TRAINING RECURRENT NEURAL NETWORKS

by

Ilya Sutskever

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Training Recurrent Neural Networks

Ilya Sutskever
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2013

Recurrent Neural Networks (RNNs) are powerful sequence models that were believed to be difficult to train, and as a result they were rarely used in machine learning applications. This thesis presents methods that overcome the difficulty of training RNNs, and applications of RNNs to challenging problems.

We first describe a new probabilistic sequence model that combines Restricted Boltzmann Machines and RNNs. The new model is more powerful than similar models while being less difficult to train.

Next, we present a new variant of the Hessian-free (HF) optimizer and show that it can train RNNs on tasks that have extreme long-range temporal dependencies, which were previously considered to be impossibly hard. We then apply HF to character-level language modelling and get excellent results.

We also apply HF to optimal control and obtain RNN control laws that can successfully operate under conditions of delayed feedback and unknown disturbances.

Finally, we describe a random parameter initialization scheme that allows gradient descent with momentum to train RNNs on problems with long-term dependencies. This directly contradicts widespread beliefs about the inability of first-order methods to do so, and suggests that previous attempts at training RNNs failed partly due to flaws in the random initialization.

Acknowledgements

Being a PhD student in the machine learning group of the University of Toronto was lots of fun, and joining it was one of the best decisions that I have ever made. I want to thank my adviser, Geoff Hinton. Geoff taught me how to really do research and our meetings were the highlight of my week. He is an excellent mentor who gave me the freedom and the encouragement to pursue my own ideas and the opportunity to attend many conferences. More importantly, he gave me his unfailing help and support whenever it was needed. I am grateful for having been his student.

I am fortunate to have been a part of such an incredibly fantastic ML group. I truly think so. The atmosphere, faculty, postdocs and students were outstanding in all dimensions, without exaggeration. I want to thank my committee, Radford Neal and Toni Pitassi, in particular for agreeing to read my thesis so quickly. I want to thank Rich for enjoyable conversations and for letting me attend the Z-group meetings.

I want to thank the current learning students and postdocs for making the learning lab such a fun environment: Abdel-Rahman Mohamed, Alex Graves, Alex Krizhevsky, Charlie Tang, Chris Maddison, Danny Tarlow, Emily Denton, George Dahl, James Martens, Jasper Snoek, Maks Volkovs, Navdeep Jaitly, Nitish Srivastava, and Vlad Mnih. I want to thank my officemates, Kevin Swersky, Laurent Charlin, and Tijmen Tieleman for making me look forward to arriving to the office. I also want to thank the former students and postdocs whose time in the group overlapped with mine: Amit Gruber, Andriy Mnih, Hugo Larochelle, Iain Murray, Jim Huang, Inmar Givoni, Nikola Karamanov, Ruslan Salakhutdinov, Ryan P. Adams, and Vinod Nair. It was lots of fun working with Chris Maddison in the summer of 2011. I am deeply indebted to my collaborators: Andriy Mnih, Charlie Tang, Danny Tarlow, George Dahl, Graham Taylor, James Cook, Josh Tenenbaum, Kevin Swersky, Nitish Srivastava, Ruslan Salakhutdinov, Ryan P. Adams, Tim Lillicrap, Tijmen Tieleman, Tomáš Mikolov, and Vinod Nair; and especially to Alex Krizhevsky and James Martens. I am grateful to Danny Tarlow for discovering T&M; to Relu Patrascu for stimulating conversations and for keeping our computers working smoothly; and to Luna Keshwah for her excellent administrative support. I want to thank students in other groups for making school even more enjoyable: Abe Heifets, Aida Nematzadeh, Amin Tootoonchian, Fernando Flores-Mangas, Izhar Wallach, Lena Simine-Nicolin, Libby Barak, Micha Livne, Misko Dzamba, Mohammad Norouzi, Orion Buske, Siavash Kazemian, Siavosh Benabbas, Tasos Zouzias, Varada Kolhatka, Yulia Eskin, Yuval Filmus, and anyone else I might have forgot. A very special thanks goes to Annat Koren for making the writing of the thesis more enjoyable, and for proofreading it.

But most of all, I want to express the deepest gratitude to my family, and especially to my parents, who have done two immigrations for me and my brother's sake. Thank you. And to my brother, for being a good sport.

Contents

0.1 Relationship to Published Work

The chapters in this thesis describe work that has been published in the following conferences and journals:

Chapter 3
- Nonlinear Multilayered Sequence Models
  Ilya Sutskever. Master's Thesis, 2007 (Sutskever, 2007)
- Learning Multilevel Distributed Representations for High-Dimensional Sequences
  Ilya Sutskever and Geoffrey Hinton. In *the Eleventh International Conference on Artificial Intelligence and Statistics* **(AISTATS)**, 2007 (Sutskever and Hinton, 2007)
- The Recurrent Temporal Restricted Boltzmann Machine
  Ilya Sutskever, Geoffrey Hinton and Graham Taylor. In *Advances in Neural Information Processing Systems 21* **(NIPS*21)**, 2008 (Sutskever et al., 2008)

Chapter 4
- Training Recurrent Neural Networks with Hessian Free optimization
  James Martens and **Ilya Sutskever**. In *the 28th Annual International Conference on Machine Learning* **(ICML)**, 2011 (Martens and Sutskever, 2011)

Chapter 5
- Generating Text with Recurrent Neural Networks
  Ilya Sutskever, James Martens, and Geoffrey Hinton. In *the 28th Annual International Conference on Machine Learning* **(ICML)**, 2011 (Sutskever et al., 2011)

Chapter 6
- joint work with Timothy Lillicrap and James Martens

Chapter 7
- joint work with James Martens, George Dahl, and Geoffrey Hinton

The publications below describe work that is loosely related to this thesis but not described in the thesis:

- ImageNet Classification with Deep Convolutional Neural Networks
  Alex Krizhevsky, **Ilya Sutskever**, and Geoffrey Hinton. In *Advances in Neural Information Processing Systems 26*, (**NIPS*26**), 2012. (Krizhevsky et al., 2012)
- Cardinality Restricted Boltzmann Machines
  Kevin Swersky, Danny Tarlow, **Ilya Sutskever**, Richard Zemel, Ruslan Salakhutdinov, and Ryan P. Adams. In *Advances in Neural Information Processing Systems 26*, (**NIPS*26**), 2012 (Swersky et al., 2012)
- Improving neural networks by preventing co-adaptation of feature detectors
  Geoff Hinton, Nitish Srivastava, Alex Krizhevksy, **Ilya Sutskever**, and Ruslan Salakhutdinov. Arxiv, 2012. (Hinton et al., 2012)
- Estimating the Hessian by Backpropagating Curvature
  James Martens, **Ilya Sutskever**, and Kevin Swersky. In *the 29th Annual International Conference on Machine Learning* **(ICML)**, 2012 (Martens et al., 2012)
- Subword language modeling with neural networks
  Tomáš Mikolov , **Ilya Sutskever**, Anoop Deoras, Hai-Son Le, Stefan Kombrink, Jan Černocký. Unpublished, 2012 (Mikolov et al., 2012)
- Data Normalization in the Learning of RBMs
  Yichuan Tang and **Ilya Sutskever**. Technical Report, UTML-TR 2011-02. (Tang and Sutskever, 2011)

- Parallelizable Sampling for MRFs
  James Martens and **Ilya Sutskever**. In *the Thirteenth International Conference on Artificial Intelligence and Statistics* **(AISTATS)**, 2010 (Martens and Sutskever, 2010)

- On the convergence properties of Contrastive Divergence
  Ilya Sutskever and Tijmen Tieleman. In *the Thirteenth International Conference on Artificial Intelligence and Statistics* **(AISTATS)**, 2010 (Sutskever and Tieleman, 2010)

- Modelling Relational Data using Bayesian Clustered Tensor Factorization
  Ilya Sutskever, Ruslan Salakhutdinov, and Joshua Tenenbaum. In *Advances in Neural Information Processing Systems 22* **(NIPS*22)**, 2009 (Sutskever et al., 2009)

- A simpler unified analysis of budget perceptrons
  Ilya Sutskever. In *the 26th Annual International Conference on Machine Learning* **(ICML)**, 2009 (Sutskever, 2009)

- Using matrices to model symbolic relationships
  Ilya Sutskever and Geoffrey Hinton. In *Advances in Neural Information Processing Systems 21* **(NIPS*21)**, 2008 (poster spotlight) (Sutskever and Hinton, 2009b)

- Mimicking Go Experts with Convolutional Neural Networks
  Ilya Sutskever and Vinod Nair. In *the 18th International Conference on Artificial Neural Networks* **(ICANN)**, 2008 (Sutskever and Nair, 2008)

- Deep Narrow Sigmoid Belief Networks are Universal Approximators
  Ilya Sutskever and Geoffrey Hinton, *Neural Computation*. November 2008, Vol 20, No 11: 2629-2636 (Sutskever and Hinton, 2008)

- Visualizing Similarity Data with a Mixture of Maps
  James Cook, **Ilya Sutskever**, Andriy Mnih, and Geoffrey Hinton. In *the Eleventh International Conference on Artificial Intelligence and Statistics* **(AISTATS)**, 2007 (Cook et al., 2007)

- Temporal Kernel Recurrent Neural Networks
  Ilya Sutskever and Geoffrey Hinton, *Neural Networks*, Vol. 23, Issue 2, March 2010, Pages 239-243. (Sutskever and Hinton, 2009a)

Chapter 1

Introduction

Recurrent Neural Networks (RNNs) are artificial neural network models that are well-suited for pattern classification tasks whose inputs and outputs are sequences. The importance of developing methods for mapping sequences to sequences is exemplified by tasks such as speech recognition, speech synthesis, named-entity recognition, language modelling, and machine translation.

An RNN represents a sequence with a high-dimensional vector (called the hidden state) of a fixed dimensionality that incorporates new observations using an intricate nonlinear function. RNNs are highly expressive and can implement arbitrary memory-bounded computation, and as a result, they can likely be configured to achieve nontrivial performance on difficult sequence tasks. However, RNNs have turned out to be difficult to train, especially on problems with complicated long-range temporal structure – precisely the setting where RNNs ought to be most useful. Since their potential has not been realized, methods that address the difficulty of training RNNs are of great importance.

We became interested in RNNs when we sought to extend the Restricted Boltzmann Machine (RBM; Smolensky, 1986), a widely-used density model, to sequences. Doing so was worthwhile because RBMs are not well-suited to sequence data, and at the time RBM-like sequence models did not exist. We introduced the Temporal Restricted Boltzmann Machine (TRBM; Sutskever, 2007; Sutskever and Hinton, 2007) which could model highly complex sequences, but its parameter update required the use of crude approximations, which was unsatisfying. To address this issue, we modified the TRBM and obtained an RNN-RBM hybrid of similar representational power whose parameter update can be computed nearly exactly. This work is described in Chapter 3 and by Sutskever et al. (2008).

