.8)	5.8 ± 5.1	0 ± 0
“]” (501.3 ± 15.1)	0.05 ± 0.03	6.0 ± 5.4
“@” (492.7 ± 21.3)	25.1 ± 14.1	0.2 ± 0.1
other	0.05 ± 0.04	0.6 ± 0.5

Table 5.2: Trigger characters for false positives, averaged over 30 runs

We also measured the average absolute value of the computed \mathbf{W}^{mem} weights, i.e. the weights leading to the WM-units (Table 5.3). Their modest size is indicative of a robust generalization, which indeed was observed, since the testing data were more challenging than the training data.

considered weights	average absolute value
input to WM-units weights	0.2327 ± 0.1813
reservoir to WM-units weights	0.0667 ± 0.0591
WM-units to WM-units weights	0.5825 ± 0.5627

Table 5.3: Average learned output weights of the ESN (over 30 runs).

The payload task our architecture had to solve was to predict the next input character. In order to measuring the performance of the network on this task, we considered as the predicted next character the most probable one (the one that corresponded to the output unit with the maximal score). The performance is quantified by simply counting the number of erroneous predictions. Note that the output units do not feed back to the reservoir and therefore a bad prediction will not affect any of the following predictions.

At any character switching step in between curly brackets, the next character k is selected according to a distribution that puts 80% weight on a single character. What we ask the network to do is to learn 65 such peaked conditional next-character distributions for each bracket level, in total 455 different distributions. Assuming that the network learns them perfectly, due to the deterministic approach of selecting the prediction of the next character, the network will always pick the character that has 80% weight, yielding an error rate of 20%, which is the best performance

that we can expect to achieve on this task.

The error rate that we found on average over the 30 runs was 24.83 ± 0.27 %.

To demonstrate the importance of the WM-units in achieving this performance level, we ran the same experiments with a ESN that had no WM-units but was otherwise set up similarly. What we expect to happen is that the network will not be able to distinguish between bracket levels anymore. Assuming that the current character is j , the network would learn in this case a distribution that gives a larger, almost equal, probability to $j + 1$ modulo 65, $j + 2$ modulo 65, ..., $j + 7$ modulo 65 (the most probable characters for the different bracket levels), and much smaller equal probability to all other characters. This implies that on average across the different bracketing levels (which cannot now be memorized for longer time spans) the network is likely to give wrong predictions in $20 + (6/7) * 80 \approx 80.5\%$ of the cases. We found an error rate of 83.75 ± 0.11 % in this condition, close to what we expected.

5.4.3 Analysing the model

In the light of the discussion carried out in Section 5.3, one question that we need to address is what is the underlying dynamical mechanism that provides this working memory behaviour that we were able to simulate with this model.

First of all, we make the observation that the memory is stable¹. As long as no curly bracket appears in the input, the model preserves the current bracket level reliably for very large amounts of time. This suggests some form of attractor-like behaviour.

There are some difficulties with this claim, as attractors are rigorously defined only for autonomous systems. It was previously suggested in [Bengio et al. \(1994\)](#), for example, that one can regard the input as bounded noise, which allows a straightforward extension of the notion. This is not a perfect solution, as discussed previously, as the input is highly structured and the model is trained to respond to it. In [Pascanu and Jaeger \(2011\)](#) we attempt to provide a definition of pseudo-attractors, called γ -attractors, for input driven dynamical systems (which

1. By stable we mean that the model can remember the stored information for what seems very long periods of time, longer than what a standard model can do. Unfortunately there is no theoretical quantification of what stable means in this case and we rely only on empirical evidence.

we refer to as γ -systems). For now let us consider simpler approach of simply ignoring the input and assume it is just bounded noise.

If the WM-units seem locked in an attractor state, not the same can be said about the rest of the network. The hidden units of the model, or at least a subset of them, seem to be free. These units are used to solve the payload task. Because there is a strong connection between the behaviour of the model and the number of open brackets, one might be tempted to fold the current behaviour into the attractor. This means that the current behaviour is confined to the support of the corresponding attractor and hence the whole model is locked into this attractor. For this to be true, the attractor itself has to be fairly complex. However, preliminary experimentation showed that the payload task can be independent of the memory content, and the number of open brackets, in such a situation, does not influence the performance on the payload task. This clearly suggests that at least some of the hidden units are not constrained by the attractor representing the number of open brackets. If we negate this claim, then the different attractors corresponding to different brackets levels would have to share some of their support set, which contradicts the definition of an attractor.

This suggests that the kind of attractor-behaviour that we observe is similar to the one reported by [Maass et al. \(2007\)](#), namely a form of high-dimensional attractor or partial attractor.

Another important trait of the trained model is the switching mechanism between different attractor states. This does not happen *randomly*, but rather when specific patterns appear in the input. This suggests that the input, for these attractors, is not equivalent to noise.

Figure 5.5 looks at the role of the input into the behaviour of the model. The plot was obtained as follows:

- We ran the WM model from the previous section 7 times for approximately 45,000 network updates, each time with the memory units clamped in one of the 7 settings coding for one bracketing level; the driving input was in each case generated from an input character sequence whose Markov chain properties were the same as used in the previous section, not containing curly brackets;
- the obtained 7 sets of reservoir states and input vectors were concatenated and the first principal components (PCs) of the reservoir states and inputs

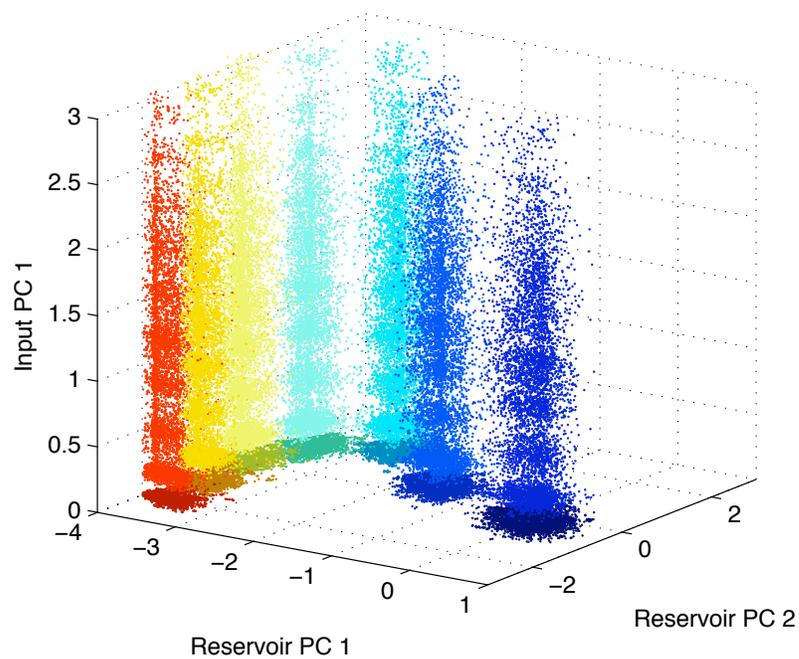


Figure 5.5: Visualization of the memory states of the WM model. The first PC of input signals is plotted against the first two PCs of reservoir states, for the 7 WM unit configurations described in the previous section. Different colors correspond to different WM configurations. Projections of the reservoir state PCs are shown in darker shading on the ground plane. 6000 points are plotted per attractor. Notice that the value ranges do no longer correspond to the $(-1, 1)$ range of tanh reservoirs because we display projections on the PCs. Picture best seen in color. For detail compare text.

were computed;

- separately for each of the 7 datasets, the first PC of the inputs was plotted against the first two PCs of the reservoir states.

One sees that even in only the first two reservoir PCs, the reservoir state sets corresponding to the different memory configurations become very well separated. This suggests that additionally to have only part of the system locked in an attractor-like state, these states seem to be stable under most inputs, with the exception of the learnt switching patterns, which allow, within one step, for the model to leave the basin of attraction of the current attractor. We do not have a proper mathematical understanding of this behaviour. In [Pascanu and Jaeger \(2011\)](#) we do, however, provide a formalization of these observations. Specifically we define an input-driven system called γ -system, and define, for these systems, γ -attractors which have all the properties that we see empirically in our experimentations.

Further work is needed to understand such phenomena. For example, one would need better mathematical tools that would enable a deeper understanding of such behaviour. A more in depth comparison of γ -attractors with other similar observed phenomena, and, if possible, a common framework that describes all these different phenomena would also be useful.

For practical applications it would be useful to understand when these attractors can manifest themselves (what properties of the parameters are required for such phenomena to be possible). How can we learn them? What is the capacity of these attractors as a function of the model size? Can they only represent discrete information? Most of these questions we leave for future work.

5.5 Learning: The exploding and vanishing gradients problem

Ideally we want to be able to train models that exhibit working memory without relying on hints. One way of analysing the ESN model with WM-units is to consider the WM-units as hidden units. We can construct the corresponding recurrent weight matrix that includes the feedback connections from the WM-units to the reservoir and the weights among the WM-units, and transform the model

into a classical recurrent neural network. The question is now: Can we learn this model by stochastic gradient descent from initial small weights?

The final trained network can not have the echo state property, as it exhibits several attractor-like phenomena. The phase portrait of the dynamical system¹ associated to this recurrent network has to be rich². Therefore, learning should be able to move a model from small initial weights to one with rich dynamical behaviour. It is to be expected (say if the largest singular value of the recurrent weight matrix is smaller than unity) that the initial model has the echo state property and hence its phase portrait has a single point attractor at the origin. Even if this is not the case, is hard to image that the topology of the phase portrait of the freshly initialized model is, by chance, the same as the one of the trained model. This means that, during learning, the model has to traverse bifurcation boundaries and to move through the phase space of a behaviourally rich model. This implies that it is likely for learning to face both the *exploding gradient* and *vanishing gradient* problems.

Let us further discuss these two problems and analyze how learning might be able to deal with them. In this section of the chapter we will make use of the following specific parametrization of a recurrent neural model to facilitate our analysis³:

$$\mathbf{h}_{[t]} = \mathbf{W}^{(rec)}\sigma(\mathbf{h}_{[t-1]}) + \mathbf{W}^{(in)}\mathbf{u}_{[t]} + \mathbf{b} \quad (5.7)$$

Let us re-write the gradients of a recurrent neural network in order to better highlight the exploding or vanishing gradients problem:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{1 \leq t \leq T} \frac{\partial \mathcal{L}_{[t]}}{\partial \theta} \quad (5.8)$$

$$\frac{\partial \mathcal{L}_{[t]}}{\partial \theta} = \sum_{1 \leq k \leq t} \left(\frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{h}_{[t]}} \frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}} \frac{\partial^+ \mathbf{h}_{[k]}}{\partial \theta} \right) \quad (5.9)$$

1. We can assign an autonomous dynamical system to a recurrent network by ignoring the input and regard it as *bounded noise*. While this is not ideal, it does provide some intuition of how the model behaves.

2. We refer to a phase portrait as being rich if there is more than one attractor

3. The more widely known formulation, also used in other sections of this work, is $\mathbf{h}_{[t]} = \sigma(\mathbf{W}^{(rec)}\mathbf{h}_{[t-1]} + \mathbf{W}^{(in)}\mathbf{u}_{[t]} + \mathbf{b})$. Both formulations behave the same (e.g. by redefining $\mathcal{L}_{[t]} := \mathcal{L}_{[t]}(\sigma(\mathbf{h}_{[t]}))$). We chose Equation (5.7) for convenience.

$$\frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}} = \prod_{t \geq i > k} \frac{\partial \mathbf{h}_{[i]}}{\partial \mathbf{h}_{[i-1]}} = \prod_{t \geq i > k} \mathbf{W}^{(rec)T} \text{diag}(\sigma'(\mathbf{h}_{[i-1]})) \quad (5.10)$$

These equations are obtained by writing the gradients in a sum-of-products form. The derivative $\frac{\partial^+ \mathbf{h}_{[k]}}{\partial \theta}$ refers to the “immediate” partial derivative¹ of the state $\mathbf{h}_{[k]}$ with respect to θ , where $\mathbf{h}_{[k-1]}$ is taken as a constant with respect to θ . Specifically, considering Equation (5.9), the value of any row i of the matrix $(\frac{\partial^+ \mathbf{h}_{[k]}}{\partial \mathbf{W}^{(rec)}})$ is just $\sigma(\mathbf{h}_{[k-1]})$. Equation (5.10) also provides the form of Jacobian matrix $\frac{\partial \mathbf{h}_{[i]}}{\partial \mathbf{h}_{[i-1]}}$ for the standard parametrization of a recurrent neural network, where diag converts a vector into a diagonal matrix, and σ' computes element-wise the derivative of σ .

Any gradient component $\frac{\partial \mathcal{L}_{[t]}}{\partial \theta}$ is also a sum (see Equation (5.9)), whose terms we refer to as *temporal* contributions or *temporal* components. One can see that each such temporal contribution $\frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{h}_{[t]}} \frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}} \frac{\partial^+ \mathbf{h}_{[k]}}{\partial \theta}$ measures how θ at step k affects the cost at step $t > k$. The factors $\frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}}$ (Equation (5.10)) transport the error “in time” from step t back to step k . We further loosely distinguish between *long term* and *short term* contributions, where long term refers to components for which $k \ll t$ and short term to everything else.

As introduced in Bengio et al. (1994), the *exploding gradient* problem refers to the large increase in the norm of the gradient during training. Such events are caused by the explosion of the long term components, which grow exponentially then short term ones.

The *vanishing gradients* problem refers to the opposite behaviour, when long term components go exponentially fast to norm 0, making it impossible for the model to learn correlation between temporally distant events.

5.5.1 The mechanics

To understand this phenomenon we need to look at the form of each temporal component of the gradient, and in particular at the factors $\frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}}$ (see equation (5.10)) that take the form of a product of l Jacobian matrices, with $l = t - k$. Intuitively, these products can grow exponentially fast with l (in some direction \mathbf{v}), leading to the explosion of long term components when l is large. Because

1. We use “immediate” partial derivatives in order to avoid confusion, though one can use the concept of total derivative and *the proper meaning* of partial derivative to express the same property.

the gradient is just a sum of these components, it follows that it should also grow exponentially fast following the long term component with $k = 0$ (for which $l = t$).

If all the matrices involved in this product have, on the other hand, the largest eigenvalue smaller than 1, then their product will have the opposite effect. It will shrink exponentially fast along any direction.

5.5.2 Linear model

Let us consider the term $\mathbf{g}_{[k]}^T = \frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{h}_{[t]}} \frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}} \frac{\partial^+ \mathbf{h}_{[k]}}{\partial \theta}$ for the linear version of the parametrization in equation (5.7) (i.e. set σ to the identity function) and assume t goes to infinity. We have that:

$$\frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}} = \left(\mathbf{W}^{(rec)T} \right)^l \quad (5.11)$$

By employing the same approach as the *power iteration method* we can show that, given certain conditions, $\frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{h}_{[t]}} \left(\mathbf{W}^{(rec)T} \right)^l$ grows exponentially.

Proof. Let $\mathbf{W}^{(rec)}$ have the eigenvalues $\lambda_{[1]}, \dots, \lambda_{[n]}$ with $|\lambda_{[1]}| > |\lambda_{[2]}| > \dots > |\lambda_{[n]}|$ and the corresponding eigenvectors $\mathbf{e}_{[1]}, \mathbf{e}_{[2]}, \dots, \mathbf{e}_{[n]}$ which form a vector basis. We can now write the row vector $\frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{h}_{[t]}}$ in terms of this basis:

$$\frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{h}_{[t]}} = \sum_{i=1}^N c_{[i]} \mathbf{e}_{[i]}^T$$

If j is the smallest index for which $c_{[j]} \neq 0$, using the fact that

$$\mathbf{e}_{[i]}^T \left(\mathbf{W}^{(rec)T} \right)^l = \lambda_{[i]}^l \mathbf{e}_{[i]}^T,$$

we have that as l goes to infinity the following approximation becomes more exact:

$$\frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{h}_{[t]}} \frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}} = c_{[j]} \lambda_{[j]}^l \mathbf{e}_{[j]}^T + \lambda_{[j]}^l \sum_{i=j+1}^n c_{[i]} \frac{\lambda_{[i]}^l}{\lambda_{[j]}^l} \mathbf{e}_{[i]}^T \approx c_{[j]} \lambda_{[j]}^l \mathbf{e}_{[j]}^T. \quad (5.12)$$

We used the fact that $\left| \lambda_{[i]} / \lambda_{[j]} \right| < 1$ for $i > j$, which means that

$$\lim_{l \rightarrow \infty} \left| \lambda_{[i]} / \lambda_{[j]} \right|^l = 0.$$

If $|\lambda_{[j]}| > 1$, it follows that $\frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}}$ grows exponentially fast with l , and it does so along the direction $\mathbf{e}_{[j]}$. \square

The proof assumes $\mathbf{W}^{(rec)}$ is diagonalizable for simplicity, though using the Jordan normal form of $\mathbf{W}^{(rec)}$ one can extend this proof by considering not just the eigenvector or largest eigenvalue but the whole subspace spanned by the eigenvectors sharing the same (largest) eigenvalue.

This result provides a necessary condition for gradients to grow in the linear case, namely that the spectral radius (the absolute value of the largest eigenvalue) of $\mathbf{W}^{(rec)}$ must be larger than 1. We will come back to this condition later on and look at it in the nonlinear case as well.

If $\mathbf{e}_{[j]}$ is not in the null space of $\frac{\partial^+ \mathbf{h}_{[k]}}{\partial \theta}$ the entire temporal component grows exponentially with l . The matrix $\frac{\partial^+ \mathbf{h}_{[k]}}{\partial \theta}$ can be thought of as rotating the vector $\mathbf{e}_{[j]}$ and scaling it by some factor $\gamma_{[k]}$, none of which should interact with the exponential scaling as long as $\gamma_{[k]}$ does not shrink exponentially fast (which is true for the linear case).

This approach extends easily to the entire gradient. If we re-write it in terms of the eigen-decomposition of \mathbf{W} , we get:

$$\frac{\partial \mathcal{L}_{[t]}}{\partial \theta} = \sum_{j=1}^n \left(\sum_{i=k}^t c_{[j]} \lambda_{[j]}^{t-k} \mathbf{e}_{[j]}^T \frac{\partial^+ \mathbf{h}_{[k]}}{\partial \theta} \right) \quad (5.13)$$

We can now pick j and k such that $c_{[j]} \mathbf{e}_{[j]}^T \frac{\partial^+ \mathbf{h}_{[k]}}{\partial \theta}$ does not have 0 norm, while maximizing $|\lambda_{[j]}|$. If for the chosen j it holds that $|\lambda_{[j]}| > 1$ then $\lambda_{[j]}^{t-k} c_{[j]} \mathbf{e}_{[j]}^T \frac{\partial^+ \mathbf{h}_{[k]}}{\partial \theta}$ will dominate the sum and because this term grows exponentially fast to infinity with t , the same will happen to the sum.

5.5.3 Nonlinear model

To generalize this result to the nonlinear case, we define the concept of expanding and non-expanding matrices for some direction \mathbf{v} . We say that the Jacobian matrix $\frac{\partial \mathbf{h}_{[i]}}{\partial \mathbf{h}_{[i-1]}}$ expands along a vector \mathbf{v} by $\alpha > 1$ if the following inequality holds.

$$\forall \mathbf{u}, \left| \mathbf{u}^T \frac{\partial \mathbf{h}_{[i]}}{\partial \mathbf{h}_{[i-1]}} \mathbf{v} \right| > \alpha |\mathbf{u}^T \mathbf{v}| \quad (5.14)$$

Intuitively we need two kinds of sufficient properties to hold for the product of Jacobian matrices to expand exponentially fast in some direction \mathbf{v} . One is that the product of these conditions expands exponentially with t , e.g., it would be achieved if as $t \rightarrow \infty$ the number of expanding Jacobian matrices $\frac{\partial \mathbf{h}_{[i]}}{\partial \mathbf{h}_{[i-1]}}$ increases, the number of non-expanding ones remains finite. The second property is that multiplying by $\frac{\partial \mathbf{h}_{[i]}}{\partial \mathbf{h}_{[i-1]}}$ does not kill off these exponentially large increases.