Martens (2010)'s recent work on the Hessian-Free (HF) approach to second-order optimization attracted considerable attention, because it solved the then-impossible problem of training deep autoencoders from random initializations (Hinton and Salakhutdinov, 2006; Hinton et al., 2006). Because of its success with deep autoencoders, we hoped that it could also solve the difficult problem of training RNNs on tasks with long-term dependencies. While HF was fairly successful at these tasks, we substantially improved its performance and robustness using a new idea that we call structural damping. It was exciting, because these problems were considered hopelessly difficult for RNNs unless they were augmented with special memory units. This work is described in Chapter 4.

Having seen that HF can successfully train general RNNs, we applied it to character-level language modelling, the task of predicting the next character in natural text (such as in English books; Sutskever et al., 2011). Our RNNs outperform every homogeneous language model, and are the only non-toy language models that can exploit long character contexts. For example, they can balance parentheses and quotes over tens of characters. All other language models (including that of Mahoney, 2005) are fundamentally incapable of doing so because they can only rely on the exact context matches from the training set. Our RNNs were trained with 8 GPUs for 5 days and are among the largest RNNs to date.

This work is presented in Chapter 5.

We then used HF to train RNNs to control a simulated limb under conditions of delayed feedback and unpredictable disturbances (such as a temperature change that introduces friction to the joints) with the goal of solving reaching tasks. RNNs are well-suited for control tasks, and the resulting controller was highly effective. It is described in Chapter 6.

The final chapter shows that a number of strongly-held beliefs about RNNs are incorrect, including many of the beliefs that motivated the research described in the previous chapters. We show that gradient descent with momentum can train RNNs to solve problems with long-term dependencies, provided the RNNs are initialized properly and an appropriate momentum schedule is used. This is surprising because first-order methods were believed to be fundamentally incapable of training RNNs on such problems (Bengio et al., 1994). These results are presented in Chapter 7.

Chapter 2

Background

This chapter provides the necessary background on machine learning and neural networks that will make this thesis relatively self-contained.

2.1 Supervised Learning

Learning is useful whenever we want a computer to perform a function or procedure so intricate that it cannot be programmed by conventional means. For example, it is simply not clear how to directly write a computer program that recognizes speech, even in the absence of time and budget constraints. However, it is in principle straightforward (if expensive) to collect a large number of example speech signals with their annotated content and to use a supervised learning algorithm to approximate the input-output relationship implied by the training examples.

We now define the supervised learning problem. Let X be an input space, Y be an output space, and D be the data distribution over $X \times Y$ that describes the data that we tend to observe. For every draw (x, y) from D, the variable x is a typical input and y is the corresponding (possibly noisy) desired output. The goal of supervised learning is to use a training set consisting of n of i.i.d. samples, $S = \{(x_i, y_i)\}_{i=1}^n \sim D^n$, in order to find a function $f : X \to Y$ whose test error

$$\mathrm{Test}_D(f) \equiv \mathbf{E}_{(x,y)\sim D}[L(f(x); y)] \tag{2.1}$$

is as low as possible. Here $L(z; y)$ is a loss function that measures the loss that we suffer whenever we predict y as z. Once we find a function whose test error is small enough for our needs, the learning problem is solved.

Although it would be ideal to find the global minimizer of the test error

$$f^* = \operatorname*{arg\,min}_{f \text{ is a function}} \mathrm{Test}_D(f) \tag{2.2}$$

doing so is fundamentally impossible. We can approximate the test error with the training error

$$\mathrm{Train}_S(f) \equiv \mathbf{E}_{(x,y)\sim S}[L(f(x); y)] \approx \mathrm{Test}_D(f) \tag{2.3}$$

(where we define S as the uniform distribution over training cases counting duplicate cases multiple times) and find a function f with a low training error, but it is trivial to minimize the training error by memorizing the training cases. Making sure that good performance on the training set translates into good performance on the test set is known as the generalization problem, which turns out to be

conceptually easy to solve by restricting the allowable functions f to a relatively small class of functions $\mathcal{F}$:

$$f^* = \arg\min_{f \in \mathcal{F}} \text{Train}_S(f) \tag{2.4}$$

Restricting f to $\mathcal{F}$ essentially solves the generalization problem, because it can be shown (as we do in sec. 2.6) that when $\log |\mathcal{F}|$ is small relative to the size of the training set (so in particular, $|\mathcal{F}|$ is finite, although similar results can be shown for infinite $\mathcal{F}$'s that are small in a certain sense (Vapnik, 2000)), the training error is close to the test error for all functions $f \in \mathcal{F}$ simultaneously. This lets us focus on the algorithmic problem of minimizing the training error while being reasonably certain that the test error will be approximately minimized as well. The cost of restricting f to $\mathcal{F}$ is that the best attainable test error may be inadequately high for our needs.

Since the necessary size of the training set grows with $\mathcal{F}$, we want $\mathcal{F}$ to be as small as possible. At the same time, we want $\mathcal{F}$ to be as large as possible to improve the performance of its best function. In practice, it is sensible to choose the largest possible $\mathcal{F}$ that can be supported by the size of the training set and the available computation.

Unfortunately, there is no general recipe for choosing a good $\mathcal{F}$ for a given machine learning problem. The theoretically best $\mathcal{F}$ consists of just one function that achieves the best test error among all possible functions, but our ignorance of this function is the reason we are interested in learning in the first place. The more we know about the problem and about its high-performing functions, the more we can restrict $\mathcal{F}$ while being reasonably sure that it contains at least one good function. In practice, it is best to experiment with function classes that are similar to ones that are successful for related problems[1].

2.2 Optimization

Once we have chosen an appropriate $\mathcal{F}$ and collected a sufficiently large training set, we are faced with the problem of finding a function $f \in \mathcal{F}$ that has a low training error. Finding the global minimizer of the training error for most interesting choices of $\mathcal{F}$ is NP-hard, but in practice there are many choices of smoothly-parameterized $\mathcal{F}$s that are relatively easy to optimize with gradient methods.

Let the function $f_\theta \in \mathcal{F}$ be a differentiable parameterization of $\mathcal{F}$ where $\theta \in \mathbb{R}^{|\theta|}$ and $|\theta|$ is the number of parameters. Let us also assume that the loss L is a differentiable function of its arguments. Then the function

$$\text{Train}_S(\theta) \equiv \text{Train}_S(f_\theta) = \mathbf{E}_{(x,y)\sim S}[L(f_\theta(x); y)] \tag{2.5}$$

is differentiable. If $f_\theta(x)$ is easy to compute, then it immediately follows that the training error $\text{Train}_S(\theta)$ and its derivative $\nabla \text{Train}_S(\theta)$ can be computed at the cost of $|S|$ evaluations of $f_\theta(x)$ and $\nabla f_\theta(x)$. In this setting, we can use Gradient Descent (GD), which is a greedy method for minimizing arbitrary differentiable functions. Given a function $F(\theta)$, GD operates as follows:

1: **for** iterations **do**
2: $\quad \theta_{t+1} \leftarrow \theta_t - \nabla F(\theta_t) \cdot \varepsilon$
3: $\quad t \leftarrow t + 1$
4: **end for**

The learning rate ε is a tunable problem-dependent parameter that has a considerable effect on the speed of the optimization.

[1] It is important to evaluate a given $\mathcal{F}$ with its performance on a set examples called the validation set which is distinct from the test set. Otherwise, our choice of $\mathcal{F}$ will be informed by the spurious regularities of the test set, which can cause serious problems when the test set is small.

GD has been extensively analyzed in a number of settings. If the objective function F is a positive definite quadratic, then GD will converge to its global minimum at a rate of

$$F(\theta_t) - F(\theta^*) = O((1 - 1/R)^t) \tag{2.6}$$

where θ^* is the global minimum, R is the condition number of the quadratic (given by the ratio of the largest to the smallest eigenvalues $R = \lambda_{\max}/\lambda_{\min}$), provided that $\varepsilon = 1/\lambda_{\max}$. When F is a general convex function, the rate of convergence can be bounded by

$$F(\theta_t) - F(\theta^*) \leq \frac{\|\theta_1 - \theta_*\|^2}{\varepsilon t} \tag{2.7}$$

provided that $1/\varepsilon$ is greater than the Lipshitz coefficient of ∇F,[2] which in the quadratic case is $\lambda_{\max}$ (Nocedal and Wright, 1999). This rate of convergence is valid across the entire parameter space and not just in the neighborhood of the global minimum, and at first sight appears to be weaker than eq. 2.6 since it is not exponential. However, it is easy to show that when F has a finite condition number, eq. 2.6 is a direct consequnce of eq. 2.7 [3].

Stochastic Gradient Descent (SGD) is an important generalization of GD that is well-suited for machine learning applications. Unlike standard GD, which computes $\nabla \mathrm{Train}_S(\theta)$ on the entire training set S, SGD uses the unbiased approximation $\nabla \mathrm{Train}_s(\theta)$, where s is a randomly chosen subset of the training set S. The "minibatch" s can consist of as little as one training case, but using more training cases is more cost-effective. SGD tends to work better than GD on large datasets where each iteration of GD is very expensive, and for very large datasets it is not uncommon for SGD to converge in the time that it takes batch GD to complete a single parameter update. On the other hand, batch GD is trivially parallelizeable, so it is becoming more attractive due to the availability of large computing clusters.

Momentum methods (Hinton, 1978; Nesterov, 1983) use gradient information to update the parameters in a direction that is more effective than steepest descent by accumulating speed in directions that consistently reduce the cost function. Formally, a momentum method maintains a velocity vector v_t which is updated as follows:

$$v_{t+1} = \mu v_t - \varepsilon \nabla F(\theta_t) \tag{2.8}$$

$$\theta_{t+1} = \theta_t + v_{t+1} \tag{2.9}$$

The momentum decay coefficient $\mu \in [0, 1)$ controls the rate at which old gradients are discarded. Its physical interpretation is the "friction" of the surface of the objective function, and its magnitude has an indirect effect on the magnitude of the velocity.

[2] The Lipshitz coefficient of an arbitrary function $H : \mathbb{R}^m \to \mathbb{R}^k$ is defined as the smallest positive real number L such that $\|H(x) - H(y)\| \leq L\|x - y\|$ for all $x, y \in \mathbb{R}^m$; if no such real number exists, the Lipshitz constant is defined to be infinite. If $L \leq \infty$, the function is continuous.

[3] We can prove an even stronger statement using proof similar to that of O'Donoghue and Candes (2012):

Theorem 2.2.1. *If F is convex, ∇F is L-Lipshitz (so $\|\nabla F(\theta) - \nabla F(\theta')\| \leq L\|\theta - \theta'\|$ for all θ, θ'), and F is σ-strongly convex (so, in particular, $\sigma\|\theta - \theta^*\|^2/2 \leq F(\theta) - F(\theta^*)$ for all θ, where θ^* is the minimum of F), then $F(\theta_t) - F(\theta_*) < (1 - \sigma/6L)^t$.*

Note that when F is quadratic, the above condition implies that its condition number is bounded by L/σ (recall that $\varepsilon = 1/L$).

Proof. The definition of strong convexity gives $\|\theta_1 - \theta_*\|^2 < 2/\sigma(F(\theta_1) - F(\theta_*))$. Applying it into eq. 2.7, we get $F(\theta_t) - F(\theta_*) < 2L/(t\sigma)(F(\theta_1) - F(\theta_*))$. Thus after $t = 4L/\sigma$ iterations, $F(\theta_t) - F(\theta_*) < (F(\theta_1) - F(\theta_*))/2$. Since this bound can be applied at any point, we get that $F(\theta_t) - F(\theta_*)$ is halved every $4L/\sigma$ iterations. Algebraic manipulations imply a convergence rate of $(1 - \sigma/6L)^t$. □

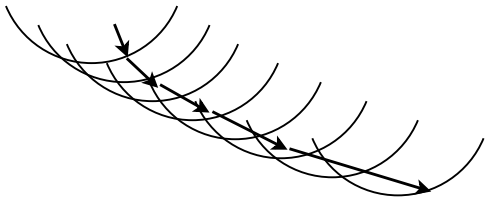


Figure 2.1: A momentum method accumulates velocity in directions of persistent reduction, which speeds up the optimization.

A variant of momentum known as Nesterov's accelerated gradient (Nesterov, 1983, described in detail in Chap. 7) has been analyzed with certain schedules of the learning rate and of the momentum decay coefficient μ, and was shown by Nesterov (1983) to exhibit the following convergence rate for convex functions F whose gradients are noiseless:

$$F(\theta_t) - F(\theta^*) \leq 4\frac{\|\theta_1 - \theta_*\|^2}{\varepsilon(t+2)^2} \tag{2.10}$$

where, as in eq. 2.7, $1/\varepsilon$ is larger than the Lipshitz constant of ∇F. This convergence rate is also global and does not assume that the parameters were initialized in the neighborhood of the global minimum.[4]

Momentum methods are faster than GD because they accelerate the optimization along directions of low but persistent reduction, similarly to the way second-order methods accelerate the optimization along low-curvature directions (but second-order methods also decelerate the optimization along high-curvature directions, which is not done by momentum methods; Nocedal and Wright, 1999) (fig. 2.1). In fact, it has been shown that when Nesterov's accelerated gradient is used with the optimal momentum on a quadratic, its convergence rate is identical to the worst-case convergence rate of the linear conjugate gradient (CG) as a function of the condition number (O'Donoghue and Candes, 2012; Shewchuk, 1994; Nocedal and Wright, 1999). This may be surprising, since CG is the optimal iterative method for quadratics (in the sense that it outperforms any method that uses linear combinations of previously-computed gradients; although CG can be obtained from eqs. 2.8-2.9 using a certain formula for μ (Shewchuk, 1994)). Thus momentum can be seen as a second-order method that accelerates the optimization in directions of low-curvature. They can also decelerate the optimization along the high-curvature directions by cancelling the high-frequency oscillations that cause GD to diverge.

Convex objective functions $F(\theta)$ are insensitive to the initial parameter setting, since the optimization will always recover the optimal solution, merely taking longer time for worse initializations. But, given that most objective functions $F(\theta)$ that we want to optimize are non-convex, the initialization has a profound impact on the optimization and on the quality of the solution. Chapter 7 shows that appropriate initializations play an even greater role than previously believed for both deep and recurrent neural networks. Unfortunately, it is difficult to design good random initializations for new models, so it is important to experiment with many different initializations. In particular, the scale of the initialization tends to have a large influence for neural networks (see Chap. 7 and Jaeger and Haas (2004); Jaeger (2012b)). For deep neural networks (sec. 2.4), the greedy unsupervised pre-training of Hinton et al. (2006), Hinton and Salakhutdinov (2006), and Bengio et al. (2007) is an effective technique for initializing the parameters, which greedily trains parameters of each layer to model the distribution of

[4] An argument similar to the one in footnote 3 can show that the convergence rate of a quadratic with condition number R is $(1 - \sqrt{1/R})^t$, provided that the momentum is reset to zero every $\sqrt{R}$ iterations. See O'Donoghue and Candes (2012) for more details. Note that it is also the worst-case convergence rate of the linear conjugate gradient (Shewchuk, 1994) as a function of the condition number.

activities in the layer below.

2.3 Computing Derivatives

In this section, we explain how to efficiently compute the derivatives of any function $F(\theta)$ that can be evaluated with a differentiable computational graph (Baur and Strassen, 1983; Nocedal and Wright, 1999). In this section, the term "input" refers to the parameters rather than to an input pattern, because we are interested in computing derivatives w.r.t. the parameters.

Consider a graph over N nodes, $1, \ldots, N$. Let $\mathcal{I}$ be the set of the input nodes, and let the last node N be the output node (the formalism allows for $N \in \mathcal{I}$, but this makes for a trivial graph). Each node i has a set of ancestors A_i (with numbers less than i) that determine its inputs, and a differentiable function f_i whose value and Jacobian are easy to compute. Then the following algorithm evaluates a computational graph:

1: distribute the input θ across the input nodes z_i for $i \in \mathcal{I}$
2: **for** i **from** 1 **to** N **if** $i \notin \mathcal{I}$ **do**
3: $\quad x_i \leftarrow \underset{j \in A_i}{\text{concat}}\, z_j$
4: $\quad z_i \leftarrow f_i(x_i)$
5: **end for**
6: **output** $F(\theta) = z_N$

where every node z_i can be vector-valued (this includes the output node $i = N$, so F can be vector-valued). Thus the computational graph formalism captures nearly all models that occur in machine learning.

If we assume that our training error has the form $L(F(\theta)) = L(z_N)$ where L is the loss function and $F(\theta)$ is the vector of the model's predictions on all the training cases, the derivative of $L(z_N)$ w.r.t. θ is given by $F'(\theta)^\top L'(z_N)$. We now show how the backpropagation algorithm computes the Jacobian-vector product $F'(\theta)^\top w$ for an arbitrary vector w of the dimensionality of z_N:

1: $dz_N \leftarrow w$
2: $dz_i \leftarrow 0$ for $i < N$
3: **for** i **from** N **downto** 1 **if** $i \notin \mathcal{I}$ **do**
4: $\quad dx_i \leftarrow f_i'(x_i)^\top \cdot dz_i$
5: $\quad$ **for all** $j \in A_i$ **do**
6: $\quad\quad dz_j \leftarrow dz_j + \text{unconcat}^j dx_i$
7: $\quad$ **end for**
8: **end for**
9: concatenate dz_i for $i \in \mathcal{I}$ onto $d\theta$
10: **output** $d\theta$, which is equal to $F'(\theta)^\top w$

where unconcat is the inverse of concat: if $x_i = \text{concat}_{j \in A_i} z_j$, then $\text{unconcat}^j x_i = z_j$. The correctness of the above algorithm can be proven by structural induction over the graph, which would show that each node dz_i is equal to $\frac{\partial z_N}{\partial z_i}^\top w$ as we descend from the node dz_N, where the induction step is proven with the chain rule. By setting w to $L'(z_N)$, the algorithm computes the sought-after derivative.

A different form of differentiation known as forward differentiation computes the derivative of each z_i w.r.t. a linear combination of the parameters. Given a vector v (whose dimensionality matches θ's), we let Rz_i be the directional derivative $\partial z_i / \partial \theta \cdot v$ (the notation is from Pearlmutter (1994)). Then the following algorithm computes the directional derivative of each node in the graph as follows:

1: distribute the input v across the input nodes Rz_i for $i \in \mathcal{I}$
2: **for** i **from** 1 **to** N **if** $i \notin \mathcal{I}$ **do**
3: $\quad Rx_i \leftarrow \operatorname{concat}_{j \in A_i} Rz_j$
4: $\quad Rz_i \leftarrow f_i'(x_i) \cdot Rx_i$
5: **end for**
6: **output** Rz_N, which is equal to $F'(\theta) \cdot v$

The correctness of this algorithm can likewise be proven by structural induction over the graph, proving that $Rz_i = \partial z_i / \partial \theta \cdot v$ for each i, starting from $i \in \mathcal{I}$ and reaching node z_N. Unlike backward differentiation, forward differentiation does not need to store the state variables z_i, since they can be computed together with Rz_i and be discarded once used.

Thus automatic differentiation can compute the Jacobian-vector products of any function expressible as a computational graph at the cost of roughly two function evaluations. The Theano compiler (Bergstra et al., 2010) computes derivatives in precisely this manner automatically and efficiently.

2.4 Feedforward Neural Networks

The Feedforward Neural Network (FNN) is the most basic and widely used artificial neural network. It consists of a number of layers of artificial neurons (termed units) that are arranged into a layered configuration (fig. 2.2). Of particular interest are deep neural networks, which are believed to be capable of representing the highly complex functions that achieve high performance on difficult perceptual problems such as vision and speech (Bengio, 2009). FNNs have achieved success in a number of domains (e.g., Salakhutdinov and Hinton, 2009; Glorot et al., 2011; Krizhevsky and Hinton, 2011; Krizhevsky, 2010), most notably in large vocabulary continuous speech recognition (Mohamed et al., 2012), where they were directly responsible for considerable improvements over previous highly-tuned, state-of-the-art systems.

Formally, a feedforward neural network with ℓ hidden layers is parameterized by $\ell + 1$ weight matrices $(W_0, \ldots, W_\ell)$ and $\ell + 1$ vectors of biases $(b_1, \ldots, b_{\ell+1})$. The concatenation of the weight matrices and the biases forms the parameter vector θ that fully specifies the function computed by the network. Given an input x, the feedforward neural network computes an output z as follows:

1: set $z_0 \leftarrow x$
2: **for** i **from** 1 **to** $\ell + 1$ **do**
3: $\quad x_i \leftarrow W_{i-1} z_{i-1} + b_i$
4: $\quad z_i \leftarrow e(x_i)$
5: **end for**
6: **Output** $z \leftarrow z_\ell$

Here $e(\cdot)$ is some nonlinear function such as the element-wise sigmoid $\operatorname{sigmoid}(x_j) = 1/(1+\exp(-x_j))$.

When $e(\cdot)$ is a sigmoid function, it can be used to implement Boolean logic, which allows deep and wide feedforward neural networks to implement arbitrary Boolean circuits and hence arbitrary computation subject to some time and space constraints. But while we cannot hope to always train feedforward neural networks so deep from labelled cases alone (provably so, under certain cryptographic assumptions (Kearns and Vazirani, 1994)[5]), wide and slightly deep networks (3-5 layers) are often easy to train in practice with SGD while being moderately expressive.