We show this by constructing the set P of matrices that are non-expanding, and considering a lower bound $\beta > 0$ on how much these matrices shrink a vector in the direction \mathbf{v} , condition formalized in the next inequality:

$$\forall \mathbf{u}, \left| \mathbf{u}^T \frac{\partial \mathbf{h}_{[i]}}{\partial \mathbf{h}_{[i-1]}} \mathbf{v} \right| > \beta |\mathbf{u}^T \mathbf{v}|, \text{ iff } \frac{\partial \mathbf{h}_{[i]}}{\partial \mathbf{h}_{[i-1]}} \in P \quad (5.15)$$

This means that if α is the least amount by which any matrix $\frac{\partial \mathbf{h}_{[i]}}{\partial \mathbf{h}_{[i-1]}} \notin P$ expands, $\frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}}$ should expand roughly by $\beta^{|P|} \alpha^{t-|P|}$. If the cardinality of P is bounded as t grows, it means this product grows exponentially fast with $t - |P|$.

It is worth mentioning that α is bounded by the largest singular value of each matrix $\frac{\partial \mathbf{h}_{[i]}}{\partial \mathbf{h}_{[i-1]}}$ (which is easy to see as $\left\| \frac{\partial \mathbf{h}_{[i]}}{\partial \mathbf{h}_{[i-1]}} \mathbf{v} \right\| \leq \left\| \frac{\partial \mathbf{h}_{[i]}}{\partial \mathbf{h}_{[i-1]}} \right\| \|\mathbf{v}\|$). If we consider the parametrization in equation (5.7), this largest singular value is in its turn bounded by the product of the largest singular values $\rho_{\mathbf{W}^{(rec)}}$ of $\mathbf{W}^{(rec)}$ and $\rho_{\sigma'}$ of $\text{diag}(\sigma'(\mathbf{h}_{[i-1]}))$. We know that $\rho_{\sigma'} < 1$ for tanh and $\rho_{\sigma'} < 1/4$ for the sigmoid function, and hence we recover a necessary condition for the gradients to explode, namely that $\rho_{\mathbf{W}^{(rec)}} > 1$ (with the tighter version for the sigmoid, $\rho_{\mathbf{W}^{(rec)}} > 4$).

It is also sufficient for the following two equations, namely equations (5.16) and (5.17), to hold, where the first equation implies that our chosen direction \mathbf{v} is not in the null space of $\frac{\partial^+ \mathbf{h}_{[k]}}{\partial \theta}$, while the second equation writes the vector $\frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{h}_{[t]}}$ in a orthonormal vector basis $\mathbf{v}_{[1]}, \dots, \mathbf{v}_{[N]}$, where $\mathbf{v}_{[1]} = \mathbf{v}$ and $c_{[i]}^{(loss)} \in \mathbb{R}$.

$$\forall \mathbf{u} \in \mathbb{R}^N, \left| \mathbf{u}^T \frac{\partial^+ \mathbf{h}_{[k]}}{\partial \theta} \mathbf{v} \right| \geq \gamma_{[k]} |\mathbf{u}^T \mathbf{v}|, \gamma_{[k]} > 0 \quad (5.16)$$

$$\frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{h}_{[t]}} = \sum_{j=1}^N c_{[j]}^{(loss)} \mathbf{v}_{[j]}^T, c_{[1]}^{(loss)} \neq 0. \quad (5.17)$$

Using these relations we can find a lower bound for $|\mathbf{g}_{[k]}^T \mathbf{v}|$, where $\mathbf{g}_{[k]}$ is the temporal component corresponding to time step k . The equation below shows a few steps of this derivation, where with out loss of generality we assigned the first

element to P , but not the second one.

$$\begin{aligned}
|\mathbf{g}_{[k]}^T \mathbf{v}_1| &\geq \left| \left(\frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{h}_{[t]}} \frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}} \right) \frac{\partial^+ \mathbf{h}_{[k]}}{\partial \theta} \mathbf{v}_{[1]} \right| \\
&\geq \gamma_{[k]} \left| \left(\frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{h}_{[t]}} \frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k+1]}} \right) \overbrace{\frac{\partial \mathbf{h}_{[k+1]}}{\partial \mathbf{h}_{[k]}}}^{\in P} \mathbf{v}_{[1]} \right| \\
&\geq \gamma_{[k]} \beta \left| \left(\frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{h}_{[t]}} \frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k+2]}} \right) \overbrace{\frac{\partial \mathbf{h}_{[k+2]}}{\partial \mathbf{h}_{[k+1]}}}^{\notin P} \mathbf{v}_{[1]} \right| \\
&\geq \gamma_{[k]} \beta \alpha \left| \left(\frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{h}_{[t]}} \frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k+3]}} \right) \frac{\partial \mathbf{h}_{[k+3]}}{\partial \mathbf{h}_{[k+2]}} \mathbf{v}_{[1]} \right| \\
&\dots \\
&\geq \gamma_{[k]} \beta^{|P|} \alpha^{l-|P|} \left| \frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{h}_{[t]}} \mathbf{v}_{[1]} \right| \\
&\geq \gamma_{[k]} \beta^{|P|} \alpha^{l-|P|} |c_{[1]}^{(loss)}| \geq C_k \alpha^{l-|P|}
\end{aligned} \tag{5.18}$$

Equation (5.18) ensures that long term components explode along \mathbf{v} as long as the coefficient C_k (where $C_k = \gamma_{[k]} \beta^{|P|} |c_{[1]}^{(loss)}|$) does not shrink faster to 0 than $\alpha^{l-|P|}$ grows to infinity. This is mostly a constraint on $\gamma_{[k]}$ (since $|P|$ is bounded from our initial assumption), which is determined by $\frac{\partial^+ \mathbf{h}_{[k]}}{\partial \theta}$. For a classical parametrization of the model the norm of the partial derivative $\frac{\partial^+ \mathbf{h}_{[k]}}{\partial \theta}$ is determined by the norm of the state and input at time k , where the constraint on the state roughly translates into not having the state going towards its saturated state faster than $\alpha^{l-|P|}$ (which can be satisfied for tanh and sigmoid).

To get the vanishing gradients problem, one simply needs to invert the proof. Instead of obtaining a necessary condition, we get a *sufficient* condition for the gradient to explode, namely, for tanh, that the largest singular value of the recurrent weight matrix is smaller than 1, or for sigmoid, smaller than 4.

5.5.4 The geometrical interpretation

Figure 5.6 shows how the norm of the stochastic gradient varies when training a recurrent network on the MuseData dataset (polyphonic music prediction, see Section 6.4) by SGD. The x-axis shows the number of stochastic gradient update steps, while y-axis shows the gradient norm. The main conclusion one can draw from it is that not only do the gradients explode, but at times, this happens very

quickly.

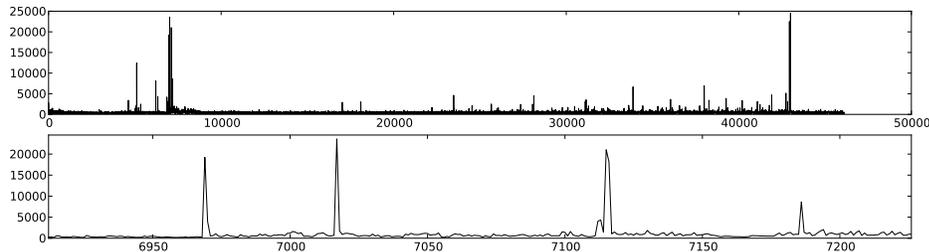


Figure 5.6: The norm of the gradient versus SGD steps for a 100 hidden units recurrent network when training on MuseData dataset. Top plot shows the norm for all steps taken until convergence. Bottom plot zooms in on one such peak to show how abrupt the change in norm is (up to 65-fold increase of the norm with respect to its mean).

If the gradient grows so quickly (along some direction), then so should the curvature. We can argue this is true in the single hidden unit case, where any long term component can be approximated by $C\alpha^{t-k}$ for some constant $C > 0$ (as suggested in Section 5.5.1). Its second derivative is $C(t-k)\alpha^{t-k-1}$. Using the same approach as before, for this case, we can see that when the gradient explodes so does the curvature.

It is not trivial to extend this reasoning to the high dimensional nonlinear case and we will not attempt to do this here, but we will use this observation to motivate that such scenarios might be likely to happen in general.

If this intuition is true, then it justifies that second order methods for training recurrent networks should do better (which seems to be true at least in the special case of the Hessian Free algorithm (Martens and Sutskever, 2011; Sutskever et al., 2011)).

The rapid growth in curvature also hints at a much simpler and cheaper solution for the exploding gradients which avoids computing the Hessian matrix. From this behaviour it follows that in the error landscape we likely have a wide valley for which at least one of the sides is like a steep wall, and the network is either inside the small slope area of the valley or on the edge of it, near where it can explode (Figure 5.7). This comes from the fact that if the explosion is caused by having the weights raised to some large power, then there is half of the space, when raising this parameters to a large power would result in their norm to vanish, and then

the other half where it would explode (for a single number think of having its value below 1 and above it).

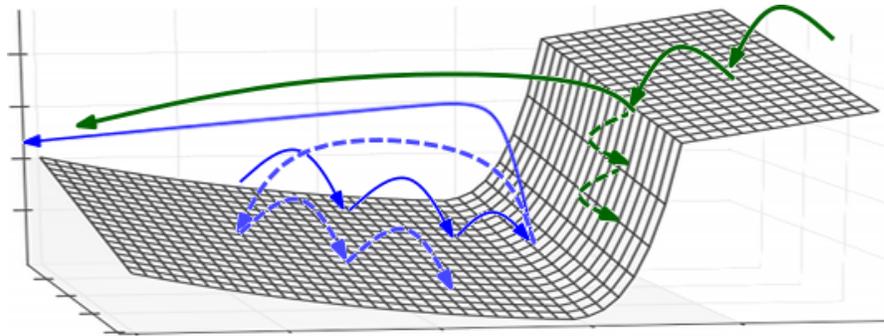


Figure 5.7: Illustrative example of a one-sided high curvature region in the error surface, where the model can be either inside the valley or on the edge of one of its walls. The solid lines depicts standard trajectories that gradient descent might follow. Using dashed arrow the diagram shows what would happen if the gradients is rescaled to a fixed size when its norm is above a threshold.

As the power iteration method shows for the linear case, one would expect the gradient to be aligned to the exploding direction \mathbf{v} , and, by our intuition, so should the curvature. This means that the steep wall of the valley is perpendicular to \mathbf{v} and the gradient. If we reach the wall and do a gradient descent step, we will jump across the valley moving perpendicular to its walls, disrupting the learning process. However we can do something different. In Figure 5.7, using dashed arrows, we depict another trajectory that gradient descent could take if we rescale the gradient to some fixed small step size when it explodes. This approach relies on the fact that this norm correction moves the model into a lower curvature region where a first order method is again suitable.

Note that the important addition in this scenario to the classical high curvature valley, is that we assume that locally the shape is not Gaussian (or symmetric). By our illustration we also do not mean to claim that this structure is the only possible one. It serves as an intuition of a possible scenario that could happen. Instead of this cliff shape, the error surface might just present a spike like structure ¹. We believe that the important observation (validated by our visualization of a really small model) is the lack of symmetry, which can be exploited to avoid divergence as we will show later.

1. This was suggested by Ilya Sutskever as a possible alternative

5.5.5 Drawing similarities with Dynamical Systems

One can consider yet another perspective, namely that of dynamical systems. Looking at dynamical systems theory for explaining the *exploding gradient* problem has been done before in [Doya \(1993\)](#). Here, we will attempt to extend and improve these previous observations.

For any parameter assignment θ , depending on the initial state $\mathbf{h}_{[0]}$, $\mathbf{h}_{[t]}$ (for an autonomous dynamical system) converges, under the repeated application of the map ρ given by the update rule of the recurrent model, to one of several possible different attractor states (e.g. point attractors). They describe the asymptotic behaviour of the model. The state space is divided into basins of attraction, one for each attractor. If the model is started in one basin of attraction it will converge to the corresponding attractor.

Dynamical systems theory tells us that as θ changes slowly, the asymptotic behaviour changes smoothly almost everywhere except for certain crucial points where drastic changes occur (the new asymptotic behaviour is no more topologically equivalent with the old one). These points are called bifurcation boundaries and are caused by attractors that appear, disappear or change shape.

Specifically, if we consider a simple model defined by equation (5.19), where we fix w to 5.0 and allow b to change, its bifurcation diagram is described by Figure 5.8. This model reproduces an illustration from [Doya \(1993\)](#). Such diagrams convey an abstract but complete picture of how the system can behave.

$$x_t = \sigma(wx_{t-1} + b) \tag{5.19}$$

The x-axis corresponds to the value of the parameter b . The bold line follows the movement of the final point attractor, $\mathbf{h}_{[\infty]}$, as b changes. At b_1 we have a bifurcation boundary where a new attractor emerges (when b decreases from ∞), while at b_2 we have another that results in the disappearance of one of the two attractors. In the interval (b_1, b_2) we are in a rich regime, where there are two attractors and the change in position of boundary between them, as we change b , is traced out by a dashed line. The vector field (gray dashed arrows) describe the evolution of the state \mathbf{h} if the network is initialized in that region.

There are two types of events that could lead to a large change in $\mathbf{h}_{[t]}$, when $t \rightarrow \infty$. One is crossing the boundary of a basin of attraction (depicted with

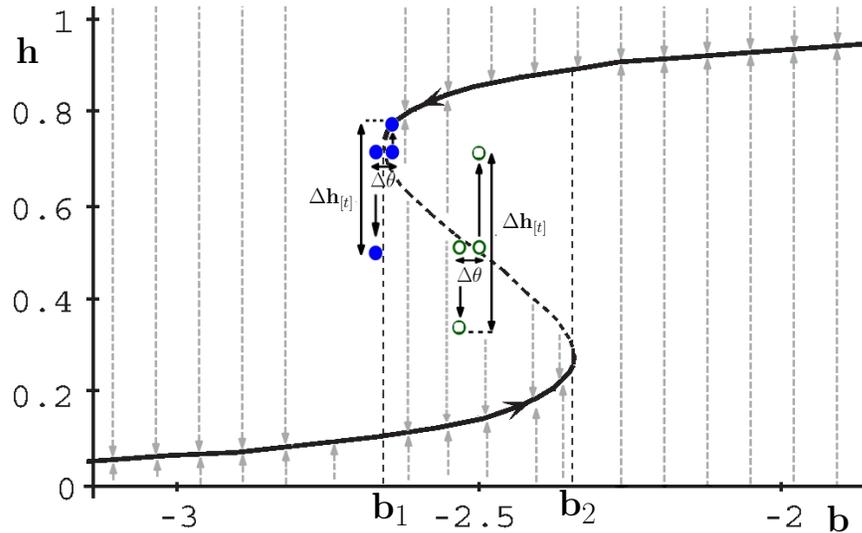


Figure 5.8: Bifurcation diagram of a single hidden unit RNN (with fixed recurrent weight of 5.0 and adjustable bias). See text for more details.

unfilled circles), the other is crossing a bifurcation boundary (filled circles). For large t , the $\Delta \mathbf{h}_{[t]}$ resulting from a change in b will be large even for very small changes in b (as the system is attracted towards different attractors) which leads to a large gradient.

Using these notions we can define a necessary condition for the gradients to explode. The condition is for a boundary of a basin of attraction to be crossed either by a change in $\mathbf{h}_{[0]}$ or a change in θ (usually one only considers the change in $\mathbf{h}_{[0]}$, but changes in θ can move this border such that, e.g., we fall into a different basin of attraction; we do not care if the border or the state moves). When crossing a bifurcation boundary that leads to large change in $\mathbf{h}_{[t]}$, by an abuse of language, we say the boundary of the basin of attraction of some attractor was also crossed implicitly (as for example if this attractor emerged, and the model landed in the new basin of attraction or it disappeared).

In [Doya \(1993\)](#) only bifurcation boundaries are considered, but we would argue this is a limited view. Crossing a bifurcation implies a global change, but locally things could stay the same (i.e., after the bifurcation we can find ourselves in the same basin of attraction). Also a change in θ means a change in the position of the boundary between basins of attractions which could lead to crossing such a bound-

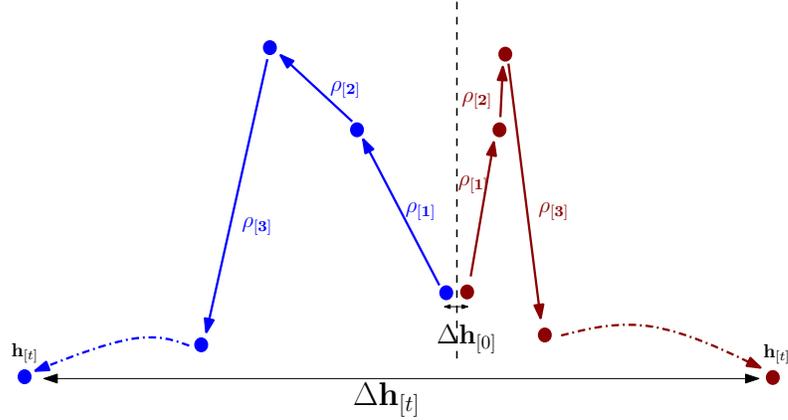


Figure 5.9: This diagram illustrates how the change in $\mathbf{h}_{[t]}$, $\Delta\mathbf{h}_{[t]}$, under the successive maps $\rho_{[t]}$ can be large for a small $\Delta\mathbf{h}_{[0]}$. The blue vs red (left vs right) trajectories are generated by the same maps $\rho_{[1]}, \rho_{[2]}, \dots$ for two different initial states.

ary, a scenario that is not considered by analyzing only bifurcations. Therefore crossing a bifurcation boundary is neither a *sufficient* nor a *necessary* condition.

One interesting observation from the dynamical systems perspective with respect to vanishing gradients is the following. If the factors $\frac{\partial\mathbf{h}_{[t]}}{\partial\mathbf{h}_{[k]}}$ go to zero (for $t - k$ large), it means that $\mathbf{h}_{[t]}$ does not depend on $\mathbf{h}_{[k]}$ (if we change $\mathbf{h}_{[k]}$ by some Δ , $\mathbf{h}_{[t]}$ stays the same). This translates into the model at $\mathbf{h}_{[t]}$ being close to convergence towards some attractor (which it would reach from anywhere in the neighbourhood of $\mathbf{h}_{[k]}$). Therefore avoiding the vanishing gradients means staying close to the boundaries between basins of attractions.

Input-driven dynamical systems

In order to be able to generalize the above mentioned observation to input driven models, one intuitive approach is to fold the input into the map. We, therefore, consider the maps $\rho_{[1]}, \dots, \rho_{[t]}$, where we apply a different map $\rho_{[i]}$ at each step i . Intuitively, we require the same behaviour as before, where (at least in some direction) the maps $\rho_{[1]}, \dots, \rho_{[t]}$ agree and change direction, for a small change in θ or $\mathbf{h}_{[0]}$ (even for the same input sequence). Figure 5.9 illustrates this behaviour.