[5]Intuitively, given a public key, it is easy to generate a large number of (encryption, message) pairs. If circuits were learnable, we could learn a circuit that could map an encryption to its secret message with high accuracy (since such a circuit exists).

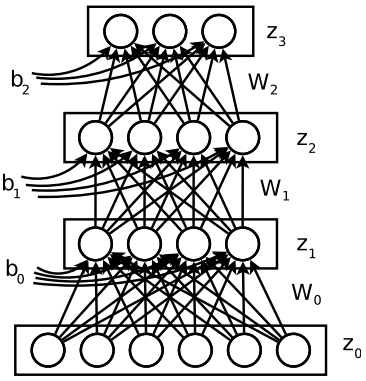


Figure 2.2: The feedforward neural network.

FNNs are trained by minimizing the training error w.r.t. the parameters using a gradient method, such as SGD or momentum.

Despite their representational power, deep FNNs have been historically considered very hard to train, and until recently have not enjoyed widespread use. They became the subject of intense attention thanks to the work of Hinton and Salakhutdinov (2006) and Hinton et al. (2006), who introduced the idea of greedy layerwise pre-training, and successfully applied deep FNNs to a number of challenging tasks. Greedy layerwise pre-training has since branched into a family of methods (Hinton et al., 2006; Hinton and Salakhutdinov, 2006; Bengio et al., 2007), all of which train the layers of a deep FNN in order, one at time, using an auxiliary objective, and then "fine-tune" the network with standard optimization methods such as stochastic gradient descent. More recently, Martens (2010) has attracted considerable attention by showing that a type of truncated-Newton method called Hessian-free optimization (HF) is capable of training deep FNNs from certain random initializations without the use of pre-training, and can achieve lower errors for the various auto-encoding tasks considered in Hinton and Salakhutdinov (2006). But recent results described in Chapter 7 show that even very deep neural networks can be trained using an aggressive momentum schedule from well-chosen random initializations.

It is possible to implement the FNN with the computational graph formalism and to use backward automatic differentiation to obtain the gradient (which is done if the FNN is implemented in Theano (Bergstra et al., 2010)), but it is also straightforward to program the gradient directly:

$dz_\ell \leftarrow dL(z_\ell; y)/dz_\ell$
for i **from** $\ell + 1$ **downto** 1 **do**
  $dx_i \leftarrow e'(x_i) \cdot dz_i$
  $dz_{i-1} \leftarrow W_{i-1}^\top dx_i$
  $db_i \leftarrow dx_i$
  $dW_{i-1} \leftarrow dx_i z_{i-1}^\top$
end for
Output $[dW_0, \ldots, dW_\ell, db_1, \ldots, db_{\ell+1}]$

2.5 Recurrent Neural Networks

We are now ready to define the Recurrent Neural Network (RNN), the central object of study of this thesis. The standard RNN is a nonlinear dynamical system that maps sequences to sequences. It is parameterized with three weight matrices and three bias vectors $[W_{hv}, W_{hh}, W_{oh}, b_h, b_o, h_0]$ whose con-

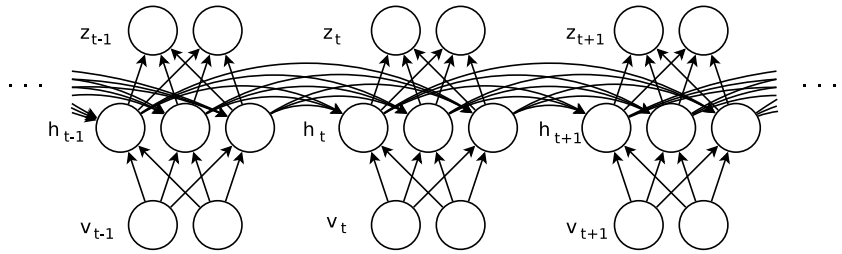


Figure 2.3: A Recurrent Neural Network is a very deep feedforward neural network that has a layer for each timestep. Its weights are shared across time.

catenation θ completely describes the RNN (fig. 2.3). Given an input sequence $(v_1, \ldots, v_T)$ (which we denote by v_1^T), the RNN computes a sequence of hidden states h_1^T and a sequence of outputs z_1^T by the following algorithm:

1: **for** t **from** 1 **to** T **do**
2: $u_t \leftarrow W_{hv} v_t + W_{hh} h_{t-1} + b_h$
3: $h_t \leftarrow e(u_t)$
4: $o_t \leftarrow W_{oh} h_t + b_o$
5: $z_t \leftarrow g(o_t)$
6: **end for**

where $e(\cdot)$ and $g(\cdot)$ are the hidden and output nonlinearities of the RNN, and h_0 is a vector of parameters that store the very first hidden state. The loss of the RNN is usually a sum of per-timestep losses:

$$L(z, y) = \sum_{t=1}^{T} L(z_t; y_t) \tag{2.11}$$

The derivatives of the RNNs are easily computed with the backpropagation through time algorithm (BPTT; Werbos, 1990; Rumelhart et al., 1986):

1: **for** t **from** T **downto** 1 **do**
2: $do_t \leftarrow g'(o_t) \cdot dL(z_t; y_t)/dz_t$
3: $db_o \leftarrow db_o + do_t$
4: $dW_{oh} \leftarrow dW_{oh} + do_t h_t^\top$
5: $dh_t \leftarrow dh_t + W_{oh}^\top do_t$
6: $dz_t \leftarrow e'(z_t) \cdot dh_t$
7: $dW_{hv} \leftarrow dW_{hv} + dz_t v_t^\top$
8: $db_h \leftarrow db_h + dz_t$
9: $dW_{hh} \leftarrow dW_{hh} + dz_t h_{t-1}^\top$
10: $dh_{t-1} \leftarrow W_{hh}^\top dz_t$
11: **end for**
12: **Return** $d\theta = [dW_{hv}, dW_{hh}, dW_{oh}, db_h, db_o, dh_0]$.

2.5.1 The difficulty of training RNNs

Although the gradients of the RNN are easy to compute, RNNs are fundamentally difficult to train, especially on problems with long-range temporal dependencies (Bengio et al., 1994; Martens and Sutskever, 2011; Hochreiter and Schmidhuber, 1997), due to their nonlinear iterative nature. A small change to an iterative process can compound and result in very large effects many iterations later; this is known

colloquially as "the butterfly-effect". The implication is that in an RNN, the derivative of the loss function at one time can be exponentially large with respect to the hidden activations at a much earlier time. Thus the loss function is very sensitive to small changes, so it becomes effectively discontinuous.

RNNs also suffer from the vanishing gradient problem, first described by Hochreiter (1991) and Bengio et al. (1994). Consider the term $\frac{\partial L(z_T; y_T)}{\partial W_{hh}}$, which is easy to analyze by inspecting line 10 of the BPTT algorithm:

$$\frac{\partial L(z_T; y_T)}{\partial W_{hh}} = \sum_{t=1}^{T} dz_t h_{t-1}^\top \tag{2.12}$$

where

$$dz_t = \left(\prod_{\tau=t+1}^{T} W_{hh}^\top e'(z_t) \right) \left(W_{oh}^\top g'(o_t) \frac{\partial L(z_T; y_T)}{\partial p_T} \right) \tag{2.13}$$

If all the eigenvalues of W_{hh} are considerably smaller than 1, then the contributions $dz_t h_{t-1}^\top$ to dW_{hh} will rapidly diminish because dz_t tends to zero exponentially as $T - t$ increases. The latter phenomenon is known as the vanishing gradient problem, and it is guaranteed to occur in any RNN that can store one bit of information indefinitely while being robust to at least some level of noise (Bengio et al., 1994), a condition that ought to be satisfied by most RNNs that perform interesting computation. A vanishing dz_t is undesirable, because it turns BPTT into truncated-BPTT, which is likely incapable of training RNNs to exploit long-term temporal structure (see sec. 2.8.6).

The vanishing and the exploding gradient problems make it difficult to optimize RNNs on sequences with long-range temporal dependencies, and are possible causes for the abandonment of RNNs by machine learning researchers.

2.5.2 Recurrent Neural Networks as Generative models

Generative models are parameterized families of probability distributions that extrapolate a finite training set to a distribution over the entire space. They have many uses: good generative models of spectrograms can be used to synthesize speech; generative models of natural language can improve speech recognition by deciding between words that the acoustic model cannot accurately distinguish; and improve machine translation by evaluating the plausibility of a large number of candidate translations in order to select the best one.

An RNN defines a generative model over sequences if the loss function satisfies $L(z_t; y_t) = -\log p(y_t; z_t)$ for some parameterized family of distributions $p(\cdot; z)$ and if $y_t = v_{t+1}$. This defines the following distribution over sequences v_1^T:

$$P(v_t | v_1^{t-1}) \equiv P(v_t; z_t) \tag{2.14}$$

$$P(v_1^T) = \prod_{t=1}^{T} P(v_t | v_1^{t-1}) \equiv \prod_{t=1}^{T} P(v_t; z_{t-1}) \tag{2.15}$$

where z_0 is an additional parameter vector that specifies the distribution over the first timestep of the sequence, v_1. This equation defines a valid distribution because z_t is a function of v_1^t and is independent of v_{t+1}^T. Samples from $P(v_1^T)$ can be obtained by sampling the conditional distribution $v_t \sim P(v_t | v_1^{t-1})$ sequentially, by iterating from $t = 1$ to T. The loss function is precisely equivalent to the average negative log probability of the sequences in the training set:

$$\text{Train}_S(\theta) = \mathbf{E}_{v \sim S} \left[-\log P_\theta(v) \right] \tag{2.16}$$

where in this equation, the dependence of P on the parameters θ is explicit.

The log probability is a good objective function to optimize if we wish to fit a distribution to data. Assuming the data distribution D is precisely equal to P_{θ^*} for some θ^* and that the mapping $\theta \to P_\theta$ is one-to-one, it is meaningful to discuss the rate at which our parameter estimates converge to θ^* as the size of the training set increases. In this setting, if the log probability is uniformly bounded across θ, the maximum likelihood estimator (i.e., the parameter setting maximizing eq. 2.16) is known to have the fastest possible rate of convergence among all possible ways of mapping data to parameters (in a minimax sense over the parameters) when the size of the training set is sufficiently large (Wasserman, 2004), which justifies its use. In practice, the true data distribution D cannot usually be represented by any setting of the parameters in the model P_{θ^*}, but the average log probability objective is still used.

2.6 Overfitting

The term overfitting refers to the gap between the training error $\mathrm{Train}_S(f)$ and the test error $\mathrm{Test}_D(f)$. We mentioned earlier that by restricting the functions of consideration f to a class of functions $\mathcal{F}$ we control overfitting and address the generalization problem. Here we explain how limiting $\mathcal{F}$ accomplishes this.

Theorem 2.6.1. *If $\mathcal{F}$ is finite and the loss is bounded $L(z; y) \in [0, 1]$, then overfitting is uniformly bounded with high probability over draws S from the training set (Kearns and Vazirani, 1994; Valiant, 1984):*

$$\mathbf{Pr}_{S\sim D^{|S|}}\left[\mathrm{Test}_D(f) - \mathrm{Train}_S(f) \le \sqrt{\frac{\log|\mathcal{F}| + \log 1/\delta}{|S|}} \;\; \textit{for all } f \in \mathcal{F}\right] > 1 - \delta \tag{2.17}$$

The result suggests that when the training set is larger than $\log|\mathcal{F}|$, then every training error will be close to every test error. The $\log|\mathcal{F}|$ term in the bound formally justifies the intuition that each training case carries a constant number of bits about the best function in $\mathcal{F}$.

Proof. We begin with the proof's intuition. The central limit theorem ensures that the training error $\mathrm{Train}_S(f)$ is centred at $\mathrm{Test}_D(f)$ and has Gaussian tails of size $1/\sqrt{|S|}$ (here we rely on $L(z; y) \in [0, 1]$). This means that, for a single function, the probability that its training error deviates from its test error by more than $\log|\mathcal{F}|/\sqrt{|S|}$ is exponentially small in $\log|\mathcal{F}|$. Hence the training error is *fairly* unlikely to deviate by more than $\log|\mathcal{F}|/\sqrt{|S|}$ for $|\mathcal{F}|$ functions simultaneously.

This intuition can be quantified by means of a one-sided Chernoff or Hoffeding bound for a single fixed f (Lugosi, 2004; Kearns and Vazirani, 1994; Valiant, 1984):

$$\mathbf{Pr}_{S\sim D^{|S|}}\left[\mathrm{Test}_D(f) - \mathrm{Train}_S(f) > t\right] \le \exp(-|S|t^2) \tag{2.18}$$

Applying the union bound, we can immediately control the maximal possible overfitting:

$$\begin{aligned}
\mathbf{Pr}\left[\mathrm{Test}_D(f) - \mathrm{Train}_S(f) > t \text{ for some } f \in \mathcal{F}\right] &= \mathbf{Pr}\left[\cup_{f\in\mathcal{F}}\{\mathrm{Test}_D(f) - \mathrm{Train}_S(f) > t\}\right] \\
&\le \sum_{f\in\mathcal{F}} \mathbf{Pr}\left[\mathrm{Test}_D(f) - \mathrm{Train}_S(f) > t\right] \\
&\le |\mathcal{F}|\exp(-|S|t^2)
\end{aligned}$$

If we set the probability of failure $\mathbf{Pr}\left[\mathrm{Test}_D(f) - \mathrm{Train}_S(f) > t \text{ for some } f \in \mathcal{F}\right]$ to δ, we can obtain an upper bound for t:

$$\begin{aligned} \delta &\leq |\mathcal{F}| \exp(-|S|t^2) \\ \log(\delta) - \log|\mathcal{F}| &\leq -|S|t^2 \\ \log\frac{1}{\delta} + \log|\mathcal{F}| &\geq |S|t^2 \\ t &\leq \sqrt{\frac{\log 1/\delta + \log|\mathcal{F}|}{|S|}} \end{aligned}$$

The above states that with probability at least $1-\delta$, there is no function $f \in \mathcal{F}$ whose overfitting exceeds $\sqrt{\frac{\log 1/\delta + \log|\mathcal{F}|}{|S|}}$. □

This is one of the simplest learning theoretic bounds (Kearns and Vazirani, 1994), and although formally it is only applicable to finite $\mathcal{F}$, we observe that it trivially applies to any implementation of any parametric model that uses the common 32 or 64-bit floating point representation for its parameters. These implementations work with finite subsets of $\mathcal{F}$ that have at most $2^{64|\theta|}$ different elements (which can be RNNs, FNNs, or other models), so the theorem applies and guarantees that such implementations will not overfit if the number of labelled cases exceeds the number of its parameters by some constant factor. Although this bound is pessimistic, it provides a formal justification for a heuristic practice called "parameter counting", where number of parameters is compared to the number of labels in the training set in order to predict the severity of overfitting.

It should be emphasized that this theorem does not suggest that learning will necessarily succeed, since it could fail if the training error of each $f \in \mathcal{F}$ is unacceptably high. It is therefore important for $\mathcal{F}$ to be well-chosen so that at least one of its functions achieves low training (and hence test) error.

It should also be emphasized that this result does not mean that overfitting must occur when $|S| \ll |\theta|$. In these cases, generalization may occur for other reasons, such as the inability of the optimization method to fully exploit the neural network's capacity, or the relative simplicity of the dataset.

2.6.1 Regularization

There are several other techniques for preventing overfitting. A common technique is regularization, which replaces the training error $\mathrm{Train}_S(\theta)$ with $\mathrm{Train}_S(\theta) + \lambda R(\theta)$, where $R(\cdot)$ is a function that penalizes "overly complex" models (as determined by some property of θ). A common choice for $R(\theta)$ is $\|\theta\|^2/2$.

While the method of regularization seems to be different from working with subsets of $\mathcal{F}$, it turns out to be closely related to the use of the smaller function class $\mathcal{F}_\lambda = \{f_\theta : R(\theta) < \lambda\}$ and is equivalent when $\mathrm{Train}_S(\theta)$ and $R(\theta)$ are convex functions of θ.

Theorem 2.6.2. *Assume that* $\mathrm{Train}_S(\theta)$ *and* $R(\theta)$ *are smooth. Then for every local minimum* θ^* *of*

$$\mathrm{Train}_S(\theta) \quad \textit{s.t.} \quad R(\theta) \leq \lambda \tag{2.19}$$

there exists a λ' *such that* θ^* *is a local minimum of*

$$\mathrm{Train}_S(\theta) + \lambda' R(\theta) \tag{2.20}$$

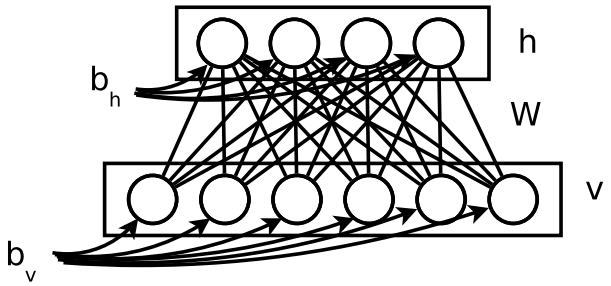


Figure 2.4: A Restricted Boltzmann Machine.

If $\mathrm{Train}_S(\theta)$ and $R(\theta)$ are convex functions of θ, then every local minimum is also global minimum, and as a result, the method of regularization and restriction are equivalent up to the choice of λ. Their equivalence breaks down in the presence of multiple local minima.

Proof. Let λ be given, and let θ^* be a local minimum of eq. 2.19. Then either $R(\theta^*) < \lambda$ or $R(\theta^*) = \lambda$.

If $R(\theta^*) < \lambda$, then the choice $\lambda' = 0$ makes θ^* a local minimum of eq. 2.20.

When $R(\theta^*) = \lambda$, consider the Lagrange function for the problem of minimizing

$$\mathrm{Train}_S(\theta) \quad \text{s.t. } R(\theta) = \lambda \tag{2.21}$$

which is given by

$$L(\theta, \mu) = \mathrm{Train}_S(\theta) + \mu \cdot (R(\theta) - \lambda) \tag{2.22}$$

A basic property of Lagrange functions states that if θ^* is a local minimum of eq. 2.21, then there exists μ^* such that

$$\nabla_\theta L(\theta^*, \mu^*) = 0 \tag{2.23}$$

Eq. 2.23 implies that $\nabla_\theta \mathrm{Train}_S(\theta^*) + \mu^* \nabla_\theta R(\theta^*) = 0$, so θ^* is a local minimum of the regularized objective $\mathrm{Train}_S(\theta) + \mu^* R(\theta)$ as well. □

The converse (that for every local minimum θ^* of $\mathrm{Train}_S(\theta) + \lambda R(\theta)$ there exists a λ' such that $\theta*$ is a local minimum of $\mathrm{Train}_S(\theta)$ subject to $R(\theta) < \lambda'$) can be proved with a similar method.

2.7 Restricted Boltzmann Machines

The Restricted Boltzmann Machine, or RBM, is a parameterized family of probability distributions over binary vectors. It defines a joint distribution over $v \in \{0,1\}^{N_v}$ and $h \in \{0,1\}^{N_h}$ via the following equation (fig. 2.4):

$$P(v,h) \equiv P_\theta(v,h) \equiv \frac{\exp\left(h^\top W v + v^\top b_v + h^\top b_h\right)}{Z(\theta)} \tag{2.24}$$

where the partition function $Z(\theta)$ is given by

$$Z(\theta) = \sum_{v \in \{0,1\}^{N_v}} \sum_{h \in \{0,1\}^{N_h}} \exp\left(h^\top W v + v^\top b_v + h^\top b_h\right) \tag{2.25}$$

Here $\theta = [W, b_v, b_h]$ is the parameter vector. Thus each setting of θ has a corresponding RBM distribution P_θ.

The partition function $Z(\theta)$ is an sum of exponentially many terms and cannot be efficiently approximated to a constant multiplicative factor unless $P = NP$ (Long and Servedio, 2010). This makes the RBM difficult to handle, because we cannot evaluate the RBM's objective function and measure the progress of learning. Nevertheless, it enjoys considerable popularity despite its intractability for two reason: first, the RBM can learn excellent generative models, and its samples often "look like" the training (and test) data (Hinton, 2002); and second, the RBM plays an important role in the training of Deep Belief Networks (Hinton et al., 2006; Hinton and Salakhutdinov, 2006), by acting as a good initialization for the FNN. The ease of posterior inference is another attractive feature since the distributions $P(h|v)$ and $P(v|h)$ are product distributions (or *factorial* distributions) and have the following simple form (recall that $\mathrm{sigmoid}(x) = 1/(1+\exp(-x))$):

$$P(h_i = 1|v) = \mathrm{sigmoid}((Wv + b_h)_i) \tag{2.26}$$
$$P(v_j = 1|h) = \mathrm{sigmoid}((W^\top h + b_v)_j) \tag{2.27}$$

To derive these equations, let $t = Wv + b_h$ and $c = \dfrac{1}{Z(\theta)P(v)} \exp(v^\top b_v)$:

$$P(h|v) = P(h,v)/P(v) \tag{2.28}$$
$$= \exp\left(h^\top (Wv + b_h) + v^\top b_v\right) \frac{1}{Z(\theta)P(v)} = \exp\left(h^\top t\right) \cdot c \tag{2.29}$$
$$= \exp\left(\sum_{i=1}^{N_h} h_i \cdot t_i\right) \cdot c = \prod_{i=1}^{N_h} \exp(h_i t_i) \cdot c \tag{2.30}$$

Thus $P(h|v)$ is a product distribution. By treating eq. 2.30 as a function of $h_i \in \{0,1\}$ and normalizing it so that it sums to 1 over its domain $\{0,1\}$, we get

$$\begin{aligned} P(h_i|v) &= \frac{\exp(t_i \cdot h_i) \cdot c}{\exp(t_i \cdot 0) \cdot c + \exp(t_i \cdot 1) \cdot c} = \frac{1}{\exp(t_i \cdot (0 - h_i)) + \exp(t_i \cdot (1 - h_i))} \\ &= \frac{1}{1 + \exp(t_i \cdot ([\![h_i = 0]\!] - [\![h_i = 1]\!]))} = \mathrm{sigmoid}(t_i \cdot (2h_i - 1)) \\ &= \mathrm{sigmoid}((Wv + b_h)_i \cdot (2h_i - 1)) \end{aligned} \tag{2.31}$$

where $[\![X]\!]$ is 1 if X is true and 0 otherwise. A similar derivation applies to the other conditional distribution.