For the specific parametrization provided by equation (5.7) we can take the analogy one step further by decomposing the maps $\rho_{[t]}$ into a fixed map $\hat{\rho}$ and

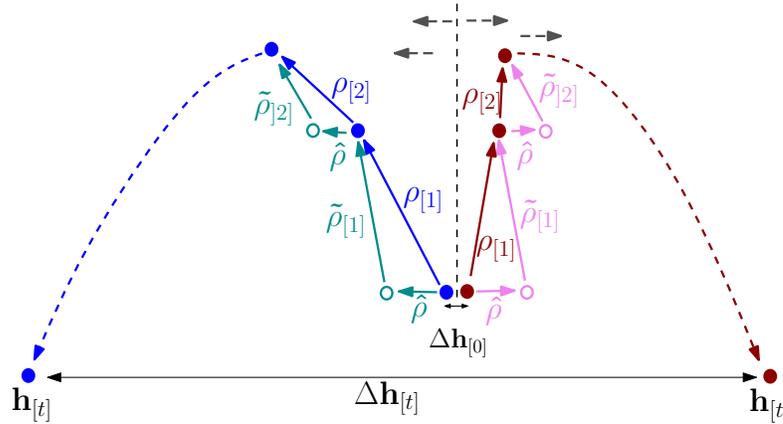


Figure 5.10: Illustrates how one can break apart the maps $f_{[1]}, .., f_{[t]}$ into a constant map $\hat{\rho}$ and the maps $\tilde{\rho}_{[1]}, .., \tilde{\rho}_{[t]}$. The dotted vertical line represents the boundary between basins of attraction, and the straight dashed arrow the direction of the map $\hat{\rho}$ on each side of the boundary. This diagram is an extension of Figure 5.9.

a time-varying one $\tilde{\rho}_{[t]}$. Then $\hat{\rho}(\mathbf{h}) = \mathbf{W}^{(rec)}\sigma(\mathbf{h})$ corresponds to an input-less recurrent network, while $\tilde{\rho}_{[t]}(\mathbf{h}) = \mathbf{h} + \mathbf{W}^{(in)}\mathbf{u}_{[t]}$ describes the effect of the input. This is depicted in in Figure 5.10. Since $\tilde{\rho}_{[t]}$ changes with time, it can not be analyzed using standard dynamical systems tools, but $\hat{\rho}$ can. This means that when a boundary of a basins of attraction is crossed for $\hat{\rho}$, the state might move towards a different attractor, which for large t can lead to a large discrepancy in $\mathbf{h}_{[t]}$. If $\tilde{\rho}_{[t]}$ is bounded, that it can interfere with this behaviour only when the state is close to the boundary, but not when it is far away. Therefore studying the asymptotic behaviour of $\hat{\rho}$ can provide some information about where such events are likely to happen.

5.6 Existing solutions for the vanishing and exploding gradients problem

Using an L1 or L2 penalty on the recurrent weights can help with exploding gradients. Assuming weights are initialized to small values, the largest singular value $\rho_{[1]}$ of $\mathbf{W}^{(rec)}$ is probably smaller than 1. The L1/L2 term can ensure that

during training $\rho_{[1]}$ stays smaller than 1, and in this regime gradients can not explode (see sec. 5.5.1). This approach limits the model to a regime where any information inserted in the model dies out exponentially fast (i.e. the model has the echo state property). This prevents the model from learning generator networks, nor can it exhibit long term memory traces.

Doya (1993) proposes to pre-program the model (to initialize the model in the right regime) or to use *teacher forcing*. The first proposal assumes that if the model exhibits from the beginning the same kind of asymptotic behaviour as the one required by the target, then there is no need to cross a bifurcation boundary. The downside is that one can not always know the required asymptotic behaviour, and, even if it is known, it might not be trivial to initialize the model accordingly. Also, such initialization does not prevent crossing the boundary between basins of attraction which as we have shown can also lead to the exploding gradients problem.

Teacher forcing refers to using targets for some or all hidden units. When computing the state at time t , we use the targets at $t-1$ as the value of all the hidden units in $\mathbf{h}_{[t-1]}$ which have a target defined. It has been shown that in practice this can reduce the chance that gradients explode, and even allow training generator models or models that work with unbounded amounts of memory (Pascanu and Jaeger, 2011; Doya and Yoshizawa, 1991). One important downside is that it requires a target to be defined at every time step.

Hochreiter (1991); Hochreiter and Schmidhuber (1997) propose the LSTM model to deal with the vanishing gradients problem. It relies on a special type of linear unit with a self connection of value 1. The flow of information into and out of the unit is guarded by learned input and output gates. There are several variations of this basic structure. This solution does not address explicitly the exploding gradients problem.

Martens and Sutskever (2011) use the Hessian-Free optimizer in conjunction with *structural damping*. This approach was argued to be able to address the vanishing gradients problem, though more detailed analysis is missing or a theoretical justification of this property. Presumably this method works because in high dimensional spaces there is a high probability for long term components to be orthogonal to short term ones. This would allow the Hessian to rescale these components independently. In practice, one can not guarantee that the components

are orthogonal, nor that the Hessian will scale them up. The method addresses the exploding gradients to some degree as well, as it takes curvature into account. Structural damping is an enhancement that forces the Jacobian matrices $\frac{\partial \mathbf{h}_t}{\partial \theta}$ to have small norm, hence further helping with the exploding gradients problem. The need for this extra term when solving the pathological problems might suggest that second order derivatives do not always grow at same rate as first order ones.

Echo State Networks (Jaeger, 2001) avoid the exploding and vanishing gradients problem by not learning $\mathbf{W}^{(rec)}$ and $\mathbf{W}^{(in)}$. They are sampled from hand crafted distributions. Because the spectral radius of $\mathbf{W}^{(rec)}$ is, by construction, smaller than 1, information fed in to the model typically dies out exponentially fast. An extension to the model is given by leaky integration units (Lukosevicius et al., 2007), where

$$\mathbf{h}_k = \alpha \mathbf{h}_{k-1} + (1 - \alpha) \sigma(\mathbf{W}^{(rec)} \mathbf{h}_{k-1} + \mathbf{W}_{in} \mathbf{u}_k + \mathbf{b}).$$

These units can be used to solve the standard benchmark proposed by Hochreiter and Schmidhuber (1997) for learning long term dependencies (Jaeger, 2013).

We would make a final note about the approach proposed by Tomas Mikolov in his PhD thesis (Mikolov, 2012) (and implicitly used in the state of the art results on language modelling (Mikolov et al., 2011)). It involves clipping the gradient’s temporal components element-wise (clipping an entry when it exceeds in absolute value a fixed threshold). This is in the same spirit of the proposal in the subsequent section, and a similar method had been independently used for LSTM (Graves, 2013).

5.7 Clipping the gradient norm

As suggested in Section 5.5.4, one mechanism to deal with the exploding gradients problem is to rescale their norm whenever it goes over a threshold:

As just mentioned, this algorithm is similar to the one proposed by Tomas Mikolov in his PhD thesis and used implicitly in his previous results for recurrent networks and to the one used by Alex Graves for LSTM networks (Graves, 2013). We only diverged from these original proposals in an attempt to provide a better

Algorithm 5 Pseudo-code for norm clipping

```
 $\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{L}}{\partial \theta}$   
if  $\|\hat{\mathbf{g}}\| \geq \textit{threshold}$  then  
     $\hat{\mathbf{g}} \leftarrow \frac{\textit{threshold}}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$   
end if
```

theoretical justification (see Section 5.5.4; we also move in a descent direction for the current mini-batch). From this perspective, we regard our work on this topic as an investigation of *why clipping works*, where previously it had been treated as a necessary implementational detail. In practice all variants of gradient clipping behave similarly. According to our geometrical interpretation of the exploding gradient, any small step that moves you back in the valley around the cliff like structure that causes the explosion is fine. The step can be even in a non-descent direction, because the several steps taken afterwards (before reaching the cliff structure again) will result in more progress towards a minimum.

Note that we make the assumption that one gets to do several steps in the normal curvature region near the cliff. That is, we assume that there is a wide valley near the cliff.

The proposed clipping is simple and computationally efficient, but it does however introduce an additional hyper-parameter, namely the threshold. One good heuristic for setting this threshold is to look at statistics on the average norm over a sufficiently large number of updates. In our experience values from half to ten times this average can still yield convergence, though convergence speed can be affected.

5.8 Preserving norm by regularization

We opt to address the vanishing gradients problem using a regularization term that represents a preference for parameter values such that back-propagated gradients neither increase or decrease in magnitude. Our intuition is that increasing the norm of $\frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}}$ means the error at time t is more sensitive to all inputs $\mathbf{u}_{[t]}, \dots, \mathbf{u}_{[k]}$ ($\frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}}$ is a factor in $\frac{\partial \mathcal{L}_{[t]}}{\partial \mathbf{u}_{[k]}}$). In practice some of these inputs will be irrelevant for the prediction at time t and will behave like noise that the network needs to learn to

ignore. The network can not learn to ignore these irrelevant inputs unless there is an error signal. These two issues can not be solved in parallel, and it seems natural to expect that we might need to force the network to increase $\left\| \frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}} \right\|$ at the expense of larger errors (caused by the irrelevant input entries) and *then* wait for it to learn to ignore these input entries. This suggests that moving towards increasing the norm of $\frac{\partial \mathbf{h}_{[t]}}{\partial \mathbf{h}_{[k]}}$ *can not* be always done while *following a descent* direction of the error \mathcal{L} (which is, e.g., what a second order method would do), and a more natural choice might be a regularization term.

The regularizer we propose prefers solutions for which the error preserves norm as it travels back in time:

$$\Omega = \sum_k \Omega_{[k]} = \sum_k \left(\frac{\left\| \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{[k+1]}} \frac{\partial \mathbf{h}_{[k+1]}}{\partial \mathbf{h}_{[k]}} \right\|}{\left\| \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{[k+1]}} \right\|} - 1 \right)^2 \quad (5.20)$$

In order to be computationally efficient, we only use the “immediate” partial derivative of Ω with respect to $\mathbf{W}^{(rec)}$ (we consider $\mathbf{h}_{[k]}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_{[k+1]}}$ as being constant with respect to $\mathbf{W}^{(rec)}$ when computing the derivative of $\Omega_{[k]}$), as depicted in eq. (5.21). This can be done efficiently because we get the values of $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_{[k]}}$ from BPTT. We use Theano to compute these gradients (Bergstra et al., 2010; Bastien et al., 2012).

$$\begin{aligned} \frac{\partial^+ \Omega}{\partial \mathbf{W}^{(rec)}} &= \sum_k \frac{\partial^+ \Omega_{[k]}}{\partial \mathbf{W}^{(rec)}} \\ &= \sum_k \frac{\partial^+ \left(\frac{\left\| \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{[k+1]}} \mathbf{W}^{(rec)T} \text{diag}(\sigma'(\mathbf{h}_{[k]})) \right\|^2}{\left\| \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{[k+1]}} \right\|^2} - 1 \right)^2}{\partial \mathbf{W}^{(rec)}} \end{aligned} \quad (5.21)$$

Note that our regularization term only forces the Jacobian matrices $\frac{\partial \mathbf{h}_{[k+1]}}{\partial \mathbf{h}_{[k]}}$ to preserve norm in the relevant direction of the error $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_{[k+1]}}$, not for all directions (we do not enforce that all eigenvalues are close to 1).

There are slight differences between this approach and the one introduced in Saxe et al. (2014), where orthonormal matrices are used for *very* deep linear MLPs. While there is generical belief that orthonormal matrices can also be used to address (at least to some degree) the vanishing gradients problem for recurrent networks, this intuition might be misleading.

There is a crucial difference between the two situations, feedforward models and

recurrent ones, and this difference is caused by the *memory* of RNNs. The amount of memory that RNNs can exhibit is finite, and, because there exists an input at each time step, the hidden state that stores this information acts as a bottleneck. In order to perform well, RNNs need to be able to *forget* information which is not relevant for the task, and this can only be done if there are directions of small eigenvalues in the recurrent weight matrix. A model that preserves information in all directions will have its memory corrupted by noise or irrelevant inputs, leading to worse performance. As pointed above, however, the network first needs to get into this regime of remembering everything, before it is able to learn what information it can forget. We believe that enforcing to preserve norm in the direction of the error is the least intrusive approach.

In [Ganguli and Sompolinsky \(2010\)](#), memory is analysed for linear recurrent models where the input is assumed to be sparse. This work shows that orthogonal matrices can be used to perform *compressed sensing* and exhibit long memory traces when the input is sparse.

In general the input of a model is not sparse, or at least the sparsity of the relevant signal is hidden behind the noise in the signal. Another observation is that nonlinear dynamics are sometimes surprisingly different from those of linear systems. In particular the memory analyzed in [Ganguli and Sompolinsky \(2010\)](#) is based on transient dynamics of stable systems (that have a single point attractor at the origin). In this chapter we argued that nonlinear dynamical system can provide more interesting behaviour, attractor-like phenomena that can lead to unbounded in time traces of memory, which transient dynamics can not. Examples are the input-induced attractors that we witnessed for the ESN model with WM-units. It is unclear that an orthogonality constraint would be useful for such dynamics. Also, the ongoing processing that the network has to do in parallel with storing information might not be realizable unless the model has directions in which it can contract (eigenvalues that are smaller than unity). The intuition behind it is that contraction can be used to *forget* or *remove* certain characteristics of the signal. These operations are needed for processing the input. Finally, it is not clear that for non-linear dynamics, non-orthonormal matrices are not useful to perform some form of compressed sensing.

We believe these intuitions to suggest that restricting the nonlinear recurrent model to an orthogonal recurrent weight matrix might severely reduce the types of

behaviours it can exhibit. Further work is needed, however, in analysing such approaches and to understand how these intuitions from linear models, such as those from [Ganguli and Sompolinsky \(2010\)](#), can be transferred to nonlinear recurrent networks. A possible first step to pursue is, for example, to show whether orthogonal weight matrices can still lead to recurrent models that can mimic a Turing machine.

The second observation is that we are using a soft constraint, which means that by chance, the behaviour of the network can switch into a mode where either the gradients vanish or they explode. A model employing this regularization term should be well equipped to deal with the exploding gradients problem, for example, by using the gradient norm clipping strategy discussed previously. From the dynamical systems perspective we can see that preventing the vanishing gradient problem implies that we are pushing the model towards the boundary of the current basin of attraction (such that during the N steps it does not have time to converge). Crossing the boundary of this attractor will cause the gradients to explode.

5.9 Learning long term correlations

5.9.1 Pathological synthetic problems

As done in [Martens and Sutskever \(2011\)](#), we address some of the pathological problems proposed by [Hochreiter and Schmidhuber \(1997\)](#) that require learning long term correlations.

Overview: The Temporal Order problem

We consider the temporal order problem as the prototypical pathological problem, extending our results to the other proposed tasks afterwards. The input is a long stream of discrete symbols not including A or B . At two points in time (in the beginning and middle of the sequence) a symbol within $\{A, B\}$ is emitted. The task consists in classifying the order (either AA, AB, BA, BB) at the end of the sequence.

Figure 5.11 shows the success rate of standard mini-batch stochastic gradient descent MSGD, MSGD-C (MSGD enhanced with our clipping strategy) and MSGD-CR (MSGD with the clipping strategy and the regularization term)¹. It was previously shown in Sutskever et al. (2013) that initialization of the model plays an important role for training RNNs. We consider three different initializations. *sigmoid* is the most adversarial initialization, where we use a sigmoid unit network where $\mathbf{W}^{(rec)}, \mathbf{W}^{(in)}, \mathbf{W}^{(out)} \sim \mathcal{N}(0, 0.01)$. *basic tanh* uses a tanh unit network where $\mathbf{W}^{(rec)}, \mathbf{W}^{(in)}, \mathbf{W}^{(out)} \sim \mathcal{N}(0, 0.1)$. *smart tanh* also uses tanh units and $\mathbf{W}^{(in)}, \mathbf{W}^{(out)} \sim \mathcal{N}(0, 0.01)$. $\mathbf{W}^{(rec)}$ is sparse (each unit has only 15 non-zero incoming connections) with the spectral radius fixed to 0.95. In all cases $\mathbf{b} = \mathbf{b}^{(out)} = 0$. The graph shows the success rate over 5 different runs (with different random seeds) for a 50 hidden unit model, where the x -axis contains the length of the sequence. We use a constant learning rate of 0.01 with no momentum. When clipping the gradients, we used a threshold of 1., and the regularization weight was fixed to 4. A run is successful if the number of misclassified sequences was under 1% out of 10000 freshly random generated sequences. We allowed a maximum number of 5 million updates, and use mini-batches of 20 examples.

This task provides empirical evidence that exploding gradients are linked with tasks that require long memory traces. As the length of the sequence increases, using clipping becomes more important to achieve a better success rate. More memory implies larger spectral radius, which leads to rich regimes where gradients are likely to explode. MSGD-CR solves the task with a success rate of 100% for sequences up to 250 steps (the maximal length used in Martens and Sutskever (2011) was 200).

Furthermore, we can train a single model to deal with any sequence of length 50 up to 200 (by providing sequences of different random lengths in this interval for different MSGD steps). We achieve a success rate of 100% over 5 seeds in this regime as well (all runs had 0 misclassified sequences in a set of 10000 randomly generated sequences of different lengths). We used an RNN of 50 tanh units initialized in the *basic tanh* regime. **The same trained model can address sequences of length up to 5000 steps, lengths never seen during training.** Specifically the same model produced 0 misclassified sequences (out of 10000 sequences of same

1. When using just the regularization term, with out clipping, learning usually fails due to exploding gradients.

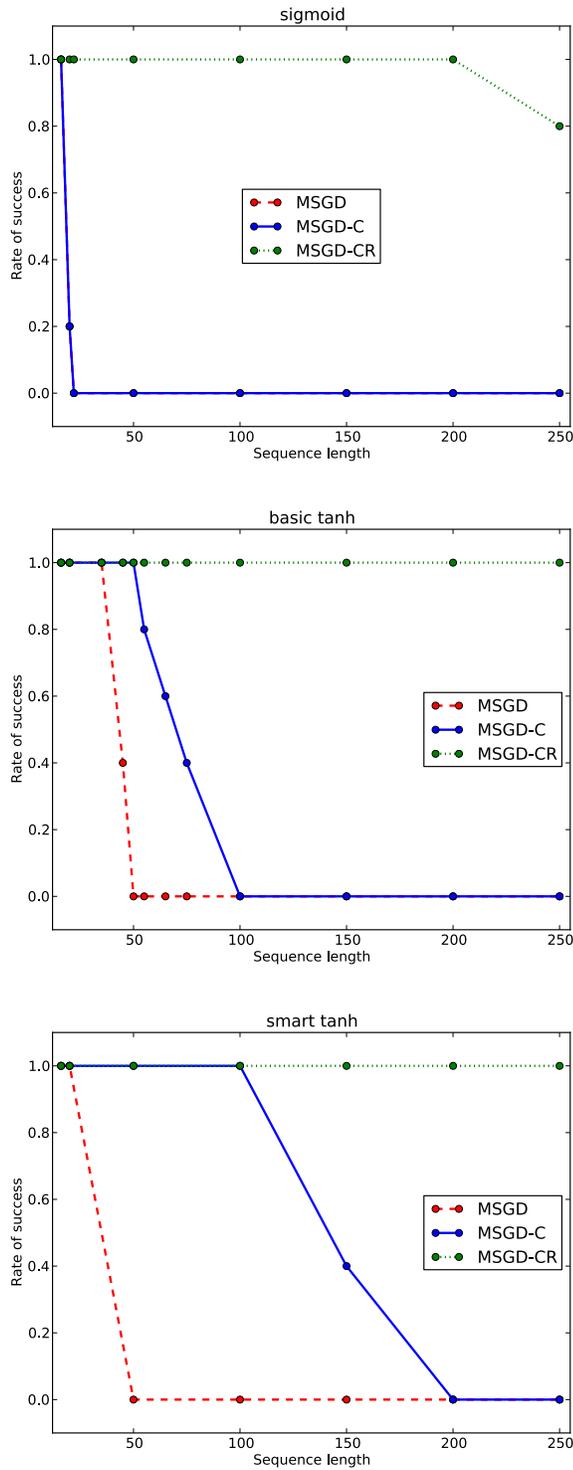


Figure 5.11: Rate of success for solving the temporal order problem versus sequence length for different initializations. See text.

length) for lengths of either 50, 100, 150, 200, 250, 500, 1000, 2000, 3000 and 5000.

This provides evidence that the model exhibits attractor-like phenomena and relies on it to achieve this possibly unbounded stable memory, similar to the ESN with WM-units discussed in Section 5.4. We take this as anecdotal evidence that one can *learn to exhibit stable memory with out relying on any kind of hints*.