As a generative model, the RBM can be trained by maximizing the average log probability of a training set S:

$$\mathrm{Train}_S(\theta) = \mathbf{E}_{v \sim S}[\log P_\theta(v)] \tag{2.32}$$

We will maximize the RBM's average log probability with a gradient method, so we need to compute the derivative w.r.t. the RBM's parameters. Letting $G(v,h) = h^\top W v + b_v^\top v + b_h^\top h$ and noticing that

$P(v,h) = \exp(G(v,h))/Z(\theta)$, we can compute the derivatives of $\mathrm{Train}_S(\theta)$ w.r.t. W as follows:

$$\begin{aligned}
\nabla_W \mathrm{Train}_s(\theta) &= \mathbf{E}_{v\sim S}[\nabla_W \log P(v)] = \mathbf{E}_{v\sim S}\left[\nabla_W \log \sum_h P(v,h)\right] \\
&= \mathbf{E}_{v\sim S}\left[\nabla_W \log \sum_h \exp(G(v,h)) - \nabla_W \log Z(\theta)\right] \\
&= \mathbf{E}_{v\sim S}\left[\frac{\nabla_W \sum_h \exp(G(v,h))}{\sum_{h'} \exp(G(v,h'))}\right] - \nabla_W \log \sum_{v,h} \exp(G(v,h)) \\
&= \mathbf{E}_{v\sim S}\left[\sum_h \frac{\exp(G(v,h))\nabla_W G(v,h)}{\sum_{h'} \exp(G(v,h'))}\right] - \frac{\sum_{v,h} \nabla_W \exp(G(v,h))}{\sum_{v',h'} \exp(G(v',h'))} \\
&= \mathbf{E}_{v\sim S}\left[\sum_h P(h|v)\nabla_W G(v,h)\right] - \sum_{v,h} \frac{\exp(G(v,h))\nabla_W G(v,h)}{\sum_{v',h'} \exp(G(v',h'))} \\
&= \mathbf{E}_{v\sim S}\left[\mathbf{E}_{h\sim P(\cdot|v)}[\nabla_W G(v,h)]\right] - \sum_{v,h} P(v,h)\nabla_W G(v,h) \\
&= \mathbf{E}_{v\sim S, h\sim P(\cdot|v)}[\nabla_W G(v,h)] - \mathbf{E}_{v,h\sim P}[\nabla_W G(v,h)]
\end{aligned}$$

We finish by noticing that $\nabla_W G(v,h) = hv^\top$. An identical derivation yields the derivatives w.r.t. the biases, where we use $\nabla_{b_h} G(v,h) = h$ and $\nabla_{b_v} G(v,h) = v$.

These derivatives consist of expectations that can be approximated using unbiased samples from $S(v)P(h|v)$ and $P(v,h)$. Unsurprisingly, it is difficult to do, because the objective can be estimated by an integral of unbiased derivative estimates. Indeed (assuming $P \neq \#P$), there is no efficient method for drawing approximately-unbiased samples from $P(v,h)$ that is valid for every setting of the RBM's parameters (Long and Servedio, 2010).

Nonetheless, it is possible to train RBMs reasonably well with Contrastive Divergence (CD). CD is an approximate parameter update (Hinton, 2002) that works well in practice. CD computes a parameter update by replacing the model distribution $P(h|v)P(v)$, which is difficult to sample, with the distribution $P(h|v)R^k(v)$ which is easy to sample. $R^k(v)$ is sampled by running a Markov chain that is initialized to the empirical data distribution S and is followed by k-steps of some transition operator T that converges to the distribution $P(v)$:

1: randomly pick v_0 from the training set S
2: **for** i **from** 1 **to** k **do**
3: $\quad v_i \sim T(\cdot|v_{i-1})$
4: **end for**
5: **return** v_k

The above algorithm computes a sample from R^k. The value of k is arbitrary, since k steps of a transition operator $T(\cdot|v)$ can be combined to one step of the transition operator $T^k(\cdot|v)$. However, it is customary to use the term "CD$_k$" to a CD update that is based on k steps of the above algorithm with some simple T.

As $k \to \infty$, the Markov chain T converges to P, and hence $R^k \to P$. Consequently, the quality of the parameter updates computed by CD increases, although the benefit of increasing k quickly diminishes.

The partition function of RBMs was believed to be difficult to estimate in practical settings until Salakhutdinov and Murray (2008) applied Annealed Importance Sampling (Neal, 2001) to obtain an unbiased estimate of it. Their results showed that CD$_1$ is inferior to CD$_3$, which in turn is inferior to

CD_{25} for fitting RBMs to the MNIST dataset. Thus the best generative models can be obtained only with fairly expensive parameter updates that are derived from CD. Note that it is possible that the results of Salakhutdinov and Murray (2008) have substantial variance and that the true value of the partition function is much larger than reported, but there is currently no evidence that it is the case.

The RBM can be slightly modified to allow the vector v to take real values; one way of achieving this is by modifying the RBM's definition as follows:

$$P(v,h) = \frac{\exp(-\|v\|^2/2\sigma^2 + v^\top b_v + h^\top b_h + h^\top W v)}{Z(\theta)} \tag{2.33}$$

This equation does not change the form of the gradients and the conditional distribution $P(h|v)$. The only change it introduces is in the conditional distribution $P(v|h)$, which is now equal to a multivariate Gaussian with parameters $\mathcal{N}((b_v + W^\top h)\sigma^2, \mathbf{I}\sigma^2)$. See Welling et al. (2005) and Taylor et al. (2007) for more details and generalizations.

2.7.1 Adding more hidden layers to an RBM

In this section we describe how to improve an ordinary RBM by introducing additional hidden layers, and creating a "better" representation of the data, as described by Hinton et al. (2006). This is useful for making the model more powerful and for allowing features of features.

Let $P(v,h)$ denote the joint distribution defined by the RBM. The idea is to get another RBM, $Q(h,u)$, which has h as its visible and u as its hidden variables, to learn to model the aggregated posterior distribution, $\tilde{Q}(h)$, of the first RBM:

$$\tilde{Q}(h) \equiv \sum_v P(h|v)S(v). \tag{2.34}$$

When a single RBM is not an adequate model for v, the aggregated posterior $\tilde{Q}(h)$ typically has many regularities that can be modelled with another model. Thus provided $Q(h)$ approximates $\tilde{Q}(h)$ better than $P(h)$ does, it can be shown that the augmented model $M_{PQ}(v,h,u) = P(v|h)Q(h,u)$ is a better model of the original data than the distribution $P(v,h)$ defined by the first RBM alone (Hinton et al., 2006). It follows from the definition that $M_{PQ}(v,h,u)$ uses the undirected connections learned by Q between h and u, but it uses directed connections from h to v. It thus inherits $P(v|h)$ from the first RBM but discards $P(h)$ from its generative model. Data can be generated from the augmented model by sampling from $Q(h,u)$ by running a Markov chain, discarding the value of u, and then sampling from $P(v|h)$ (in a single step) to obtain v. Provided $N_u \geq N_v$, the RBM Q can be initialized by using the parameters from P to ensure that the two RBMs define the same distribution over h. Starting from this initialization, optimization then ensures that $Q(h)$ models $\tilde{Q}(h)$ better than $P(h)$ does.

The second RBM, $Q(h,u)$, learns by fitting the distribution $\tilde{Q}(h)$, which is not equivalent to maximizing $\log M_{PQ}(v)$. Nevertheless, it can be proved (Hinton et al., 2006) that this learning procedure maximizes a variational lower bound on $\log M_{PQ}(v)$. Even though $M_{PQ}(v,h,u)$ does not involve the conditional $P(h|v)$, we can nonetheless approximate the posterior distribution $M_{PQ}(h|v)$ by $P(h|v)$. Applying the standard variational bound (Neal and Hinton, 1998), we get

$$\mathcal{L} \geq \mathbf{E}_{P(h|v),v\sim S}\left[\log Q(h)P(v|h)\right] + \mathbb{H}_{v\sim S}(P(h|v)). \tag{2.35}$$

where $\mathbb{H}(P(h|v))$ is the entropy of $P(h|v)$. Maximizing this lower bound with respect to the parameters of Q whilst holding the parameters of P and the approximating posterior $P(h|v)$ fixed is precisely equivalent to fitting $Q(h)$ to $\tilde{Q}(h)$. Note that the details of Q are unimportant; Q could be any model and not an RBM. The main advantage of using another RBM is that it is possible to initialize $Q(h)$

to be equal to $P(h)$, so the variational bound starts as an equality and any improvement in the bound guarantees that $M_{PQ}(v)$ is a better model of the data than $P(v)$.

This procedure can be repeated recursively as many times as desired, creating very deep hierarchical representations. For example, a third RBM, $R(u, x)$ can be used to model the aggregated approximate posterior over u obtained by

$$\tilde{R}(u) = \sum_v \sum_h Q(u|h)P(h|v)S(v). \tag{2.36}$$

Provided $R(u)$ is initialized to be the same as $Q(u)$, the distribution $M_{PQR}(h)$ will be a better model of $\tilde{Q}(h)$ than $Q(h)$, but this does not mean that $M_{PQR}(v)$ is necessarily a better model of $S(v)$ than $M_{PQ}(v)$. It does mean, however, that the variational lower bound using $P(h|v)$ and $Q(u|h)$ to approximate the posterior distribution $M_{PQR}(u|v)$ will be equal to the variational lower bound to $M_{PQ}(v)$ of eq. 2.35, and learning R will further improve this variational bound.