This contrasts the believe that recurrent networks mostly rely on transient dynamics to gain memory (as for example ESN networks do in Jaeger (2013)) which do not permit generalization to longer sequences.

Overview: Other pathological tasks

SGD-CR was also able to solve (100% success on the lengths listed below, for all but one task) other pathological problems proposed in Hochreiter and Schmidhuber (1997), namely the *addition* problem, the *multiplication* problem, the *3-bit temporal order problem*, the *random permutation* problem and the *noiseless memorization* problem in two variants (when the pattern needed to be memorized is 5 bits in length and when it contains over 20 bits of information; see Martens and Sutskever (2011)). For every task we used 5 different runs (with different random seeds). For the first 4 problems we used a single model for lengths up to 200, while for the *noiseless memorization* we used a different model for each sequence length (50, 100, 150 and 200). The hardest problems for which only two runs out of 5 succeeded was the random permutation problem. For the addition and multiplication task we observe successful generalization to sequences up to 500 steps (we notice an increase in error with sequence length, though it stays below 1%). Note that for the addition and multiplication problem a sequence is misclassified with the square error is larger than .04. In most cases, these results outperforms Martens and Sutskever (2011) in terms of success rate, they deal with longer sequences than in Hochreiter and Schmidhuber (1997) and compared to Jaeger (2013) they generalize to longer sequences.

5.9.2 Overview:Language modelling and polyphonic music prediction

We address the task of polyphonic music prediction, using the datasets Piano-midi.de, Nottingham and MuseData described in Boulanger-Lewandowski et al.

DATA SET	DATA FOLD	MSGD	MSGD+C	MSGD+CR	SOTA FOR RNN	SOTA
PIANO-MIDI.DE (NLL)	TRAIN	6.87	6.81	7.01	7.04	6.32
	TEST	7.56	7.53	7.46	7.57	7.05
NOTTINGHAM (NLL)	TRAIN	3.67	3.21	2.95	N/A	1.81
	TEST	3.80	3.48	3.36	3.23	2.31
MUSEDATA (NLL)	TRAIN	8.25	6.54	6.43	6.47	5.20
	TEST	7.11	7.00	6.97	6.99	5.60
PENN TREE-BANK 1 STEP (BITS/CHAR)	TRAIN	1.46	1.34	1.36	N/A	N/A
	TEST	1.50	1.42	1.41	1.41	1.24
PENN TREE-BANK 5 STEPS (BITS/CHAR)	TRAIN	N/A	3.76	3.70	N/A	N/A
	TEST	N/A	3.89	3.74	N/A	N/A

Table 5.4: Results on polyphonic music prediction in negative log likelihood per time step and natural language task in bits per character. Lower is better. Bold face shows state of the art for RNN models. Note that SOTA stands for state of the art.

(2012) and language modelling at the character level on the Penn Treebank dataset (Marcus et al., 1993). We also explore a modified version of the task, where we require to predict the 5th character in the future (instead of the next). We assume that for solving this modified task long term correlations are more important than short term ones, and hence our regularization term should be more helpful.

The training and test scores for all natural problems are reported in Table 5.4 as well as state of the art for these tasks. We note that keeping the regularization weight fixed (as for the previous tasks) seems to harm learning. One needs to use a $1/t$ decreasing schedule for this term. We hypothesis that minimizing the regularization term affects the ability of the model to learn short term correlations which are important for these tasks, though a more careful investigation is lacking.

These results suggest that clipping solves an optimization issue and does not act as a regularizer, as both the training and test error improve in general. Results on Penn Treebank achieved the same error as the ones in Mikolov et al. (2012), where a different gradient clipping strategy was used, thus providing evidence that both behave similarly. An LSTM model Graves (2013) with weight noise regularization

and dynamic evaluation has the state of the art on this task. This model also employs a gradient clipping strategy.

Similar observation can be done for the polyphonic music prediction task. Clipping helps the optimization, resulting in lower training and testing error, while adding the regularization term that prevents gradients to vanish helps to get only better test error.

We need to make two remarks for the music results. The first one is that state of the art for this task is held by an RNN-NADE model trained with Hessian-Free Optimization. The structured output layer of this model might be the key of its better performance. The second observation is that these results are taken from our published paper [Pascanu, Mikolov, and Bengio \(2013\)](#). For the Nottingham dataset, we explored, in our more recent paper [Pascanu, Gulcehre, Cho, and Bengio \(2014\)](#), a more fine grid search for MSGD-C, resulting in a lower test and validation error. However, to be fair with the results for MSGD-CR, we did not present this result in table 5.4, rather the original one. The new result (together with others) is introduced in detail in the next chapter of this thesis.

The state of the art results for RNN presented in table 5.4 are best results for standard recurrent networks reported in the literature for the different datasets. For Nottingham the result is taken from [Pascanu, Gulcehre, Cho, and Bengio \(2014\)](#), while for MuseData and Piano-midi the result is from [Bengio, Boulanger-Lewandowski, and Pascanu \(2013\)](#). In [Pascanu, Gulcehre, Cho, and Bengio \(2014\)](#) we also explored deep recurrent networks for these task which performed better. We did not considered these numbers for the state of the art results for RNN column.

5.9.3 Details of the experimental setup and results

For both training and testing sets of the pathological tasks, we always generate data on the fly, i.e. every time we compute the test error we generate a new set of 10000 sequences (the same when we need to do a MSGD step). Code for running some of the experiments is provided at https://github.com/pascanur/training_RNNs.

Temporal order

For the temporal order task the length of the sequence is given by T . We have a fixed set of two symbols $\{A, B\}$ and 4 distractor symbols $\{c, d, e, f\}$. The sequence

entries are uniformly sampled from the distractor symbols everywhere except at two random positions, the first position sampled from $[\frac{T}{10}, \frac{2T}{10}]$, while the second from $[\frac{5T}{10}, \frac{6T}{10}]$. The task is to predict the order in which the non-distractor symbols were provided, i.e. either $\{AA, AB, BA, BB\}$.

We used a grid search, where we tried the following values:

- number of hidden units N in 50,100
- learning rate lr in .01,.001
- threshold for gradient clipping μ in 6., 1., 4.
- regularization weight α in 4., 2., 1., .5
- maximal number of updates 100k, 5G

We did not explore any decreasing scheme for the learning rate or for the regularization weight. Momentum was also not used. The best hyper-parameters were selected based on a few initial jobs with sequences of length 100 (i.e. we did not explore every combination for all possible lengths and all seeds).

The final hyper-parameters used were:

- $N = 50$ (number hidden units)
- $lr = 0.01$ (learning rate)
- $\mu = 1.$ (threshold for gradient clipping)
- $\alpha = 4.$ (regularization term)
- maximal number of updates 5G

Note that other combinations also lead to convergence (e.g. μ of 6 and smaller learning rate of .001), though we find this combination to converge faster, and work better in the regime of sequences of length 250 steps.

For generating the subplots of Figure 5.11, we explored the following lengths T (i.e. we trained 5 models for each of this value, the 5 models using the same random seeds):

- for *sigmoid* initialization, considered T values are: 16, 20, 22, 50, 100, 150, 200, 250
- for *basic tanh* initialization, considered T values are : 16, 20, 35, 45, 50, 55, 65, 75, 100, 150, 200, 250
- for *smart tanh* initialization, considered T values are : 16, 20, 50, 100, 150, 200, 250

The number of updates required for convergence greatly depends on the length of the sequence. Using both clipping and the regularization term, on average, for

$T = 200$, the *sigmoid* initialization takes 685k updates, the *basic tanh* takes 476k updates, and *smart tanh* 434k updates. In comparison, if we fix $T = 20$ and the initialization to *smart tanh*, we have for MSGD 21k updates, for MSGD-C 11k updates and for MSGD-CR 2k updates.

When training a single model for random lengths from 50 to 200 steps, we decreased the regularization weight α to 2. All values of μ seem to work, though convergence seems faster with $\mu = 1$. All 20 sequences in a training mini-batch had the same length T uniformly sampled from $[50, 200]$. The test set was split into 500 mini-batches, each of a different length uniformly sampled from $[50, 100]$. Each test evaluation implied generating a new set of such 500 mini-batches. For all tasks where we considered varying length sequences, we stopped learning and considered the run successful when no sequence was misclassified out of 10000 sequences of varying length. We then evaluate the model on 10000 sequences of length 50, 10000 sequences of length 100, 10000 sequences of length 150 and 10000 sequences of length 200. Additionally we explored T in $\{250, 500, 1000, 2000, 3000, 5000\}$. For the temporal order problem in all cases 0 sequences were misclassified in each run.

Addition problem

For this task, the input consists of a sequence of random numbers, where two random positions (one in the beginning and one in the middle of the sequence) are marked. The model needs to predict the sum of the two random numbers after the entire sequence was seen. For each generated sequence we sample its actual length T' from $[T, \frac{11}{10}T]$, though for clarity we refer to T as the length of the sequence. The position of the first marked number is sampled from $[1, \frac{T'}{10}]$, while the second position is sampled from $[\frac{4T'}{10}, \frac{5T'}{10}]$. These positions i, j are marked in a different input channel that is 0 everywhere except for the two sampled positions when it is 1. The model needs to predict the sum of the random numbers found at the sampled positions i, j divided by 2.

To address this problem we use a 50 hidden units model, using the *basic tanh* initialization. We explored the learning rates $\{.01, .001\}$ and $\alpha \in \{.5, 1., 2.\}$. We have tried $\mu \in \{6., 1.\}$. The final choice of hyper-parameters are $lr = .01$, $\alpha = .5$. We did not notice any significant difference in picking $\mu = 6$ or $\mu = 1$, though clipping was required (i.e. $\mu = \infty$ fails). $\mu = 1$. seems to result in slightly faster convergence (on average 1.226G versus 1.299G updates).

T	50	100	150	200	250	300	350	400	600
error	.08%	.01%	.01%	0%	0%	.02%	.01%	.04%	.52%

Table 5.5: Addition task. Error as percentage of misclassified sequences out of 10000 for different values of T for the same trained model.

T	50	100	150	200	250	300	350	400	600
error	.55%	.04%	.04%	.04%	.08%	.26%	.28%	.73%	2.11%

Table 5.6: Multiplication task. Error as percentage of misclassified sequences out of 10000 for different values of T for the same trained model.

We directly explored training the model on sequences of varying length T between 50 and 200, following the same procedure as for the temporal order task.

A single model manages to handle (within the permitted 1% error) lengths of 50,100, 150, 200, 250, 300, 400 and even 600. In Table5.5 you can see how the misclassification error behaves for different lengths for a trained model (one out of the 5 random seeds used). The trained models for the other different random seeds behave similarly.

Multiplication problem

This task is similar to the addition problem, just that the predicted value is the product of the random numbers instead of the sum. We used the same hyperparameters as for the addition problem (with out validating them through a grid search), and obtained very similar results (Table5.6 shows the results for one of the 5 random seeds used; for the other seeds we see a similar behaviour). We only explored the value of μ which seemed to reduce more the number of updates required for convergence (1.728G updates for $\mu = 1$ versus 3.418G updates for $\mu = 6$).

3-bit temporal order problem

This task is similar to the temporal order problem, except that we have 3 random positions, first one sampled from $[\frac{T}{10}, \frac{2T}{10}]$, second one from $[\frac{3T}{10}, \frac{4T}{10}]$ and last one from $[\frac{6T}{10}, \frac{7T}{10}]$.

As before, we explored directly training the model on sequences of varying length. We explored the number of hidden units in $\{50, 100\}$, learning rate in

T	50	100	150	200	250	300	350	400	600
error	.0%	.01%	.0%	.06%	.01%	0%	2.75%	14.13%	24.81%

Table 5.7: 3-bit temporal order task. Error as percentage of misclassified sequences out of 10000 for different values of T for the same trained model.

T	50	100	150	200	250	300	350	400	600
error	.04%	.01%	.02%	.03%	.02%	0%	.03%	.02%	.04%

Table 5.8: Random permutation task. Error as percentage of misclassified sequences out of 10000 for different values of T for the same trained model.

$\{.01, .001\}$ and $\mu \in \{1., 6.\}$. We explored $\alpha \in \{2., 1.\}$. Best combination (with regard to convergence speed) was $lr = .01$, $\mu = 1.$, $\alpha = 2.$ We chose 100 hidden units. Average number of updates to train was 569k. Table 5.7 describes a single trained model, the other 4 trained model behave similarly.

Random permutation problem

For this problem we have a dictionary of 100 symbols. Except the first and last position which have the same value sampled from $\{1, 2\}$ the other entries are randomly picked from $[3, 100]$. The task is to do next symbol prediction, though the only predictable symbol is the last one (based on the first symbol seen).

We explored 50 and 100 hidden units, with a learning rate of either .01 and .001. $\alpha \in \{2., 1., .5\}$ and $\mu \in \{6., 1.\}$. We run experiments only on varying length sequences. The task proved harder to train. 2 out of 5 runs were successful for the final chosen hyper-parameters: 100 hidden units, with a learning rate of .001 and $\alpha = 1.$ and $\mu = 6.$ The error for different lengths are listed in Table 5.8 for one of the two successful run.

Noiseless memorization problem

For the noiseless memorization we are presented with a binary pattern of length 5, followed by T steps of constant value. After these T steps the model needs to generate the pattern seen initially. We also consider the extension of this problem from [Martens and Sutskever \(2011\)](#), where the pattern has length 10, and the symbol set has cardinality 5 instead of 2.

Task	50 steps	100 steps	150 steps	200 steps
Original task	0%	0%	0%	0%
Extended task	0.844%	0.724 %	0.75%	0.77%

Table 5.9: Noiseless memorization problem. Average (over the 5 seeds) percentage of misclassified sequences out of 10000 for the different lengths considered and the two variations of the task

All runs on varying length sequences failed, so we trained a different model for the considered lengths (50, 100, 150, 200).

We explored $\alpha \in \{1, 2\}$, learning rate in $\{.01, .001\}$, number of hidden units in $\{50, 100\}$ and $\mu \in \{1., 6.\}$. We explored hyper-parameters on the original task (where the symbol set had cardinality 2). We chose to use $\alpha = 1$, learning rate 0.01, 100 hidden units and $\mu = 1$ for faster convergence.

We manage a 100% success rate (i.e. all 5 runs had under 1% misclassified sequences out of 10000) on these tasks for 5 different random seeds, though we train 4 models for each of the 4 possible lengths (50, 100, 150, 200).

Polyphonic music prediction

We train our model, a sigmoid units RNN, on sequences of 200 steps. The cut-off coefficient threshold is the same in all cases, namely 8^1 .

In case of the Piano-midi.de dataset we use 300 hidden units and an initial learning rate of 1.0. For all natural tasks we halved the learning rate every time the error over an epoch increased instead of decreasing. For the regularized model we used a initial value for regularization coefficient α of 0.5, where α follows a $\frac{1}{2t}$ schedule, i.e. $\alpha_t = \frac{1}{2t}$ (where t measures the number of epochs). We found that keeping alpha constant as before results in worse performance for $\alpha \in \{0.5, 1., 2.\}$. Further more we speculate that for tasks where short term information is important one needs to use a decreasing schedule for the regularization weight.

For the Nottingham dataset we used 400 hidden units, with an initial $\alpha = 5$. that started decreasing with $\frac{1}{\max(1, t-10)}$ after the first 10 epochs. μ was set to 8. The learning rate was set to 1. 100 of the 400 hidden units where leaky integration

1. Note this value changes considerably if one takes the sum over the sequence length versus the mean. Of course, to make taking the mean a consistent choice, all sequences used have to be of the same length. We used the mean, or, stated differently, we divided the cost by 200

units (Bengio, Boulanger-Lewandowski, and Pascanu, 2013) with their leaky factor randomly sampled from $[0.02, 0.2]$.

For MuseData we used 400 units. The learning rate was also decreased to 0.5. For the regularized model, the initial value for α was 0.1, and $\alpha_t = \frac{1}{2t}$. 100 of the 400 hidden units were leaky integration units, as used in Bengio, Boulanger-Lewandowski, and Pascanu (2013). Their leaky-integration factor was randomly sampled in $[0.02, 0.2]$.

We have explored number of hidden units in the range $\{200, 300, 400\}$, learning rates of $\{2, 1, .5\}$, $\mu \in \{4, 8, 12\}$ and $\alpha \in \{1, 5, .5\}$. We tried keeping α constant, or start decreasing from epoch 0, epoch 10 or epoch 20. We use either $\frac{1}{t}$ or $\frac{1}{2t}$ schedule. Not all hyper-parameters were explored on all tasks. Most of the hyper-parameter selection was done on the Piano-midi.de task.

Language modelling

For the language modelling task we used a 500 sigmoidal hidden units model with no biases (Mikolov et al., 2012). We use the normal distribution $\mathcal{N}(0, 0.1)$ to sample the weights. The model is trained over sequences of 200 steps, where the hidden state is carried over from one step to the next one.

We use a cut-off threshold of 45 (though we do not rescale the cost based on the sequence length for each step) in this case. For next character prediction we have a learning rate of 0.01 when using clipping with no regularization term, 0.05 when we add the regularization term and 0.001 when we do not use clipping. When predicting the 5th character in the future we use a learning rate of 0.05 with the regularization term and 0.1 without it.

Interestingly enough, the strategy for the regularization factor α when solving the next character prediction that performed best was to keep it constant to the value .01. For the modified task a $\frac{1}{t}$ schedule again seems to perform better, and we used an initial value for α of 0.05 with $\alpha_t = \alpha \frac{1}{1 + \frac{t}{2800}}$, where t is the update index.

We have explored constant $\alpha \in \{6, 1, 0.5, 0.01, 0.001\}$ and $\frac{1}{1 + \frac{\max(0, t-t_0)}{\beta}}$ schedule with $\beta \in \{2800, 500\}$ and t_0 in $\{0, 28108\}$. The number of hidden units was kept constant to 500.

5.10 Learning to memorize: discussion and summary

In this chapter we looked at recurrent neural networks, a specific family of neural models. Compared to their feedforward counterpart, recurrent networks exhibit memory, which allows them to be as powerful as a Turing machine (Siegelmann and Sontag, 1991). However, it is unclear if we can learn such rich behaviour, and, even more fundamentally, what kind of intrinsic dynamical mechanisms could be used to achieve it.

In the introduction of this chapter, we provide a brief introduction to working memory. We discussed what properties are “desirable” for memory, and why obtaining this behaviour within a recurrent network can be advantageous.

In this chapter we presented two main results. The first one shows that a recurrent network can exhibit working memory in a practical manner. This is done using a modified ESN model, where we need to rely on hints and “teacher-forcing” to learn this behaviour. Further analysis suggests that this memory is not obtained via transient dynamics, but rather via attractor like phenomena.

We call this phenomena an input-induced attractor, and we believe it is, in nature, similar to the previously reported high-dimensional attractors from Maass et al. (2007). This attractors seems to be resilient to most input patterns except specific patterns that are learnt, which lead to the model being “kicked” out of the attracting state. We believe that this behaviour is not due to neural noise, but we are actually dealing with a behaviour specific to input driven dynamical systems, behaviour that still requires more rigorous investigation.

While this result is very encouraging, the use of hints during training reduces its applicability. For most task of interest we do not know before hand what information and when it needs to be stored (or deleted). If we want the model to discover memory by simply learning from data, the result above suggests that we need to be able to learn in rich dynamical regimes that can show many different attractors and be riddled with bifurcations.

This observation guided our second main result presented in this chapter. If we need to move through these rich regimes, it means that learning needs to be able to address at least two issues specific to recurrent neural networks: the exploding and vanishing gradients problem (Hochreiter, 1991; Bengio et al., 1994).

We provide an intuitive geometrical explanation for one strategy to deal with the exploding gradients problem, namely clipping the norm of the gradients. We argue that such issues arise from cliff-like structures in the error surface, and by clipping the norm of the gradient, we force the model to stay in the valley surrounding this cliff. The valley is wide and well behaved, so, by this strategy, we get to take several steps towards some minimum before we reach the cliff-like structure again, once we’ve clipped the gradients.