2.8 Recurrent Neural Network Algorithms

2.8.1 Real-Time Recurrent Learning

Real-Time Recurrent Learning (RTRL; Williams and Zipser, 1989) is an elegant forward-pass only algorithm that computes the derivatives of the RNN w.r.t. its parameters at each timestep. Unlike BPTT, which requires an entire forward and a backward pass to compute a single parameter update, RTRL maintains the exact derivative of the loss so far at each timestep of the forward pass, without a backward pass and without the need to store the past hidden states. This property allows it to update the parameters after each timestep, which makes the learning "online" (as opposed to the "batch" learning of BPTT that requires an entire forward and a backward pass before the parameters can be updated.

Sadly the computational cost of RTRL is prohibitive, as it uses $|\theta|$ concurrent applications of forward differentiation, each of which obtains the derivative of the cumulative loss w.r.t. a single parameter at every timestep. It requires $|\theta|/2$ times more computation and $|\theta|/T$ more memory than BPTT. Although it is possible to make RTRL time-efficient with the aid of parallelization, the amount of resources required to do so is prohibitive.

2.8.2 Skip Connections

The vanishing gradients problem is one of the main difficulties in the training of RNNs. To mitigate it, we could reduce the number of nonlinearities separating the relevant past information from the current hidden unit by introducing direct connections between the past and the current hidden state. Doing so reduces the number of nonlinearities in the shortest path which makes the learning problem less "deep" and therefore easier.

One of the earlier uses of skip connections was in the Nonlinear AutoRegressive with eXogenous inputs method (NARX; Lin et al., 1996), where they improved the RNN's ability to infer finite state machines. They were also successfully used by the Time-Delay Neural network (TDNN; Waibel et al., 1989), although the TDNN was not recurrent and could process temporal information only with the aid of the skip connections.

2.8.3 Long Short-Term Memory

Long Short-Term Memory (LSTM; Hochreiter and Schmidhuber, 1997) is an RNN architecture that elegantly addresses the vanishing gradients problem using "memory units". These linear units have a

self-connection of strength 1 and a pair of auxiliary "gating units" that control the flow of information to and from the unit. When the gating units are shut, the gradients can flow through the memory unit without alteration for an indefinite amount of time, thus overcoming the vanishing gradients problem. While the gates never isolate the memory unit in practice, this reasoning shows that the LSTM addresses the vanishing gradients problem in at least some situations, and indeed, the LSTM easily solves a number of synthetic problems with pathological long-range temporal dependencies that were previously believed to be unsolvable by standard RNNs.[6] LSTMs were also successfully applied to speech and handwritten text recognition (Graves and Schmidhuber, 2009, 2005), robotic control (Mayer et al., 2006), and to solving Partially-Observable Markov Decision Processes (Wierstra and Schmidhuber, 2007; Dung et al., 2008).

We now define the LSTM. Let N be the number of memory units of the LSTM. At each timestep t, the LSTM maintains a set of vectors, described in table 2.1, whose evolution is governed by the following equations:

$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{hv}v_t + W_{hm}\tilde{m}_{t-1}) && (2.37)\\
i^g_t &= \mathrm{sigmoid}(W_{igh}h_t + W_{igv}v_t + W_{igm}\tilde{m}_{t-1}) && (2.38)\\
i_t &= \tanh(W_{ih}h_t + W_{iv}v_t + W_{im}\tilde{m}_{t-1}) && (2.39)\\
o_t &= \mathrm{sigmoid}(W_{oh}h_t + W_{ov}v_t + W_{om}\tilde{m}_{t-1}) && (2.40)\\
f_t &= \mathrm{sigmoid}(b_f + W_{fh}h_t + W_{fv}v_t + W_{fm}\tilde{m}_{t-1}) && (2.41)\\
m_t &= m_{t-1} \odot f_t + i_t \odot i^g_t \quad \text{the input gate allows the memory unit to be updated} && (2.42)\\
\tilde{m}_t &= m_t \odot o_t \quad \text{the output gate determines if information can leave the unit} && (2.43)\\
z_t &= g(W_{yh}h_t + W_{ym}\tilde{m}_t) && (2.44)
\end{aligned}
$$

The product $\odot$ denotes element-wise multiplication. Eqs. 2.37-2.41 are standard RNN equations, but eqns. 2.42-2.43 are the main LSTM-specific equations defining the memory units that describe how the input and output gates guard the contents of the memory unit. They also describe how the forget gate can make the memory unit forget its contents.

The gating units are implemented by multiplication, so it is natural to restrict their domain to $[0, 1]^N$, which corresponds to the $\mathrm{sigmoid}$ nonlinearity. The other units do not have this restriction, so the $\tanh$ nonlinearity is more appropriate.

We have included an explicit bias b_f for the forget gates because it is important for them to be approximately 1 at the early stages of learning, which is accomplished by initializing b_f to a large value (such as 5). If it is not done, it will be harder to learn long range dependencies because the smaller values of the forget gates will create a vanishing gradients problem.

Since the forward pass of the LSTM is relatively intricate, the equations for the correct derivatives of the LSTM are highly complex, making them tedious to implement. Fortunately, LSTMs can now be easily implemented with Theano, which can compute arbitrary derivatives efficiently (Bergstra et al., 2010).

2.8.4 Echo-State Networks

The Echo-State Network (ESN; Jaeger and Haas, 2004) is a standard RNN that is trained with the ESN training method, which learns neither the input-to-hidden nor the hidden-to-hidden connections, but sets them to draws from a well-chosen distribution, and only uses the training data to learn the hidden-to-output connections.

[6] We discovered (in Chap. 7) that standard RNNs are capable of learning to solve these problems provided they use an appropriate random initializaiton.

variable name	description
i_t^g	$[0,1]^N$-valued vector of input gates
i_t	$[-1,1]^N$-valued vector of inputs to the memory units
o_t	$[0,1]^N$-valued vector of output gates
f_t	$[0,1]^N$-valued vector of the forget gates
v_t	$\mathbb{R}^v$-valued input vector
h_t	$[-1,1]^h$-valued conventional hidden state
m_t	$\mathbb{R}^N$-valued state of the memory units
$\tilde{m}_t$	$\mathbb{R}^N$-valued memory state available to the rest of the LSTM
z_t	the output vector

Table 2.1: A list and a description of the variables used by the LSTM.

It may at first seem surprising that an RNN with random connections can be effective, but random parameters have been successful in several domains. For example, random projections have been used in machine learning, hashing, and dimensionality reduction (Datar et al., 2004; Johnson and Lindenstrauss, 1984), because they have the desirable property of approximately preserving distances. And, more recently, random weights have been shown to be effective for convolutional neural networks on problems with very limited training data (Jarrett et al., 2009; Saxe et al., 2010). Thus it should not be surprising that random connections are effective at least in some situations.

Unlike random projections or convolutional neural networks with random weights, RNNs are highly sensitive to the scale of the random recurrent weight matrix, which is a consequence of the exponential relationship between the scale and the evolution of the hidden states (most easily seen when the hidden units are linear). Too small recurrent connections cause the hidden state to have almost no memory of its past inputs, while too large recurrent connections cause the hidden state sequence to be chaotic and difficult to decode. But when the recurrent connections are sparse and are scaled so that their spectral radius is slightly less than 1, the hidden state sequence remembers its inputs for a limited but nontrivial number of timesteps while applying many random transformations to it, which are often useful for pattern recognition.

Some of the most impressive applications of ESNs are to problems with pathological long range dependencies (Jaeger, 2012b). It turns out that the problems of Chapter 4 can easily be solved with an ESN that has several thousand hidden units, provided the ESN is initialized with a semi-random initialization that combines the correctly-scaled random initialization with a set of manually fixed connections that implement oscillators with various periods that drive the hidden state. [7]

Despite its impressive performance on the synthetic problems from Martens and Sutskever (2011), the ESN has a number of limitations. Its capacity is limited because its recurrent connections are not learned, so it cannot solve data-intensive problems where high-performing models must have millions of parameters. In addition, while ESNs achieve impressive performance on toy problems (Jaeger, 2012b),

[7] ESNs can trivially solve other problems with pathological long range dependencies using explicit integration units, whosed dynamics are given by

$$h_t = \alpha h_{t-1} + (1-\alpha)\tanh(W_{hh}h_{t-1} + W_{vh}v_t) \tag{2.45}$$

However, explicit integration trivializes most of the synthetic problems from Martens and Sutskever (2011) (see Jaeger (2012b)), since if W_{hh} is set to zero and α is set to nearly 1, then $h_T \approx (1-\alpha)\sum_{t=1}^{T}\tanh(W_{vh}v_t)$, and simple choices of the scales of W_{vh} make it trivial for h_T to represent the solution. However, integration is not useful for the memorization problems of Chapter 4, and when integration is absent (or ineffective), the ESN must be many times larger than the smallest equivalent standard RNN that can learn to solve these problems (Jaeger, 2012a).

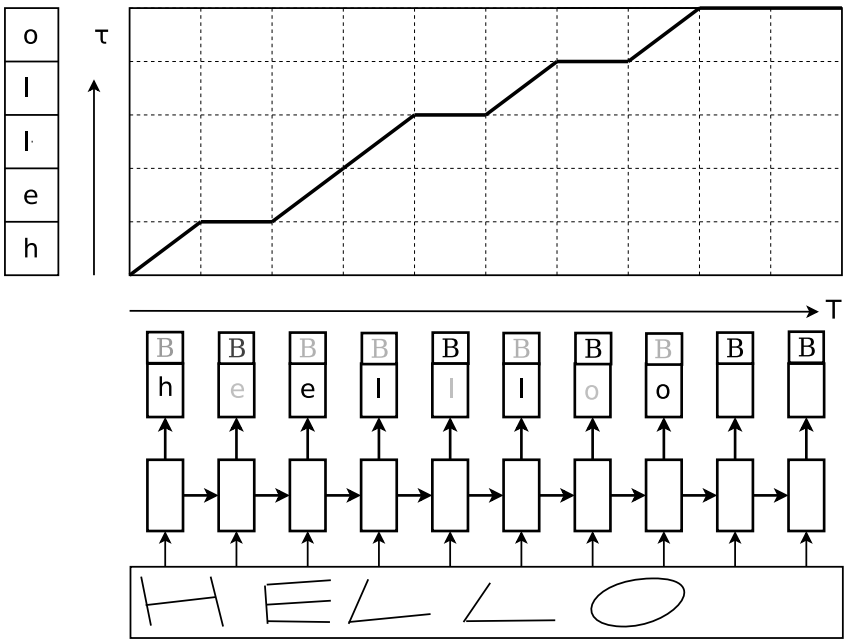


Figure 2.5: A diagram of the alignment computed by CTC. A neural network makes a prediction each timestep of the long signal. CTC aligns the network's predictions to the target label ("hello" in the figure), and reinforces this alignment with gradient descent. The figure also shows the network's prediction of the blank symbol. The matrix in the figure represents the S matrix from the text, which represents a distribution over the possible alignments of the input image to the label.

the size of high-performing ESNs grows very quickly with the information that the hidden state needs to carry. For example, the 20-bit memorization problem (Chap. 7) requires an ESN with at least 2000 units (while being solvable by RNNs that have 100 units whose recurrent connections are allowed to adapt; Chapter 4). Similarly, the ESN that achieves nontrivial performance on the TIMIT speech recognition benchmark used 20,000 hidden units (Triefenbach et al., 2010), and it is likely that ESNs achieving nontrivial performance on language modelling (Sutskever et al., 2011; Mikolov et al., 2010, 2011) will require even larger hidden states due to the information-intensive nature of the problem. This explanation is consistent with the performance characteristics of random convolutional networks, which excel only when the number of labelled cases is very small, so systems that adapt all the parameters of the neural network lose because of overfitting.