For the vanishing gradients problem, we propose a soft constraint that enforces gradients not to loose norm as they “travel” back in time. Our choice is motivated by the intuition that preserving gradient norm might lead to moving in a non-descent direction. Memory is a finite quantity whose size is determined by the hidden state size. Given that we have an input at each time step, the hidden layer behaves like a bottleneck. For a potentially infinite input sequence, there will not be enough memory to remember the entire input sequence.

A recurrent model needs to learn to memorize only the important bits, and ignore information that is not relevant for the task (or sacrifice some of the information in the input). We believe that doing so can only happen if there is an error signal from this irrelevant inputs, therefore the model need first to try to remember everything (at the expense of larger errors) and then learn to forget irrelevant inputs. A regularization term can address this through the regularization weight which decides how important is for the constraint to be respected versus reducing the loss of the task.

Using both these solutions, gradient norm clipping and the regularization term, we obtain state of the art results on the pathological problems proposed by [Hochreiter and Schmidhuber \(1997\)](#). Specifically, for the temporal order task, we are able to learn to solve the task with out any hints, and generalize to sequences that are an order of magnitude longer than the ones seen during training. This suggests that the trained model exhibits some sort of stable memory, probably similar to the ESN network trained with hints.

We regard this observation as anecdotal evidence that such unbounded memory behaviour can be learnt by local optimization techniques, such as stochastic gradient descent, provided that the task is properly set up (for example via this regularization term and the clipping strategy employed). This results, together with the theoretical superiority of recurrent models versus feedforward ones, highlights

the importance of recurrent network research.

Moving towards artificial intelligence, we believe that the different neural components of this system will have to be equipped with working memory, and hence recurrent connections. We argued in Chapter 3 that depth is a crucial components to neural networks because it induces an exponential efficiency in representations. Recurrent connections are also such a crucial component, because they allows the network to exhibit memory and move beyond simple mappings from input to output. Context, at each time scale, can be crucial to better disambiguate the proper next action of the agent.

Regarding the experiments carried out in this section, we regard them as just an initial step. In particular, there are many intriguing extensions that need to be explored.

For the ESN model that relies on hints, one important step forward is to show its practicality on real world tasks. For language, or speech, additional temporal information can be constructed that could be used as hints. For example, a character level model could learn to keep track of which word it is in (or word class based on some clustering), or which phrase is currently parsing (when such information is available). For speech we could keep track of who is the current speaker (when we have multiple speakers taking turns).

One other possible direction is to use the same strategy (the WM units trained with teacher forcing) for normal recurrent networks to bootstrap learning. This could be done, for example, when hints are available for only a small subset of the training set.

The regularization term used for addressing the vanishing gradients problem seemed to underperform on natural task that require both short term and long term information. We believe that the constraint might force the model to ignore short term information. Exploring possible solutions for this problem is another future direction. One approach could be to simply split the model into two components, one that is meant to address short term information, the other long term information. Other direction is to explore more complex schedules for the regularization term that allows the model ample time to focus on both the short term information and long term one. Yet another possibility is to alter the structure of the RNN to one that is more efficient at encoding complex behaviours. More complex optimizers can resolve saddle structures in the error surface, and allow for

better optimization of the model.

Finally, a lot more work is needed to understand the mechanisms behind memory for recurrent networks. Such analysis can be very insightful in understanding the difficulties we have with learning to exhibit memory and can suggest structural changes to RNNs.

6

Depth for recurrent models

The previous chapter looked at some particular optimization issues surrounding recurrent models. We know that the ability of recurrent networks to exhibit complex behaviour patterns is strongly connected to its ability to exploit context.

We looked at how an RNN can use long term context and specifically we discussed attractor-like phenomena that could be used to construct working memory which would allow exploiting this long term context.

While learning is a crucial component, in this chapter we return to a different fundamental question, namely that of efficiency. In Chapter 3 we connected the depth of a feedforward MLP with how efficiently (for a fixed number of parameters) this model can represent certain families of functions.

Given the rich behaviours that we expect from recurrent networks, it is natural to ask whether some structural change of the model could make it more efficient at encoding complex behaviour. Following the intuitions in Chapter 3 we explore here the concept of a deep recurrent network empirically, providing intuitive justifications for the different proposed structural changes.

The content of this chapter is based on the paper [Pascanu, Gulcehre, Cho, and Bengio \(2014\)](#) that I published with my co-authors at the International Conference of Learning Representations (ICLR). In writing the chapter I *borrowed* figures and paragraphs from this paper. Please see Section 1.1 for a detailed description of my personal contribution to the paper.

6.1 Motivation

Deep learning is built around a hypothesis that a deep, hierarchical model can be exponentially more efficient at representing some functions than a shallow

one (Bengio, 2009). A number of recent theoretical results support this hypothesis (see, e.g., Le Roux and Bengio, 2010; Delalleau and Bengio, 2011; Montúfar and Ay, 2011; Pascanu et al., 2014). Furthermore, there is a wealth of empirical evidences supporting this hypothesis (see, e.g., Goodfellow et al., 2013; Krizhevsky et al., 2012; Hinton et al., 2012). See also Chapter 3 for a detailed treatment of this question. These findings make us suspect that the same argument should apply to recurrent neural networks.

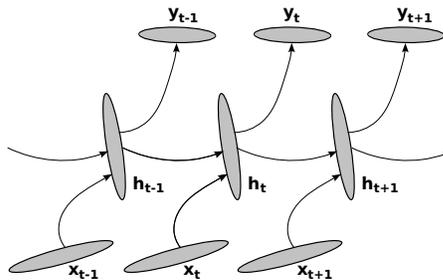


Figure 6.1: A conventional recurrent neural network unfolded in time.

The *depth* is defined in the case of feedforward neural networks as having multiple nonlinear layers between input and output. Unfortunately this definition does not apply trivially to a recurrent neural network because of its temporal structure. For instance, any RNN when unfolded in time as in Figure 6.1 is deep, because a computational path between the input at time $k < t$ to the output at time t crosses several nonlinear layers. However, if we considered the different computations carried out by a recurrent model, one can identify ways in which the model is “shallow”. For example the computations carried out to obtain the new hidden state, provided the previous one and the input is given by a shallow model. We argue that *depth* is an ambiguous term for RNNs, and one needs to specify in *which sense the model is deep*.

For example, Schmidhuber (1992); El Hahi and Bengio (1996) earlier proposed one way of building a deep RNN by stacking multiple recurrent hidden states on top of each other. This approach potentially allows the hidden state at each level to operate at different timescale (see, e.g., Hermans and Schrauwen, 2013). This shows that the model can be *deep* in a way that is orthogonal to being deep in time (i.e. when unfolding the graph in time).

To understand what a deep recurrent network means, we first introduced a new interpretation of a recurrent network. Based on this interpretation we can define segments of the model that *can be made deep*.

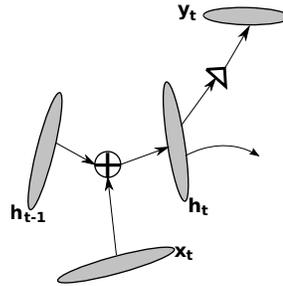


Figure 6.2: A view of an RNN under the operator-based framework: \oplus and \triangleright are the *plus* and *predict* operators, respectively.

6.2 Another Perspective: Neural Operators

In the operator-based framework, one first defines a set of operators of which each is implemented by a multilayer perceptron. For instance, a *plus* operator \oplus may be defined as a function receiving two vectors \mathbf{u} and \mathbf{h} and returning the summary¹ or history \mathbf{h}' of them:

$$\mathbf{h}' = \mathbf{h} \oplus \mathbf{u},$$

where we may constrain that the dimensionality of \mathbf{h} and \mathbf{h}' are identical. Additionally, we can define another operator \triangleright which *predicts* the most likely output symbol \mathbf{y} given a summary \mathbf{h} , such that

$$\mathbf{y} = \triangleright \mathbf{h}$$

It is possible to define many other operators, but for now we stick to these two operators which are sufficient to express standard RNNs as well as the modifications we will later propose.

It is clear to see that the plus operator \oplus and the predict operator \triangleright correspond to the transition function and the output function in Equations (6.1)–(6.2) describing a recurrent model. Thus, at each step, an RNN can be thought as performing the plus operator to update the hidden state given an input ($\mathbf{h}_{[t]} = \mathbf{h}_{[t-1]} \oplus \mathbf{u}_{[t]}$) and then the predict operator to compute the output ($\mathbf{y}_{[t]} = \triangleright \mathbf{h}_{[t]} = \triangleright(\mathbf{h}_{[t-1]} \oplus \mathbf{u}_{[t]})$). See Figure 6.2 for the illustration of how an RNN can be understood from the

1. By summary we meant a summarization of all the previously seen inputs.

operator-based framework.

$$\mathbf{h}_{[t]} = \rho_h(\mathbf{h}_{[t-1]}, \mathbf{u}_t) \quad (6.1)$$

$$\mathbf{y}_{[t]} = \rho_o(\mathbf{h}_{[t]}), \quad (6.2)$$

where ρ_h and ρ_o are a state transition function and an output function, respectively. Each function is parameterized by a set of parameters; $\mathbf{W}^{(rec)}$, \mathbf{b} and $\mathbf{W}^{(out)}$, $\mathbf{b}^{(out)}$. Specifically, for a standard RNN, for example,

$$\rho_h(\mathbf{h}_{[t-1]}, \mathbf{u}_{[t]}) = \sigma(\mathbf{W}^{(rec)}\mathbf{h}_{[t-1]} + \mathbf{W}^{(in)}\mathbf{u}_{[t]} + \mathbf{b})$$

The goal achieved by this operator view is that we assign semantics of all the intermediary computations carried out by the recurrent model. These semantics provide better means to reason about these intermediary quantities. Each operator can be, potentially, parameterized as an MLP with one or more hidden layers, hence a neural operator, since we cannot simply expect the operation will be linear with respect to the input vector(s).

Another advantage of the operator view is that, by the semantics the operators represent, they can provide us insight on how the constructed RNN can be regularized. For instance, one may regularize the model such that the plus operator \oplus is commutative.

Note that this is different from (Mikolov et al., 2013) where the learned embeddings of words happened to be suitable for algebraic operators. The operator-based framework proposed here is rather geared toward *learning* these operators directly.

6.3 Deep Operators

A close analysis of the computation carried out by an RNN (see Figure 6.3) at each time step individually, shows that certain transitions (corresponding to different neural operators) are not deep. They are only results of a linear projection followed by an element-wise nonlinearity.

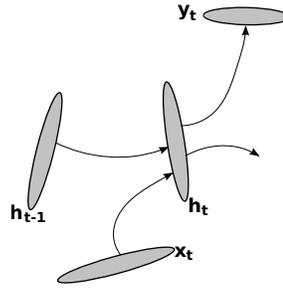


Figure 6.3: A conventional RNN

It is clear that the hidden-to-hidden transition, given by the \oplus operator, ($\mathbf{h}_{[t-1]} \rightarrow \mathbf{h}_{[t]}$), the \triangleright operator representing the hidden-to-output transition ($\mathbf{h}_{[t]} \rightarrow \mathbf{y}_{[t]}$) and input-to-hidden transition ($\mathbf{u}_{[t]} \rightarrow \mathbf{h}_{[t]}$) are all *shallow* in the sense that there exists no intermediate, nonlinear hidden layer.

We can now consider different types of depth of an RNN by considering those transitions separately. We may make the hidden-to-hidden transition deeper by having one or more intermediate nonlinear layers between two consecutive hidden states ($\mathbf{h}_{[t-1]}$ and $\mathbf{h}_{[t]}$). At the same time, the hidden-to-output function can be made deeper, as described previously, by plugging, multiple intermediate nonlinear layers between the hidden state $\mathbf{h}_{[t]}$ and the output $\mathbf{y}_{[t]}$. Each of these choices has a different implication that can be intuitively argued for based on the semantics assigned to them by their corresponding neural operators.

Deep Input-to-Hidden Function

A model can exploit more non-temporal structure from the input by making the input-to-hidden function deep. Previous work has shown that higher-level representations of deep networks tend to better disentangle the underlying factors of variation than the original input (Goodfellow et al., 2009; Glorot et al., 2011b) and flatten the manifolds near which the data concentrate (Bengio et al., 2013). We hypothesize that such higher-level representations should make it easier to learn the temporal structure between successive time steps because the relationship between abstract features can generally be expressed more easily. This has been, for instance, illustrated by the recent work (Mikolov et al., 2013) showing that word embeddings from neural language models tend to be related to their temporal neighbors by simple algebraic relationships, with the same type of relationship

(adding a vector) holding over very different regions of the space, allowing a form of analogical reasoning.

This approach of making the input-to-hidden function deeper is in the line with the standard practice of replacing input with extracted features in order to improve the performance of a machine learning model (see, e.g., [Bengio, 2009](#)). Recently, [Chen and Deng \(2013\)](#) reported that a better speech recognition performance could be achieved by employing this strategy, although they did not jointly train the deep input-to-hidden function together with other parameters of an RNN.

Deep Hidden-to-Output Function

A deep hidden-to-output function can be useful to disentangle the factors of variations in the hidden state, making it easier to predict the output. This allows the hidden state of the model to be more compact and may result in the model being able to summarize the history of previous inputs more efficiently. Let us denote an RNN with this deep hidden-to-output function a deep output RNN (DO-RNN).

Instead of having feedforward, intermediate layers between the hidden state and the output, [Boulanger-Lewandowski et al. \(2012\)](#) proposed to replace the output layer with a conditional generative model such as restricted Boltzmann machines or neural autoregressive distribution estimator ([Larochelle and Murray, 2011](#)). In this chapter we only consider feedforward intermediate layers.

Deep Hidden-to-Hidden Transition

The third knob we can play with is the depth of the hidden-to-hidden transition. The state transition between the consecutive hidden states effectively adds a new input to the summary of the previous inputs represented by the fixed-length hidden state. This interpretation is made apparent, for example, by the \oplus operator, which “sums” together the two inputs, the previous hidden state and the input to the model.

Previous work with RNNs has generally limited the architecture to a shallow operation; affine transformation followed by an element-wise nonlinearity. Instead, we argue that this procedure of constructing a new summary, or a hidden state, from the combination of the previous one and the new input should be highly nonlinear. This nonlinear transition could allow, for instance, the hidden state

of an RNN to rapidly adapt to quickly changing modes of the input, while still preserving a useful summary of the past. This may be impossible to be modeled by a function from the family of generalized linear models. However, this highly nonlinear transition can be modeled by an MLP with one or more hidden layers which has an universal approximator property (see, e.g., [Hornik et al., 1989](#)).

An RNN with this deep transition will be called a deep transition RNN (DT-RNN). This model is shown in Figure 6.4 (a).

This approach of having a deep transition, however, introduces a potential problem. As the introduction of deep transition increases the number of nonlinear steps the gradient has to traverse when propagated back in time, it might become more difficult to train the model to capture long-term dependencies ([Bengio et al., 1994](#)). One possible way to address this difficulty is to introduce shortcut connections (see, e.g., [Raiko et al., 2012](#)) in the deep transition, where the added shortcut connections provide shorter paths, skipping the intermediate layers, through which the gradient is propagated back in time. We refer to an RNN having deep transition with shortcut connections by DT(S)-RNN (See Figure 6.4 (a*)).

Furthermore, we will call an RNN having both a deep hidden-to-output function and a deep transition a deep output, deep transition RNN (DOT-RNN). See Figure 6.4 (b) for the illustration of DOT-RNN. If we consider shortcut connections as well in the hidden to hidden transition, we call the resulting model DOT(S)-RNN.

An approach similar to the deep hidden-to-hidden transition has been proposed recently by [Pinheiro and Collobert \(2014\)](#) in the context of parsing a static scene. They introduced a recurrent convolutional neural network (RCNN) which can be understood as a recurrent network whose the transition between consecutive hidden states (and input to hidden state) is modeled by a convolutional neural network. The RCNN was shown to speed up scene parsing and obtained the state-of-the-art result in Stanford Background and SIFT Flow datasets. [Ko and Dieter \(2009\)](#) proposed deep transitions for Gaussian Process models. Earlier, [Valpola and Karhunen \(2002\)](#) used a deep neural network to model the state transition in a nonlinear, dynamical state-space model.

Stack of Hidden States

An RNN may be extended deeper in yet another way by stacking multiple recurrent hidden layers on top of each other ([Schmidhuber, 1992](#); [El Hahi and](#)

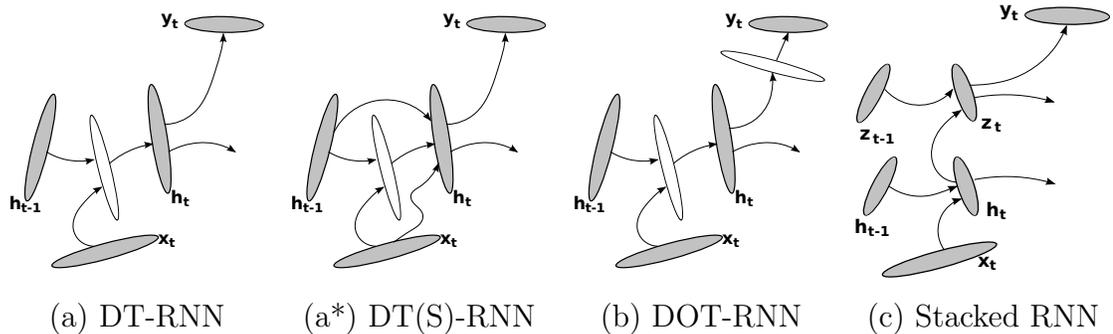


Figure 6.4: Illustrations of four different recurrent neural networks (RNN). (a) Deep Transition (DT) RNN. (a*) DT-RNN with shortcut connections (b) Deep Transition, Deep Output (DOT) RNN. (c) Stacked RNN

Bengio, 1996; Jaeger, 2007a; Graves, 2013). We call this model a stacked RNN (sRNN) to distinguish it from the other proposed variants. The goal of a such model is to encourage each recurrent level to operate at a different timescale.

It should be noticed that the DT-RNN and the sRNN extend the conventional, shallow RNN in different aspects. If we look at each recurrent level of the sRNN separately, it is easy to see that the transition between the consecutive hidden states is still shallow. As we have argued above, this limits the family of functions it can represent. For example, if the structure of the data is sufficiently complex, incorporating a new input frame into the summary of what had been seen up to now might be an arbitrarily complex function. In such a case we would like to model this function by something that has universal approximator properties, as an MLP. The model can not rely on the higher layers to do so, because the higher layers do not feed back into the lower layer. On the other hand, the sRNN can deal with multiple time scales in the input sequence, which is not an obvious feature of the DT-RNN. The DT-RNN and the sRNN are, however, orthogonal in the sense that it is possible to have both features of the DT-RNN and the sRNN by stacking multiple levels of DT-RNNs to build a stacked DT-RNN which we do not explore more in this chapter.

6.3.1 Formal descriptions of deep RNNs

Here we give a more formal description on how the deep transition recurrent neural network (DT-RNN) and the deep output RNN (DO-RNN) as well as the

stacked RNN are implemented.

Deep Transition RNN

We noticed from the state transition equation of the dynamical system simulated by RNNs in Eq. (6.1) that there is no restriction on the form of ρ_h . Hence, we propose here to use a multilayer perceptron to approximate ρ_h instead.

In this case, we can implement ρ_h by L intermediate layers such that

$$\mathbf{h}_{[t]} = \rho_h(\mathbf{h}_{[t-1]}, \mathbf{u}_{[t]}) = \sigma_{[L]} \left(\mathbf{W}_{[L]} \sigma_{[L-1]} \left(\mathbf{W}_{[L-1]} \sigma_{[L-2]} \left(\cdots \sigma_{[1]} \left(\mathbf{W}_{[1]} \mathbf{h}_{[t-1]} + \mathbf{W}^{(in)} \mathbf{u}_{[t]} \right) \right) \right) \right),$$

where $\sigma_{[l]}$ and $\mathbf{W}_{[l]}$ are the element-wise nonlinear function and the weight matrix for the l -th layer. This RNN with a multilayered transition function is a deep transition RNN (DT-RNN).

An illustration of building an RNN with the deep state transition function is shown in Figure 6.4 (a). In the illustration the state transition function is implemented with a neural network with a single intermediate layer.

This formulation allows the RNN to learn a non-trivial, highly nonlinear transition between the consecutive hidden states.

Deep Output RNN

Similarly, we can use a multilayer perceptron with L intermediate layers to model the output function ρ_o in Equation (6.2) such that

$$\mathbf{y}_{[t]} = \rho_o(\mathbf{h}_t) = \sigma_{[L]}^{(out)} \left(\mathbf{W}_{[L]}^{(out)} \sigma_{[L-1]}^{(out)} \left(\mathbf{W}_{[L-1]}^{(out)} \sigma_{[L-2]}^{(out)} \left(\cdots \sigma_{[1]}^{(out)} \left(\mathbf{W}_{[1]}^{(out)} \mathbf{h}_{[t]} \right) \right) \right) \right),$$

where $\sigma_{[l]}^{(out)}$ and $\mathbf{W}_{[l]}^{(out)}$ are the element-wise nonlinear function and the weight matrix for the l -th layer. An RNN implementing this kind of multilayered output function is a deep output recurrent neural network (DO-RNN).

Fig. 6.4 (b) draws a deep output, deep transition RNN (DOT-RNN) implemented using both the deep transition and the deep output with a single intermediate layer each.

Stacked RNN

The stacked RNN (Schmidhuber, 1992; El Hahi and Bengio, 1996) has multiple levels of transition functions defined by

$$\mathbf{h}_{[t]}^{(l)} = \rho_h^{(l)}(\mathbf{h}_{[t-1]}^{(l)}, \mathbf{h}_{[t]}^{(l-1)}) = \sigma\left(\mathbf{W}_{[l]}\mathbf{h}_{[t-1]}^{(l)} + \mathbf{W}_{[l]}\mathbf{h}_{[t]}^{(l-1)}\right),$$

where $\mathbf{h}_{[t]}^{(l)}$ is the hidden state of the l -th level at time t . When $l = 1$, the state is computed using $\mathbf{u}_{[t]}$ instead of $\mathbf{h}_{[t]}^{(l-1)}$. The hidden states of all the levels are recursively computed from the bottom level $l = 1$.

Once the top-level hidden state is computed, the output can be obtained using the usual formula. Alternatively, one may use all the hidden states to compute the output (Hermans and Schrauwen, 2013). Each hidden state at each level may also be made to depend on the input as well (Graves, 2013). Both of them can be considered approaches using shortcut connections discussed earlier.

The illustration of this stacked RNN is in Fig. 6.4 (c).

6.4 Experiments

We train four types of RNNs described in this chapter on a number of benchmark datasets to evaluate their performance. For each benchmark dataset, we try the task of predicting the next symbol.

The task of predicting the next symbol is equivalent to the task of modeling the distribution over a sequence. For each sequence $\mathbf{u}_{[1]}, \dots, \mathbf{u}_{[T]}$, we decompose it into

$$p(\mathbf{u}_{[1]}, \dots, \mathbf{u}_{[T]}) = p(\mathbf{u}_{[1]}) \prod_{t=2}^T p(\mathbf{u}_{[t]} \mid \mathbf{u}_{[1]}, \dots, \mathbf{u}_{[t-1]}),$$

and each term on the right-hand side will be replaced with a single timestep of an RNN. In this setting, the RNN predicts the probability of the next symbol $\mathbf{u}_{[t]}$ in the sequence given the all previous symbols $\mathbf{u}_{[1]}, \dots, \mathbf{u}_{[t-1]}$. Then, we train the RNN by maximizing the log-likelihood.

We try this task of modeling the joint distribution on three different tasks; polyphonic music prediction, character-level and word-level language modeling.

We test the RNNs on the task of polyphonic music prediction using three datasets which are Nottingham, JSB Chorales and MuseData (Boulanger-Lewandowski et al., 2012). On the task of character-level and word-level language modeling, we use Penn Treebank Corpus (Marcus et al., 1993).

6.4.1 Model Descriptions

We compare the conventional recurrent neural network (RNN), deep transition RNN with shortcut connections in the transition MLP (DT(S)-RNN), deep output/transition RNN with shortcut connections in the hidden to hidden transition MLP (DOT(S)-RNN) and stacked RNN (sRNN).

		RNN	DT(S)-RNN	DOT(S)-RNN	sRNN 2 layers
Music	Nottingham	600	400,400	400,400,400	400
	JSB Chorales	200	400,400	400,400,400	400
	MuseData	600	400,400	400,400,400	600
Language	Char-level	600	400,400	400,400,600	400
	Word-level	200	200,200	200,200,200	400

Table 6.1: The sizes of the trained models. For DT(S)-RNN, the two numbers provided are the size of the hidden state and that of the intermediate layer, respectively. For DOT(S)-RNN, the three numbers are the size of the hidden state, that of the intermediate layer between the consecutive hidden states and that of the intermediate layer between the hidden state and the output layer. For sRNN, the number corresponds to the size of the hidden state at each level.

The size of each model is chosen from a limited set $\{100, 200, 400, 600, 800\}$ to minimize the validation error for each polyphonic music task (See Table 6.1 for the final models). In the case of language modeling tasks, we chose the size of the models from $\{200, 400\}$ and $\{400, 600\}$ for word-level and character-level tasks, respectively. In all cases, we use a logistic sigmoid function as an element-wise nonlinearity of each hidden unit. Only for the character-level language modeling we used rectified linear units (Glorot et al., 2011a) for the intermediate layers of the output function, which gave lower validation error. In Gulcehre, Cho, Pascanu, and Bengio (2014) we also explore maxout and L_p units for polyphonic music prediction, which give better performance.

6.4.2 Training

We use stochastic gradient descent, where we clipped the norm of the gradients when exceeding some threshold (see Section 5.7). Training stops when the validation cost stops decreasing.

Polyphonic Music Prediction: For Nottingham and MuseData datasets we compute each gradient step on subsequences of at most 200 steps, while we use subsequences of 50 steps for JSB Chorales. We do not reset the hidden state for each subsequence, unless the subsequence belongs to a different song than the previous subsequence.

The cost (negative log likelihood) is divided by the expected length of the subsequence (200 and 50 respectively). Note that not all subsequences have this length (for e.g. being at the end of a song), though we divide the cost by 200 or 50 in that case as well.

The cutoff threshold for the gradients is set to 1. The hyperparameter for the learning rate schedule¹ is tuned manually for each dataset. We set the hyperparameter β to 2330 for Nottingham, 1475 for MuseData and 100 for JSB Chorales. They correspond to two epochs, a single epoch and a third of an epoch, respectively.

The weights of the connections between any pair of hidden layers are sparse, having only 20 non-zero incoming connections per unit (see, e.g., Sutskever et al., 2013). Each weight matrix is rescaled to have a unit largest singular value. The weights of the connections between the input layer and the hidden state as well as between the hidden state and the output layer are initialized randomly from a Gaussian distribution with standard deviation fixed to 0.1 and 0.01, respectively. In the case of deep output functions (DOT(S)-RNN), the weights of the connections between the hidden state and the intermediate layer are sampled initially from the white Gaussian distribution of standard deviation 0.01. In all cases, the biases are initialized to 0.

To regularize the models, we add white Gaussian noise of standard deviation 0.075 to each weight parameter every time the gradient is computed (Graves, 2011).

Language Modeling: We used the same strategy for initializing the parame-

1. We use at each update τ , the following learning rate $\eta_\tau = \frac{1}{1 + \frac{\max(0, \tau - \tau_0)}{\beta}}$, where τ_0 and β indicate respectively when the learning rate starts decreasing and how quickly the learning rate decreases. In the experiment, we set τ_0 to coincide with the time when the validation error starts increasing for the first time.

ters in the case of language modeling. For character-level modeling, the standard deviations of the white Gaussian distributions for the input-to-hidden weights and the hidden-to-output weights, we used 0.01 and 0.001, respectively, while those hyperparameters were both 0.1 for word-level modeling. In the case of DOT(S)-RNN, we sample the weights of between the hidden state and the rectifier intermediate layer of the output function from the white Gaussian distribution of standard deviation 0.01. When using rectifier units (character-based language modeling) we fix the biases to 0.1.

In language modeling, the learning rate starts from an initial value and is halved each time the validation cost does not decrease significantly (Mikolov et al., 2010). We do not use any regularization for the character-level modeling, but for the word-level modeling we use the same strategy of adding weight noise as we do with the polyphonic music prediction.

For all the tasks (polyphonic music prediction, character-level and word-level language modeling), the stacked RNN and the DOT(S)-RNN were initialized with the weights of the conventional RNN and the DT(S)-RNN, which is similar to layer-wise pretraining of a feedforward neural network (see, e.g., Hinton and Salakhutdinov, 2006). We use a ten times smaller learning rate for each parameter that was pretrained as either RNN or DT(S)-RNN.

	RNN	DT(S)-RNN	DOT(S)-RNN	sRNN	DOT(S)-RNN*
Nothingam	3.225	3.206	3.215	3.258	2.95
JSB Chorales	8.338	8.278	8.437	8.367	7.92
MuseData	6.990	6.988	6.973	6.954	6.59

Table 6.2: The performances of the four types of RNNs on the polyphonic music prediction. The numbers represent negative log-probabilities on test sequences. (*) We obtained these results using DOT(S)-RNN with L_p units in the deep transition, maxout units in the deep output function and dropout (Gulcehre, Cho, Pascanu, and Bengio, 2014).

6.4.3 Result and Analysis

Polyphonic Music Prediction

The log-probabilities on the test set of each data are presented in the first four columns of Table 6.2. We were able to observe that in all cases one of the proposed deep RNNs outperformed the conventional, shallow RNN. Though, the

suitability of each deep RNN depended on the data it was trained on. The best results obtained by the DT(S)-RNNs on Nottingham and JSB Chorales are close to, but worse than the result obtained by RNNs trained with the technique of fast dropout (FD) which are 3.09 and 8.01, respectively (Bayer et al., 2014).

In order to quickly investigate whether the proposed deeper variants of RNNs may also benefit from the recent advances in feedforward neural networks, such as the use of non-saturating activation functions¹ and the method of dropout. We have built another set of DOT(S)-RNNs that have the recently proposed L_p units (Gulcehre, Cho, Pascanu, and Bengio, 2014) in deep transition and maxout units (Goodfellow et al., 2013) in deep output function. Furthermore, we used the method of dropout (Hinton et al., 2012) instead of weight noise during training. Similarly to the previously trained models, we searched for the size of the models as well as other learning hyperparameters that minimize the validation performance. We, however, did not pretrain these models.

The results obtained by the DOT(S)-RNNs having L_p and maxout units trained with dropout are shown in the last column of Table 6.2. On every music dataset the performance by this model is significantly better than those achieved by all the other explored models as well as the best results reported with recurrent neural networks in (Bayer et al., 2014). This suggests us that the proposed variants of deep RNNs also benefit from having non-saturating activations and using dropout, just like feedforward neural networks. We reported these results and more details on the experiment in (Gulcehre, Cho, Pascanu, and Bengio, 2014).

We, however, acknowledge that the model-free state of the art results for the music datasets were obtained using an RNN combined with a conditional generative model, such as restricted Boltzmann machines or neural autoregressive distribution estimator (Larochelle and Murray, 2011), in the output (Boulanger-Lewandowski et al., 2012).

1. Note that it is not trivial to use non-saturating activation functions in conventional RNNs, as this may cause the explosion of the activations of hidden states. However, it is perfectly safe to use non-saturating activation functions at the intermediate layers of a deep RNN with deep transition.

1. Reported by Mikolov et al. (2012) using mRNN with Hessian-free optimization technique.
2. Reported by Mikolov et al. (2011) using the dynamic evaluation.
3. Reported by Graves (2013) using the dynamic evaluation and weight noise.

	RNN	DT(S)-RNN	DOT(S)-RNN	sRNN	*	★
Character-Level	1.414	1.409	1.386	1.412	1.41 ¹	1.24 ³
Word-Level	117.7	112.0	107.5	110.0	123 ²	117 ³

Table 6.3: The performances of the four types of RNNs on the tasks of language modeling. The numbers represent bit-per-character and perplexity computed on test sequence, respectively, for the character-level and word-level modeling tasks. * The previous/current state-of-the-art results obtained with shallow RNNs. ★ The previous/current state-of-the-art results obtained with RNNs having long-short term memory units.

Language Modeling

In Table 6.3, we can see the perplexities on the test set achieved by all four models. Deep RNNs (DT(S)-RNN, DOT(S)-RNN and sRNN) outperform the conventional, shallow RNN significantly. On these tasks DOT(S)-RNN outperformed all the other models, which suggests that it is important to have nonlinear mapping from the hidden state to the output in the case of language modeling.

The results by both the DOT(S)-RNN and the sRNN for word-level modeling surpassed the previous best performance achieved by an RNN with 1000 long short-term memory (LSTM) units (Graves, 2013) as well as that by a shallow RNN with a larger hidden state (Mikolov et al., 2011), even when both of them used dynamic evaluation¹. The results we report here are with out dynamic evaluation.

For character-level modeling the state of the art results were obtained using an optimization method Hessian-free with a specific type of RNN architecture called mRNN (Mikolov et al., 2012) or a regularization technique called adaptive weight noise (Graves, 2013). Our result, however, is better than the performance achieved by conventional, shallow RNNs with out any of these enhancements (Mikolov et al., 2012), where they reported the best performance of 1.41 using an RNN trained with the Hessian-free learning algorithm (Martens and Sutskever, 2011).

1. Dynamic evaluation refers to an approach where the parameters of a model are updated as the validation/test data is predicted.

6.5 Conclusion and Outlook

In this chapter we looked at the meaning of *depth* for recurrent neural networks. The question we are after is how efficient is a recurrent model in terms of the size needed to represent some behaviour. Model size is not only connected to computational costs, but also to statistical efficiency in learning. By the arguments used for feedforward models, we can hope that depth can make recurrent models more efficient at representing certain behaviours. Unfortunately, in contrast to feedforward models, for recurrent networks, the meaning of *depth* is ambiguous.

A recurrent network can be understood as a series of operations that are repeated at each time step. For example, constructing the new hidden state can be seen as such an operation, that takes the previous history (hidden state) and the current input and constructs a new state that incorporates the new input example. From this operator view of the recurrent model it is obvious that any such operator that composes one step of the model can be made deep. This leads to different variants of deep recurrent models, each with different characteristics. For example, the DT-RNN has deep transitions between hidden states and it is able to learn more efficiently behaviours that involve very non-smooth changes in the hidden state, changes that also depend on the context.

We limit this work to an empirical evaluation of these different variants of deep recurrent models. However, a similar approach to the one used in Chapter 3 might be a possible future direction to also provide a theoretical treatment of this questions. It could also lead to insights into what kind of behaviours the rectifier model can learn.

Yet another direction that we intend to pursue as future work is to understand how these structural changes interact with different phenomena that can be obtained in recurrent models. For example, what are the properties of the weight matrices compounding the transition of a DT-RNN for this model to have the echo state property?

Another practical direction is to explore and validate the different alterations proposed for deep feedforward models within these deep transitions. For example, unbounded, piecewise linear functions seem to behave very well for feedforward networks. Using rectifiers for RNN, however, is not straight forward, as it can easily lead to an unstable system. Instead of dealing with only the exploding

gradients problem, now even the forward activation can explode. Arguably, bits of information are stored in the activation of the hidden units by saturating the sigmoid units. This also can not be done with unbounded activations. For a DT-RNN model, we do not need to deal with these issues if we use these kind of activations in the intermediary layers of the deep transitions.

Additionally, more explorations is needed on how these different ways of being deep can be used together and what different aspects of a problem they might address. Constructing rules of thumb for these variants of the model is important, as it is to be expected that the kind of deep RNN that performs best depends on the task at hand.

Finally, as observed in the experiments we run, learning is more difficult for these models. Preliminary experiments that used more than a single intermediary layer in the deep hidden to hidden transitions severely underperformed. As future work, we intend to explore different optimization algorithms, such as natural gradient descent or Saddle-Free Newton method. We also want, in parallel, to look at the regularization term introduced in Section 5.8 that prevents gradients to vanish as a possible solution of this learning difficulty. And, other strategies to improve learning that we want to consider are to either employ layer-wise pretraining, to use additional local error signals or to use better activation functions and initializations of the intermediary layers.

7

A final remark

The work presented in this thesis is an attempt of deepening our understanding of deep models. We started off in Chapter 2 with an introduction to the relevant concepts for the rest of the document. In Chapter 3 we analysed the importance of *depth*, showing that deep rectifier models can be exponentially more efficient (for a fixed number of parameters) at modelling certain families of functions than shallow models. Also we noted that the restriction of this efficiency to functions that are symmetric in their input space is the reason why deep models can generalize well.

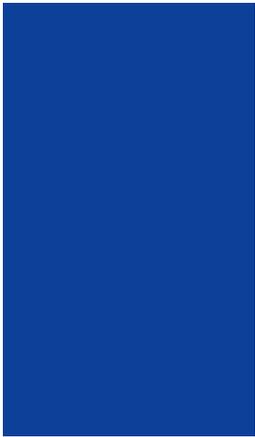
Chapter 4 looked at the optimization problem. While we know that deep models can be more expressive, it is not clear how well we can find a suitable parametrization for them. We explored the relationship between a few higher-order optimization techniques, including natural gradient descent, Hessian-Free Optimization and Krylov Subspace Descent. We proposed a new framework and showed that all these algorithm can be casted into it. We also introduced the saddle-free problem and provided motivation for why it is important to address saddle points in large scale non-convex problems. We also introduced a new algorithm called Saddle-Free Newton method, that should move optimally around saddle points.

Chapters 5–6 focused on a special class of deep models, namely recurrent networks. Compared to feedforward models, recurrent ones are as powerful as Turing machines. However, learning complex behaviour in recurrent models is not trivial. In Chapter 5 we looked at one such complex behaviour, working memory. We asked the question of whether this behaviour can be learnt and what are the basic mechanism behind it. In Chapter 6 we returned to the question stated previously, in Chapter 3, namely that of efficiency. Based on intuitions from feedforward models, we explored ways into which a recurrent model can be made deep as well, and empirically showed the efficiency of these deep recurrent models.

Similar to other work in the field, our efforts are just a step towards answering these questions. And the answers we provide also end up leading to more questions,

but provide new possible research directions for the future.

Now, at the end of this document, I want to *thank you*, reader, for going through the intricate intuitions that I attempted to put forward. The thought of having someone spend time going through the many different sections of this thesis is what pushed me to continue writing and to not spare any detail. I hope you found this work interesting and useful and that it managed to inspire you and helped you to define your own questions and/or to find your own answers. I apologize for any inconsistencies that you might have encountered.



Bibliography

- Absil, P.-A., R. Mahony, and R. Sepulchre (2008). Optimization Algorithms on Matrix Manifolds. *Princeton, NJ: Princeton University Press*. (Cited on page 109.)
- Amari, S. (1985). Differential geometrical methods in statistics. *Lecture notes in statistics 28*. (Cited on pages 41, 96, 107 and 108.)
- Amari, S. (1997). Neural learning in structured parameter spaces - natural Riemannian gradient. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 127–133. (Cited on page 41.)
- Amari, S., K. Kurata, and H. Nagaoka (1992). Information geometry of Boltzmann machines. In *IEEE Transactions on Neural Networks 3*, 260–271. (Cited on pages 41 and 96.)
- Amari, S.-I. (1998). Natural gradient works efficiently in learning. In *Journal of Neural Computation 10*(2), 251–276. (Cited on pages 41 and 96.)
- Arnold, L., A. Auger, N. Hansen, and Y. Olivier (2011). Information-geometric optimization algorithms: A unifying picture via invariance principles. *Technical Report ArXiv:1106.3708*. (Cited on pages 41 and 96.)
- Atiya, A. F. and A. G. Parlos (2000). New results on recurrent network training: Unifying the algorithms and accelerating convergence. In *IEEE Transactions on Neural Networks 11*, 697–709. (Cited on pages 26 and 32.)
- Babloyantz, A. and C. Lourenço (1994). Computation with chaos: A paradigm for cortical activity. In *Proceedings of the National Academy of Sciences of the USA 91*, 9027–9031. (Cited on page 164.)
- Baldi, P. and K. Hornik (1989). Neural networks and principal component analysis: Learning from examples without local minima. In *Neural Networks 2*, 53–58. (Cited on page 120.)

-
- Barron, A. R. (1993). Universal approximation bounds for superpositions of a sigmoidal function. In *IEEE Transactions on Information Theory* 39(3), 930–945. (Cited on page 20.)
- Barron, A. R. (1994). Approximation and estimation bounds for artificial neural networks. In *Journal of Machine Learning* 14, 115–133. (Cited on page 20.)
- Bastien, F., P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, and Y. Bengio (2012). Theano: new features and speed improvements. In *Deep Learning and Unsupervised Feature Learning NIPS Workshop*. (Cited on pages 3 and 193.)
- Bayer, J., C. Osendorfer, D. Korhammer, N. Chen, S. Urban, and P. van der Smagt (2013). On fast dropout and its applicability to recurrent networks. In *International Conference on Learning Representations (ICLR): Conference Track*. (Cited on page 224.)
- Bengio, Y. (2009). Learning deep architectures for AI. In *Foundations and Trends in Machine Learning* 2(1), 1–127. Also published as a book. Now Publishers, 2009. (Cited on pages 44, 212 and 216.)
- Bengio, Y., F. Bastien, A. Bergeron, N. Boulanger-Lewandowski, T. Breuel, Y. Chherawala, M. Cisse, M. Côté, D. Erhan, J. Eustache, X. Glorot, X. Muller, S. Pannetier Lebeuf, R. Pascanu, S. Rifai, F. Savard, and G. Sicard (2011). Deep learners benefit more from out-of-distribution examples. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*. (Cited on page 148.)
- Bengio, Y., N. Boulanger-Lewandowski, and R. Pascanu (2013). Advances in optimizing recurrent networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. (Cited on pages 3, 200 and 206.)
- Bengio, Y. and O. Delalleau (2011). On the expressive power of deep architectures. In *Algorithmic Learning Theory*, Volume 6925 of *Lecture Notes in Computer Science*, pp. 18–36. (Cited on page 45.)
- Bengio, Y., P. Lamblin, D. Popovici, and H. Larochelle (2007). Greedy layer-wise training of deep networks. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 153–160. (Cited on page 1.)

-
- Bengio, Y., G. Mesnil, Y. Dauphin, and S. Rifai (2013). Better mixing via deep representations. In *International Conference on Machine Learning (ICML)*. (Cited on page 215.)
- Bengio, Y., P. Simard, and P. Frasconi (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks* 5(2), 157–166. (Cited on pages 24, 32, 33, 174, 179, 207 and 217.)
- Bergstra, J., R. Bardenet, Y. Bengio, and B. Kégl (2011). Algorithms for hyperparameter optimization. In *Advances on Neural Information Processing Systems (NIPS)*. (Cited on page 12.)
- Bergstra, J. and Y. Bengio (2012). Random search for hyper-parameter optimization. In *Journal of Machine Learning Research (JMLR)* 13, 281–305. (Cited on pages 12 and 133.)
- Bergstra, J., O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio (2010). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*. (Cited on pages 3, 137 and 193.)
- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer. (Cited on pages 9, 40 and 99.)
- Boulanger-Lewandowski, N., Y. Bengio, and P. Vincent (2012). Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. In *International Conference of Machine Learning (ICML)*. (Cited on pages 198, 216, 221 and 224.)
- Bray, A. J. and D. S. Dean (2007, Apr). Statistics of critical points of gaussian fields on large-dimensional spaces. In *Physics Review Letters* 98. (Cited on pages 119, 120 and 136.)
- Buehner, M. and P. Young (2006). A tighter bound for the echo state property. In *IEEE Transactions on Neural Networks* 17(3), 820–824. (Cited on pages 35 and 168.)
- Callahan, J. (2010). *Advanced Calculus: A Geometric View*. Undergraduate Texts in Mathematics. Springer. (Cited on pages 118 and 119.)

-
- Casey, M. (1996, August). The dynamics of discrete-time computation, with application to recurrent neural networks and finite state machine extraction. In *Journal of Neural Computation* 8(6), 1135–1178. (Cited on page 22.)
- Chapelle, O. and D. Erhan (2011). Improved preconditioner for Hessian-Free Optimization. In *Deep Learning and Unsupervised Feature Learning NIPS Workshop*. (Cited on page 137.)
- Chen, J. and L. Deng (2013). A new method for learning deep recurrent neural networks. *Technical Report ArXiv:1311.6091*. (Cited on page 216.)
- Choi, S.-C. T., C. C. Paige, and M. A. Saunders (2011). MINRES-QLP: A Krylov subspace method for indefinite or singular symmetric systems. *Technical Report ArXiv:1301.2707*. (Cited on pages 87 and 95.)
- Cruse, H. (2009). Neural networks as cybernetic systems - 3rd and revised edition. In *Brains, Minds & Media* (3). (Cited on page 25.)
- Dahl, G. E., M. Ranzato, A. Mohamed, and G. E. Hinton (2010). Phone recognition with the mean-covariance restricted Boltzmann machine. In *Advances in Neural Information Processing Systems (NIPS)*. (Cited on pages 2 and 44.)
- Delalleau, O. and Y. Bengio (2011). Shallow vs. deep sum-product networks. In *Advances in Neural Information Processing Systems (NIPS)*. (Cited on page 212.)
- Desjardins, G. (2009). Training Deep Convolutional Architectures for Vision. *Master Thesis, University of Montreal*. (Cited on page 19.)
- Desjardins, G., R. Pascanu, A. Courville, and Y. Bengio (2013). Metric-free natural gradient for joint-training of Boltzmann machines. In *International Conference on Learning Representations (ICLR): Conference Track*. (Cited on pages 3, 5, 96 and 144.)
- Dominey, P. F., M. A. Arbib, and J. P. Joseph (1995). A model of cortico-striatal plasticity for learning oculomotor associations and sequences. In *Cognitive Neuroscience Journal*. (Cited on page 35.)
- Doya, K. (1993). Bifurcations of recurrent neural networks in gradient descent learning. In *IEEE Transactions on Neural Networks* 1, 75–80. (Cited on pages 186, 187 and 190.)

-
- Doya, K. and S. Yoshizawa (1991). Adaptive synchronization of neural and physical oscillators. In *Advances of Neural Information Processing Systems (NIPS)*, pp. 109–116. (Cited on page 190.)
- Duchi, J., E. Hazan, and Y. Singer (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research (JMLR)*. (Cited on page 39.)
- Durstewitz, D., J. K. Seamans, and T. J. Sejnowski (2000). Neurocomputational models of working memory. In *Nature Neuroscience* 3, 1184–91. (Cited on pages 162, 163 and 164.)
- Eck, D. and J. Schmidhuber (2002). Finding temporal structure in music: Blues improvisation with LSTM recurrent networks. In *International Workshop on Neural Networks for Signal Processing (NNSP)*, pp. 747–756. (Cited on page 35.)
- El Hahi, S. and Y. Bengio (1996). Hierarchical recurrent neural networks for long-term dependencies. In *Advances in Neural Information Processing Systems (NIPS)*. (Cited on pages 25, 212, 217 and 220.)
- Elman, J. L. (1990). Finding structure in time. In *Cognitive Science* 14, 179–211. (Cited on pages 24 and 25.)
- Elman, J. L. and D. Zipser (1988). Learning the Hidden Structure of Speech. In *Journal of the Acoustical Society of America* 83, pp. 1615–1625. (Cited on page 25.)
- Erhan, D., A. Courville, Y. Bengio, and P. Vincent (2010). Why does unsupervised pre-training help deep learning? In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 201–208. (Cited on page 148.)
- Freeman, W. J. (2007a). Definitions of state variables and state space for brain-computer interface. part 1: Multiple hierarchical levels of brain function. In *Cognitive Neurodynamics* 1(1), 3–14. (Cited on page 164.)
- Freeman, W. J. (2007b). Definitions of state variables and state space for brain-computer interface. part 2. extraction and classification of feature vectors. In *Cognitive Neurodynamics* 1(2), 85–96. (Cited on page 164.)

-
- Freund, Y. and D. Haussler (1994). Unsupervised learning of distributions on binary vectors using two layer networks. *Technical Report UCSC-CRL-94-25*, University of California, Santa Cruz. (Cited on page 15.)
- Fyodorov, Y. V. and I. Williams (2007). Replica symmetry breaking condition exposed by random matrix calculation of landscape complexity. In *Journal of Statistical Physics* 129(5-6), 1081–1116. (Cited on pages 119 and 120.)
- Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. In *Biological Cybernetics*, 36, pp. 193–202. (Cited on page 19.)
- Ganguli, S. and H. Sompolinsky (2010). Short-term memory in neuronal networks through dynamical compressed sensing. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 667–675. (Cited on pages 194 and 195.)
- Glorot, X., A. Bordes, and Y. Bengio (2011a). Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*. (Cited on pages 46 and 221.)
- Glorot, X., A. Bordes, and Y. Bengio (2011b). Domain adaptation for large-scale sentiment classification: A deep learning approach. In *International Conference on Machine Learning (ICML)*. (Cited on page 215.)
- Gonzalez, A. and J. Dorronsoro (2006). Natural conjugate gradient training of multilayer perceptrons. In *Artificial Neural Networks ICANN 2006*, pp. 169–177. (Cited on pages 109, 110, 137 and 142.)
- Goodfellow, I., Q. Le, A. Saxe, and A. Ng (2009). Measuring invariances in deep networks. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 646–654. (Cited on page 215.)
- Goodfellow, I. J., Y. Bulatov, J. Ibarz, S. Arnaud, and V. Shet (2014). Multi-digit number recognition from street view imagery using deep convolutional neural networks. In *International Conference on Learning Representations (ICLR): Conference Track*. (Cited on pages 2 and 44.)

-
- Goodfellow, I. J., M. Mirza, A. Courville, and Y. Bengio (2013). Multi-prediction deep Boltzmann machines. In *Advances in Neural Information Processing Systems (NIPS)*. (Cited on page 27.)
- Goodfellow, I. J., D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio (2013). Maxout networks. In *International Conference on Machine Learning (ICML)*. (Cited on pages 2, 3, 212 and 224.)
- Graves, A. (2011). Practical variational inference for neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 2348–2356. (Cited on page 222.)
- Graves, A. (2013, August). Generating sequences with recurrent neural networks. *Technical Report ArXiv:1308.0850*. (Cited on pages 33, 191, 199, 218, 220, 224 and 225.)
- Graves, A., N. Jaitly, and A. Mohamed (2013). Hybrid speech recognition with deep bidirectional lstm. In *IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pp. 273–278. (Cited on pages 2, 35, 36 and 44.)
- Graves, A., M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, and J. Schmidhuber (2009). A novel connectionist system for unconstrained handwriting recognition. In *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31(5), 855–868. (Cited on pages 2, 35 and 36.)
- Graves, A., S. Fernandez and J. Schmidhuber (2007). Multi-Dimensional Recurrent Neural Networks. In *International Conference on Artificial Neural Networks*, pp. 549–558. (Cited on page 27.)
- Gros, C. (2009). Cognitive computation with autonomously active neural networks: an emerging field. In *Cognitive Computation* 1, 77–90. (Cited on page 164.)
- Gulcehre, C. and Y. Bengio (2013). Knowledge matters: Importance of prior information for optimization. In *International Conference on Learning Representations (ICLR): Conference Track*. (Cited on page 167.)
- Gulcehre, C., K. Cho, R. Pascanu, and Y. Bengio (2014). Learned-norm pooling for deep feedforward and recurrent neural networks. *Technical Report ArXiv:1311.1780*. (Cited on pages 3, 221, 223 and 224.)

-
- Hajnal, A., W. Maass, P. Pudlak, M. Szegedy, and G. Turan (1993). Threshold circuits of bounded depth. In *Journal of Computer and System Sciences* 46(2), 129–154. (Cited on page 45.)
- Håstad, J. (1986). Almost optimal lower bounds for small depth circuits. In *Proceedings of the 18th annual ACM Symposium on Theory of Computing*, pp. 6–20. (Cited on page 45.)
- Håstad, J. and M. Goldmann (1991). On the power of small-depth threshold circuits. In *Computational Complexity* 1, 113–129. (Cited on page 45.)
- Hermans, M. and B. Schrauwen (2013). Training and analysing deep recurrent neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 190–198. (Cited on pages 212 and 220.)
- Heskes, T. (2000). On natural learning and pruning in multilayered perceptrons. In *Journal of Neural Computation* 12, 1037–1057. (Cited on pages 96 and 104.)
- Hinton, G., L. Deng, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury (2012). Deep neural networks for acoustic modeling in speech recognition. In *IEEE Signal Processing Magazine* 29(6), 82–97. (Cited on pages 2, 44 and 212.)
- Hinton, G. E., S. Osindero, and Y. Teh (2006). A fast learning algorithm for deep belief nets. In *Journal of Neural Computation* 18, 1527–1554. (Cited on page 1.)
- Hinton, G. E. and R. Salakhutdinov (2006). Reducing the dimensionality of data with neural networks. In *Science* 313(5786), 504–507. (Cited on pages 21 and 223.)
- Hinton, G. E., N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2012). Improving neural networks by preventing co-adaptation of feature detectors. *Technical Report ArXiv:1207.0580*. (Cited on pages 58 and 224.)
- Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. *Ph.D. thesis, Technical University of Munich*. (Cited on pages 24, 25, 32, 33, 190 and 207.)

-
- Hochreiter, S. and J. Schmidhuber (1997). Long short-term memory. In *Journal of Neural Computation* 9(8), 1735–1780. (Cited on pages 24, 25, 32, 33, 34, 190, 191, 195, 198 and 208.)
- Holzmann, G. and H. Hauser (2009). Echo state networks with filter neurons and a delay&sum readout. In *Neural Networks* 23(2), 244–256. (Cited on page 36.)
- Honkela, A., T. Raiko, M. Kuusela, M. Tornio, and J. Karhunen (2010). Approximate Riemannian conjugate gradient learning for fixed-form variational bayes. In *Journal of Machine Learning Research (JMLR)* 11, 3235–3268. (Cited on pages 109 and 110.)
- Honkela, A., M. Tornio, and T. Raiko (2006). Variational Bayes for continuous-time nonlinear state-space models. In *Dynamical Systems, Stochastic Processes and Bayesian Inference NIPS Workshop*, (Cited on page 27.)
- Honkela, A., M. Tornio, T. Raiko, and J. Karhunen (2008). Natural conjugate gradient in variational inference. In *Neural Information Processing*, pp. 305–314. (Cited on pages 109 and 137.)
- Horne, B. G., C. L. Giles, P. C. Collingwood, S. O. Computing, M. Sci, P. Tino, and P. Tino (1998). Finite state machines and recurrent neural networks – automata and dynamical systems approaches. In *Neural Networks and Pattern Recognition*, pp. 171–220. (Cited on page 22.)
- Hornik, K., M. Stinchcombe, and H. White (1989). Multilayer feedforward networks are universal approximators. *Neural Networks* 2, 359–366. (Cited on pages 19 and 217.)
- Howard, M. (2009). Computational models of memory. In *The new encyclopedia of neuroscience*. Elsevier. (Cited on page 163.)
- Hubel, D. and Wiesel, T. (1968). Receptive fields and functional architecture of monkey striate cortex. *Journal of Physiology (London)*, 195, pp. 215–243. (Cited on page 18.)
- Hyötyniemi, H. (1996). Turing machines are recurrent neural networks. In *Proceedings of Finnish Artificial Intelligence Conference (STeP)*, pp. 13–24. (Cited on page 24.)

-
- Ilies, I., H. Jaeger, O. Kosuchinas, M. Rincon, V. Sakenas, and N. Vaskevicius (2007). Stepping forward through echoes of the past: forecasting with echo state networks. *Technical Report*, Jacobs University Bremen. (Cited on page 35.)
- Inoue, M., H. Park, and M. Okada (2003). On-line learning theory of soft committee machines with correlated hidden units –steepest gradient descent and natural gradient descent–. In *Journal of the Physical Society of Japan* 72(4), 805–810. (Cited on pages 121 and 124.)
- Jaeger, H. (1995). Identification of behaviors in an agent’s phase space. *GMD Technical Report 951*, German National Research Institute for Computing Science. (Cited on page 164.)
- Jaeger, H. (2001). The ”echo state” approach to analysing and training recurrent neural networks. *GMD Technical Report 148*, German National Research Institute for Computer Science. (Cited on pages 26, 35, 168 and 191.)
- Jaeger, H. (2002). Short term memory in echo state networks. *GMD Technical Report 152*, German National Research Institute for Computer Science. (Cited on page 170.)
- Jaeger, H. (2007a). Discovering multiscale dynamical features with hierarchical echo state networks. *Technical Report*, Jacobs University. (Cited on page 218.)
- Jaeger, H. (2007b). Echo state network. In *Scholarpedia*, Volume 2, pp. 2330. (Cited on page 168.)
- Jaeger, H. (2009). Advanced Machine Learning. *Lecture Notes*. (Cited on page 22.)
- Jaeger, H. (2013). Long short-term memory in echo state networks: Details of a simulation study. *Technical Report*, Jacobs University. (Cited on pages 36, 191 and 198.)
- Jaeger, H. and H. Haas (2004). Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. In *Science* 304(5667), 78–80. (Cited on pages 2, 35 and 36.)
- Jordan, M. I. (1990). Attractor dynamics and parallelism in a connectionist sequential machine. In *Artificial Neural Networks*, pp. 112-127. (Cited on page 25.)

-
- Kakade, S. (2001). A natural policy gradient. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 1531–1538. (Cited on pages [41](#) and [96](#).)
- Kalinke, Y. and H. Lehmann (1998). Computation in recurrent neural networks: From counters to iterated function systems. In *Advanced Topics in Artificial Intelligence, volume 1502 of LNAI*. Springer. (Cited on page [22](#).)
- Kiros, R. (2013). Training neural networks with stochastic Hessian-Free Optimization. In *International Conference of Learning Representation (ICLR): Conference Track*. (Cited on page [142](#).)
- Ko, J. and F. Dieter (2009). Gp-bayesfilters: Bayesian filtering using gaussian process prediction and observation models. In *Autonomous Robots*. (Cited on page [217](#).)
- Kohonen, T., M. R. Schroeder, and T. S. Huang (Eds.) (2001). *Self-Organizing Maps* (3rd ed.). Secaucus, NJ, USA: Springer-Verlag New York, Inc. (Cited on page [25](#).)
- Krizhevsky, A., I. Sutskever, and G. Hinton (2012). ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS)*. (Cited on pages [2](#), [44](#) and [212](#).)
- Larochelle, H., M. Mandel, R. Pascanu, and Y. Bengio (2012, March). Learning algorithms for the classification restricted Boltzmann machine. In *Journal of Machine Learning Research (JMLR)* *13*, 643–669. (Cited on page [3](#).)
- Larochelle, H. and I. Murray (2011). The Neural Autoregressive Distribution Estimator. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*. (Cited on pages [216](#) and [224](#).)
- Le Roux, N. and Y. Bengio (2010). Deep belief networks are compact universal approximators. In *Journal of Neural Computation* *22*(8), 2192–2207. (Cited on pages [45](#) and [212](#).)
- Le Roux, N. and A. Fitzgibbon (2010). A fast natural newton method. In *International Conference on Machine Learning (ICML)*, pp. 623–630. (Cited on pages [108](#), [109](#), [113](#) and [114](#).)

-
- Le Roux, N., P.-A. Manzagol, and Y. Bengio (2008). Topmoumoute online natural gradient algorithm. In *Advances on Neural Information Processing Systems (NIPS)*. (Cited on pages [96](#), [106](#), [107](#) and [125](#).)
- LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner (1998). Gradient based learning applied to document recognition. In *IEEE Proceedings*. (Cited on pages [19](#) and [38](#).)
- Lee, H., R. Grosse, R. Ranganath, and A. Y. Ng (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *International Conference on Machine Learning (ICML)*. (Cited on page [44](#).)
- Lloyd, S. P. (1982). Least squares quantization in PCM. In *IEEE Transactions on Information Theory* *28*(2), 129–137. (Cited on page [15](#).)
- Lukosevicius, M., H. Jaeger, D. Popovici, and U. Siewert (2007). Optimization and applications of echo state network with leaky integrator neurons. In *Neural Networks*. (Cited on pages [35](#) and [191](#).)
- Lukoševičius, M. and H. Jaeger (2009, August). Reservoir computing approaches to recurrent neural network training. In *Computer Science Review* *3*(3), 127–149. (Cited on pages [26](#) and [35](#).)
- Maass, W. and C. M. Bishop (Eds.) (2001). *Pulsed Neural Networks*. Cambridge, MA, USA: MIT Press. (Cited on pages [26](#) and [35](#).)
- Maass, W., P. Joshi, and E. Sontag (2007). Computational aspects of feedback in neural circuits. In *PLOS Computational Biology* *3*(1), 1–20. (Cited on pages [165](#), [175](#) and [207](#).)
- Mandel, M., R. Pascanu, D. Eck, Y. Bengio, L. M. Aeillo, R. Schifanella, and F. Menczer (2011, October). Contextual tag inference. In *ACM Transactions on Multimedia Computing, Communications and Applications*. (Cited on page [3](#).)
- Manjunath, G. and H. Jaeger (2013). Echo state property linked to an input: Exploring a fundamental characteristic of recurrent neural networks. In *Journal on Neural Computation*, Volume 25. (Cited on pages [159](#) and [168](#).)

-
- Marcus, M. P., M. A. Marcinkiewicz, and B. Santorini (1993, June). Building a large annotated corpus of English: The Penn Treebank. In *Computational Linguistics* 19(2), 313–330. (Cited on pages [199](#) and [221](#).)
- Martens, J. (2010). Deep learning via Hessian-Free optimization. In *International Conference on Machine Learning (ICML)*, pp. 735–742. (Cited on pages [101](#), [105](#), [107](#) and [137](#).)
- Martens, J., A. Chattopadhyaya, T. Pitassi, and R. Zemel (2013). On the expressive power of Restricted Boltzmann Machines. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 2877–2885. (Cited on page [45](#).)
- Martens, J. and I. Sutskever (2011). Learning Recurrent Neural Networks with Hessian-Free Optimization. In *International Conference on Machine Learning (ICML)*, pp. 1033–1040. (Cited on pages [26](#), [31](#), [36](#), [101](#), [106](#), [184](#), [190](#), [195](#), [196](#), [198](#), [204](#) and [225](#).)
- May, R. (1976). Simple mathematical models with very complicated dynamics. In *Nature* 261(5560), 459–467. (Cited on page [162](#).)
- Medler, D. A. (1998). A brief history of connectionism. In *Neural Computing Surveys* 1, 61–101. (Cited on page [1](#).)
- Mikolov, T. (2012). *Statistical Language Models based on Neural Networks*. Ph. D. thesis, Brno University of Technology. (Cited on page [191](#).)
- Mikolov, T., K. Chen, G. Corrado, and J. Dean (2013). Efficient estimation of word representations in vector space. In *International Conference on Learning Representations (ICLR): Workshops Track*. (Cited on page [215](#).)
- Mikolov, T., A. Deoras, S. Kombrink, L. Burget, and J. Cernocky (2011). Empirical evaluation and combination of advanced language modeling techniques. In *Conference of the International Speech Communication Association (INTERSPEECH)*. (Cited on pages [2](#), [44](#) and [191](#).)
- Mikolov, T., M. Karafiát, L. Burget, J. Cernocky, and S. Khudanpur (2010). Recurrent neural network based language model. In *Conference of the International Speech Communication Association (INTERSPEECH)*, pp. 1045–1048. (Cited on page [223](#).)

-
- Mikolov, T., S. Kombrink, L. Burget, J. Cernocky, and S. Khudanpur (2011). Extensions of recurrent neural network language model. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. (Cited on pages 224 and 225.)
- Mikolov, T., I. Sutskever, K. Chen, G. Corrado, and J. Dean (2013). Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 3111–3119. (Cited on page 214.)
- Mikolov, T., I. Sutskever, A. Deoras, H.-S. Le, S. Kombrink, and J. Cernocky (2012). Subword language modeling with neural networks. *preprint* (<http://www.fit.vutbr.cz/~imikolov/rnnlm/char.pdf>). (Cited on pages 199, 206, 224 and 225.)
- Mizutani, E. and J. Demmel (2003). Iterative scaled trust-region learning in Krylov subspaces via Peralmutter’s implicit sparse Hessian-vector multiply. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 209–216. (Cited on page 111.)
- Mizutani, E. and S. Dreyfus (2010). An analysis on negative curvature induced by singularity in multi-layer neural-network learning. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 1669–1677. (Cited on pages 121 and 124.)
- Montúfar, G. and N. Ay (2011). Refinements of universal approximation results for deep belief networks and restricted Boltzmann machines. In *Journal of Neural Computation* 23(5), 1306–1319. (Cited on pages 45 and 212.)
- Montúfar, G. and J. Morton (2012). When does a mixture of products contain a product of mixtures? *Technical Report ArXiv:1206.0387*. (Cited on page 45.)
- Montúfar, G., J. Rauh, and N. Ay (2011). Expressive power and approximation errors of restricted Boltzmann machines. *Advances in Neural Information Processing Systems (NIPS)*, pp. 415–423. (Cited on page 45.)
- Montufar, G. F., R. Pascanu, K. Cho, and Y. Bengio (2014). On the number of

-
- linear regions of deep neural networks. *Technical Report ArXiv:1402.1869*. (Cited on pages [5](#), [46](#) and [81](#).)
- More, J. (1978). The Levenberg-Marquardt algorithm: implementation and theory. In *Numerical analysis*, 105–116. (Cited on page [92](#).)
- Murphy, K. (2012). Machine Learning: A Probabilistic Perspective. *Adaptive computation and machine learning series*. MIT Press. (Cited on page [14](#).)
- Murray, W. (2010). Newton-type methods. *Technical Report*, Department of Management Science and Engineering, Stanford University. (Cited on page [128](#).)
- Nair, V. and G. E. Hinton (2010). Rectified linear units improve restricted Boltzmann machines. In *International Conference on Machine Learning (ICML)*, pp. 807–814. (Cited on page [46](#).)
- Negrello, M. and F. Pasemann (2008). Attractor landscapes and active tracking: The neurodynamics of embodied action. In *Adaptive Behaviour 16*, 196 – 216. (Cited on page [165](#).)
- Nocedal, J. and S. Wright (2006). *Numerical Optimization*. Springer. (Cited on pages [39](#), [40](#), [87](#), [90](#), [92](#), [124](#) and [128](#).)
- Paine, T., H. Jin, J. Yang, Z. Lin, and T. Huang (2014). GPU asynchronous stochastic gradient descent to speed up neural network training. In *International Conference on Learning Representations (ICLR): Workshop Track*. (Cited on page [153](#).)
- Parisi, G. (2007). Mean field theory of spin glasses: statistics and dynamics. *Technical Report ArXiv:0706.0094*. (Cited on page [119](#).)
- Park, H., S.-I. Amari, and K. Fukumizu (2000). Adaptive natural gradient learning algorithms for various stochastic models. In *Neural Networks 13(7)*, 755 – 764. (Cited on pages [41](#), [96](#) and [99](#).)
- Pascanu, R. and Y. Bengio (2014). Revisiting natural gradient for deep networks. In *International Conference on Learning Representations (ICLR): Conference Track*. (Cited on pages [5](#), [32](#), [84](#) and [107](#).)

-
- Pascanu, R., Y. Dauphin, S. Ganguli, and Y. Bengio (2014). On the saddle point problem for non-convex optimization. *Technical Report ArXiv:1405.4604*. (Cited on pages 6 and 84.)
- Pascanu, R., C. Gulcehre, K. Cho, and Y. Bengio (2014). How to construct deep recurrent neural networks. In *International Conference on Learning Representations (ICLR): Conference Track*. (Cited on pages 2, 7, 36, 44, 200 and 211.)
- Pascanu, R. and H. Jaeger (2011). A neurodynamical model for working memory. In *Neural Networks 24*(2), 199–207. (Cited on pages 6, 27, 155, 159, 174, 177 and 190.)
- Pascanu, R., T. Mikolov, and Y. Bengio (2013). On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning (ICML)*. (Cited on pages 6, 7, 36, 155 and 200.)
- Pascanu, R., G. F. Montufar, and Y. Bengio (2014). On the number of response regions of deep feed forward networks with piece-wise linear activations. In *International Conference on Learning Representations (ICLR): Conference Track*. (Cited on pages 4, 46, 49 and 212.)
- Pearlmutter, B. (1994). Fast exact multiplication by the Hessian. In *Journal of Neural Computation 6*(1), 147–160. (Cited on page 88.)
- Peters, J. and S. Schaal (2008). Natural actor-critic. In *Journal of Neurocomputing* (7-9), 1180–1190. (Cited on pages 41 and 96.)
- Peters, J., S. Vijayakumar, and S. Schaal (2003). Policy gradient methods for robot control. *Technical Report*, University of Southern California. (Cited on page 99.)
- Pinheiro, P. and R. Collobert (2014). Recurrent convolutional neural networks for scene labeling. In *International Conference on Machine Learning (ICML)*, pp. 82–90. (Cited on page 217.)
- Poggio, T. and F. Girosi (1989). A theory of networks for approximation and learning. *Technical report 1140. Laboratory, Massachusetts Institute of Technology*. (Cited on page 19.)

-
- Poon, H. and P. Domingos (2011). Sum-product networks: A new deep architecture. In *IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pp. 689–690. (Cited on page 45.)
- Puskorius, G. and L. Feldkamp (1994). Neurocontrol of nonlinear dynamical systems with kalman filter trained recurrent networks. In *IEEE Transactions on Neural Networks* 5(2), 279–297. (Cited on pages 25 and 31.)
- Rabinovich, M. I., R. Huerta, P. Varona, and V. S. Afraimovich (2008). Transient cognitive dynamics, metastability, and decision making. In *PLOS Computational Biology* 4(5). (Cited on page 164.)
- Radicchi, F. and H. Meyer-Ortmanns (2006). Entrainment of coupled oscillators on regular networks by pacemakers. In *Physics Review E* 73. (Cited on page 164.)
- Raiko, T., H. Valpola, and Y. LeCun (2012, April). Deep learning made easier by linear transformations in perceptrons. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 924–932. (Cited on page 217.)
- Ranzato, M., C. Poultney, S. Chopra, and Y. LeCun (2007). Efficient learning of sparse representations with an energy-based model. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 1137–1144. (Cited on page 1.)
- Ratnay, M., D. Saad, and S. I. Amari (1998). Natural Gradient Descent for On-Line Learning. In *Physical Review Letters* 81(24), 5461–5464. (Cited on pages 121 and 124.)
- Reeke, G. N. and G. M. Edelman (1989). Real brains and artificial intelligence. In *The Artificial Intelligence Debate: False Starts, Real Foundations*, pp. 143–173. (Cited on page 1.)
- Rosenblatt, F. (1958). The Perceptron: A probabilistic model for information storage and organization in the brain. In *Psychological Review* 65, 386–408. (Cited on page 15.)
- Rumelhart, D. E., J. L. McClelland, and the PDP Research Group (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Cambridge: MIT Press. (Cited on page 21.)

-
- Saad, D. and S. A. Solla (1995). On-line learning in soft committee machines. In *Physics Review E* 52, 4225–4243. (Cited on page 120.)
- Saxe, A., J. McClelland, and S. Ganguli (2014). Exact solutions to the nonlinear dynamics of learning in deep linear neural network. In *International Conference on Learning Representations (ICLR): Conference Track*. (Cited on pages 120 and 193.)
- Scarselli, F. and A. C. Tsoi (1998, January). Universal approximation using feedforward neural networks: a survey of some existing methods, and some new results. In *Neural Networks 11*, 15–37. (Cited on page 19.)
- Schaul, T. (2012). Natural evolution strategies converge on sphere functions. In *Genetic and Evolutionary Computation Conference (GECCO)*. (Cited on page 98.)
- Schaul, T., I. Antonoglou, and D. Silver (2014). Unit tests for stochastic optimization. In *International Conference on Learning Representations (ICLR): Conference Track*. (Cited on page 116.)
- Schaul, T. and Y. LeCun (2013). Adaptive learning rates and parallelization for stochastic, sparse, non-smooth gradients. In *International Conference on Learning Representations (ICLR): Conference Track*. (Cited on page 39.)
- Schmidhuber, J. (1992). Learning complex, extended sequences using the principle of history compression. In *Journal on Neural Computation* 4(2), 234–242. (Cited on pages 25, 212, 217 and 220.)
- Schmidhuber, J. (2014). Deep learning in neural networks: An overview. *Technical Report ArXiv:1404.7828*. (Cited on page 2.)
- Schmidhuber, J., D. Wierstra, M. Gagliolo, and F. Gomez (2007). Training recurrent networks by evolution. In *Journal on Neural Computation* 19, 2007. (Cited on page 26.)
- Schöner, G., M. Dose, and C. Engels (1995). Dynamics of behavior: theory and applications for autonomous robot architectures. In *Robotics and Autonomous Systems* 16(2), 213–246. (Cited on page 164.)

-
- Schraudolph, N. N. (2001). Fast curvature matrix-vector products. In *International Conference on Artificial Neural Networks (ICANN)*, pp. 19–26. (Cited on pages [101](#), [102](#) and [107](#).)
- Sejnowski, T. J. and C. R. Rosenberg (1988). *NETtalk: a parallel network that learns to read aloud*, pp. 661–672. (Cited on page [24](#).)
- Serre, T., G. Kreiman, M. Kouh, C. Cadieu, U. Knoblich, and T. Poggio (2007). A quantitative theory of immediate visual recognition. In *Progress in Brain Research, Computational Neuroscience: Theoretical Insights into Brain Function 165*, 33–56. (Cited on page [45](#).)
- Shewchuk, J. R. (1994). An introduction to the conjugate gradient method without the agonizing pain. *Technical Report*, Pittsburgh, PA, USA. (Cited on page [92](#).)
- Siegelmann, H. T. (1995). Computation beyond the Turing limit. In *Science 268*(5210), 545–548. (Cited on page [23](#).)
- Siegelmann, H. T. and E. D. Sontag (1991). Turing computability with neural nets. In *Applied Mathematics Letters 4*, 77–80. (Cited on pages [23](#), [24](#) and [207](#).)
- Siegelmann, H. T. and E. D. Sontag (1995). On the computational power of neural nets. In *Journal of Computer and System Sciences 50*(1), 132–150. (Cited on page [23](#).)
- Snoek, J., H. Larochelle, and R. P. Adams (2012). Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems (NIPS)*. (Cited on page [12](#).)
- Sohl-Dickstein, J. (2012). The natural gradient by analogy to signal whitening, and recipes and tricks for its use. *Technical Report ArXiv:1205.1828*. (Cited on page [145](#).)
- Stanley, R. (2004). An introduction to hyperplane arrangements. In *Lecture notes, IAS/Park City Mathematics Institute* (Cited on pages [51](#) and [52](#).)
- Steil, J. J. (2004, 2004). Backpropagation-decorrelation: online recurrent learning with $o(n)$ complexity,. In *International Joint Conference on Neural Networks (IJCNN)*. (Cited on pages [32](#) and [35](#).)

-
- Stollenwerk, N. and F. Pasemann (1996). Control strategies for chaotic neuromodules. *International Journal of Bifurcation and Chaos* 6(4), 693–703. (Cited on page 164.)
- Strogatz, S. (1994). *Nonlinear Dynamics and Chaos: with Applications to Physics, Biology, Chemistry, and Engineering (Studies in Nonlinearity)* (1 ed.). Studies in nonlinearity. Perseus Books Group. (Cited on page 158.)
- Sun, Y., D. Wierstra, T. Schaul, and J. Schmidhuber (2009). Stochastic search using the natural gradient. In *International Conference on Machine Learning (ICML)*. (Cited on pages 41 and 96.)
- Sussillo, D. and O. Barak (2013). Opening the black box: Low-dimensional dynamics in high-dimensional recurrent neural networks. In *Journal of Neural Computation* 25(3), 626–649. (Cited on page 164.)
- Susskind, J., A. Anderson, and G. E. Hinton (2010). The Toronto face dataset. *Technical Report UTML TR 2010-001*, U. Toronto. (Cited on pages 58 and 146.)
- Sutskever, I. and G. E. Hinton (2008). Deep, narrow sigmoid belief networks are universal approximators. In *Journal of Neural Computation* 20(11), 2629–2636. (Cited on page 45.)
- Sutskever, I., J. Martens, G. Dahl, and G. Hinton (2013). On the importance of initialization and momentum in deep learning. In *International Conference on Machine Learning (ICML)*. (Cited on pages 36, 196 and 222.)
- Sutskever, I., J. Martens, and G. Hinton (2011). Generating text with recurrent neural networks. In *International Conference on Machine Learning (ICML)*, pp. 1017–1024. (Cited on pages 36 and 184.)
- Timme, M., F. Wolf, and T. Geisel (2002). Prevalence of unstable attractors in networks of pulse-coupled oscillators. In *Physical Review Letters* 89(15), 154105. (Cited on page 165.)
- Tsuda, I. (2001). Towards an interpretation of dynamic neural activity in terms of chaotic dynamical systems. In *Behavioural and Brain Sciences* 24(5), 793–847. (Cited on page 164.)

-
- Valpola, H. and J. Karhunen (2002). An unsupervised ensemble learning method for nonlinear dynamic state-space models. In *Journal of Neural Computation* 14(11), 2647–2692. (Cited on page 217.)
- van der Maaten, L. and G. E. Hinton (2008). Visualizing data using t-SNE. In *Journal of Machine Learning Research (JMLR)* 9, 2579–2605. (Cited on page 15.)
- Verstraeten, D., B. Schrauwen, and D. Stroobandt (2006). Reservoir-based techniques for speech recognition. In *In Proceedings of the World Conference on Computational Intelligence*. (Cited on page 35.)
- Vinyals, O. and D. Povey (2012). Krylov Subspace Descent for Deep Learning. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*. (Cited on pages 110, 111 and 145.)
- Voegtlin, T. (2002). Recursive self-organizing maps. In *Neural Networks* 15, 979–991. (Cited on page 25.)
- Waibel, A. (1989). Modular construction of time-delay neural networks for speech recognition. In *Journal of Neural Computation* 1, 39–46. (Cited on page 24.)
- Werbos, P. J. (1988). Generalization of backpropagation with application to a recurrent gas market model. In *Neural Networks* 1, 339–356. (Cited on pages 24 and 25.)
- Wigner, E. P. (1958). On the distribution of the roots of certain symmetric matrices. In *The Annals of Mathematics* 67(2), 325–327. (Cited on page 120.)
- Williams, R. J. and J. Peng (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories. In *Journal of Neural Computation* 2, 490–501. (Cited on page 30.)
- Williams, R. J. and D. Zipser (1989). A learning algorithm for continually running fully recurrent neural networks. In *Journal of Neural Computation* 1, 270–280. (Cited on pages 25 and 31.)
- Yao, Y. and W. Freeman (1990). A model of biological pattern recognition with spatially chaotic dynamics. In *Neural Networks* 3(2), 153–170. (Cited on page 164.)

Zaslavsky, T. (1975). Facing Up to Arrangements: Face-Count Formulas for Partitions of Space by Hyperplanes. *in Memoirs of the American Mathematical Society*. (Cited on page 51.)

Zeiler, M. D. and R. Fergus (2013). Visualizing and understanding convolutional networks. *Technical Report ArXiv:1311.2901*. (Cited on pages 2, 44 and 62.)