But while ESNs do not solve the problem of RNN training, their impressive performance suggests that an ESN-based initialization could be successful. This is confirmed by the results of Chapter 7.

2.8.5 Mapping Long Sequences to Short Sequences

Our RNN formulation assumes that the length of the input sequence is equal to the length of the output sequence. It is a fairly severe limitation, because most sequence pattern recognition tasks violate this assumption. For example, in speech recognition, we may want to map long sequence of frames (where each frame is a spectrogram segment that can span between 50-200 ms) to the much shorter phoneme sequence or the even shorter character sequence of the correct transcription. Furthermore, the length of the target sequence need not directly depend on the length of the input sequence.

The problem of mapping long sequences to short sequences has been addressed by Bengio (1991);

Bottou et al. (1997); LeCun et al. (1998), using dynamic programming techniques, which was successfully applied to handwritten text recognition. In this section, we focus on Connectionist Sequence Classification (CTC; Graves et al., 2006), which is a more recent embodiment of the same idea. It has been used with LSTMs to obtain the best results for Arabic handwritten text recognition (Graves and Schmidhuber, 2009) and the best performance on the slightly easier "online text recognition" problem (where the text is written on a touch-pad) (Graves et al., 2008).

CTC computes a gradient by aligning the network's predictions (a long sequence) with the target sequence (a short sequence), and uses the alignent to provide a target for each timestep of the RNN (fig. 2.5). This idea is formalized probabilistically as follows. Let there be K distinct output labels, $\{1, \ldots, K\}$ for each timestep, and suppose that the RNN (or the LSTM) outputs a sequence of T predictions. The prediction of each timestep t is a distribution over the $K+1$ labels $(p_t^1, \ldots, p_t^K, p_t^{\mathrm{B}})$ which includes the K output labels and a special blank symbol B which represents the absence of a prediction. CTC defines a distribution over sequences $(l_1, \ldots, l_M)$ (where each symbol $l_i \in \{1, \ldots, K\}$ and whose length is $M \leq T$):

$$P(l|p) = \sum_{1 \leq i_1 < \ldots < l_M \leq T} \prod_{j=1}^{M} p_{i_j}^{l_j} \prod_{m \notin \{i_1, \ldots, i_M\}} p_m^{\mathrm{B}} \tag{2.46}$$

In other words, CTC defines a distribution over label sequences $P(l|p)$ which is obtained by independently sampling a symbol at each timestep t and dropping all the blank symbols B.

We now show that $\log P(l|p)$ can be efficiently computed with dynamic programming, so its derivatives can be obtained with automatic differentiation. Recalling that the length of p is T and of l is M, the idea is to construct a matrix S of size $(M+1) \times (T+1)$ whose indicies start with 0, so that every monotonic path from the bottom-left entry up to the top-right entry corresponds to an alignment of p to l (fig. 2.5). The entries of the matrix for $m > 0$ and $t > 0$ are given by the expression

$$S_{m,t} = \sum_{1 \leq i_1 < \ldots < l_m \leq t} \prod_{j=1}^{m} p_{i_j}^{l_j} \prod_{j \notin \{i_1, \ldots, i_m\}, j \leq t} p_j^{\mathrm{B}} \tag{2.47}$$

It is a useful matrix because its top-right entry satisfies $S_{M,T} = P(l|p)$ and because it is efficiently computable with dynamic programming:

$$S_{0,0} = 1 \tag{2.48}$$

$$S_{m,0} = 0 \qquad m > 0 \tag{2.49}$$

$$S_{0,t} = p_t^{\mathrm{B}} \cdot S_{0,t-1} \qquad t > 0 \tag{2.50}$$

$$S_{m,t} = p_t^{\mathrm{B}} \cdot S_{m,t-1} + p_t^{l_m} \cdot S_{m-1,t-1} \tag{2.51}$$

The S matrix encodes a distribution over alignments, and its gradient reinforces those alignments that have a relatively high probability. Thus CTC is a latent variable model (with the alignment being the latent variable) based on the EM algorithm.

Unfortunately, it is not obvious how CTC could be implemented with neural network-like hardware due to the need to store a large alignment matrix in memory. Hence it is worth devising more neurally-plausible approaches to alignment, based on the CTC or otherwise.

At prediction time we need to solve the decoding problem, which is the problem of computing the MAP prediction

$$\arg\max_l P(l|p) \tag{2.52}$$

Unfortunately the problem is intractable because it is necessary to use dynamic programming to evaluate $P(l|p)$ for a single l, so classical search techniques such as beam-search are used for approximate decoding (Graves et al., 2006).

2.8.6 Truncated Backpropagation Through Time

Truncated backpropagation (Williams and Peng, 1990) is arguably the most practical method for training RNNs. The earliest use of truncated BPTT was by Elman (1990), and since then truncated-BPTT has successfully trained RNNs on word-level language modelling (Mikolov et al., 2010, 2011, 2009) that achieved considerable improvements over much larger N-gram models.

One of the main problems of BPTT is the high cost of a single parameter update, which makes it impossible to use a large number of iterations. For instance, the gradient of an RNN on sequences of length 1000 costs the equivalent of a forward and a backward pass in a neural network that has 1000 layers. The cost can be reduced with a naive method that splits the 1000-long sequence into 50 sequences (say) each of length 20 and treats each sequence of length 20 as a separate training case. This is a sensible approach that can work well in practice, but it is blind to temporal dependencies that span more than 20 timesteps. Truncated BPTT is a closely related method that has the same per-iteration cost, but it is more adept at utilizing temporal dependencies of longer range than the naive method. It processes the sequence one timestep at a time, and every k_1 timesteps, it runs BPTT for k_2 timesteps, so a parameter update can be cheap if k_2 is small. Consequently, its hidden states have been exposed to many timesteps and so may contain useful information about the far past, which would be opportunistically exploited. This cannot be done with the naive method.

Truncated BPTT is given below:

1: **for** t **from** 1 **to** T **do**
2: Run the RNN for one step, computing h_t and z_t
3: **if** t **divides** k_1 **then**
4: Run BPTT (as described in sec. 2.5), from t down to $t - k_2$
5: **end if**
6: **end for**

Chapter 3

The Recurrent Temporal Restricted Boltzmann Machine

In the first part of this chapter, we describe a new family of non-linear sequence models that are substantially more powerful than hidden Markov models (HMM) or linear dynamical systems (LDS). Our models have simple approximate inference and learning procedures that work well in practice. Multilevel representations of sequential data can be learned one hidden layer at a time, and adding extra hidden layers improves the resulting generative models. The models can be trained with very high-dimensional, very non-linear data such as raw pixel sequences. Their performance is demonstrated using synthetic video sequences of two balls bouncing in a box. In the second half of the chapter, we show how to modify the model to make it easier to train by introducing a deterministic hidden state that makes it possible to apply BPTT.

3.1 Motivation

Many different models have been proposed for high-dimensional sequential data such as video sequences or the sequences of coefficient vectors that are used to characterize speech. Models that use latent variables to propagate information through time can be divided into two classes: *tractable* models for which there is an efficient procedure for inferring the exact posterior distribution over the latent variables and *intractable* models for which there is no exact and efficient inference procedure. Tractable models such as linear dynamical systems and hidden Markov models have been widely applied but they are very limited in the types of structure that they can model. To make inference tractable when there is componential hidden state[1], it is necessary to use linear models with Gaussian noise so that the posterior distribution over the latent variables is Gaussian. Hidden Markov Models combine non-linearity with tractable inference by using a posterior that is a discrete distribution over a fixed number of mutually exclusive alternatives, but the mutual exclusion makes them exponentially inefficient at dealing with componential structure: to allow the history of a sequence to impose N bits of constraint on the future of the sequence, an HMM requires at least 2^N nodes. Inference remains tractable in mixtures of linear dynamical systems (Ghahramani and Hinton, 2000), but if we want to switch from one linear dynamical system to another *during* a sequence, exact inference becomes intractable (Ghahramani and Hinton,

[1] A componential hidden state differs from a non-componential hidden state chiefly by its number of possibly configuration. The number of configurations of a componential hidden state are exponential in its size. In contrast, the number of configuration of a non-componential hidden state, such as the hidden state of the HMM, is linear in its size.

2000). Inference is also tractable in products of hidden Markov models (Brown and Hinton, 2001). [2]

To overcome the limitations of the tractable models, many different schemes have been proposed for performing approximate inference (Isard and Blake, 1996; Ghahramani and Jordan, 1997). Boyen and Koller (1998) investigated the properties of a class of approximate inference schemes in which the true posterior density in the latent space is approximated by a simpler "assumed" density such as a mixture of a modest number of Gaussians (Ihler et al., 2004). At each time step, the model dynamics and/or the likelihood term coming from the next observation causes the inferred posterior density to become more complicated, but the inferred posterior is then approximated by a simpler distribution that lies in the space of assumed distributions. Boyen and Koller showed that the stochastic dynamics attenuates the approximation error created by projecting into the assumed density space and that this attenuation typically prevents the approximation error from diverging.

In this chapter we describe a family of generative models for sequential data that can capture many of the regularities that cannot be modeled efficiently by hidden Markov models or linear dynamical systems. The key idea is to use an undirected model for the interactions between the hidden and visible variables. This ensures that the contribution of the likelihood term to the posterior over the hidden variables is approximately factorial, which greatly facilitates inference. The model family has some attractive properties:

- It has componential hidden state which means it has an exponentially large state space[3].
- It has non-linear dynamics and it can make multimodal predictions.
- There is a very simple on-line filtering procedure which provides a reasonable approximation to the true conditional distribution over the hidden variables given the data observed so far.
- Even though maximum likelihood learning is intractable, there is a simple and efficient learning algorithm that finds good values for the parameters.
- There is a simple way to learn multiple layers of hidden variables and this can greatly improve the overall generative model.

By using approximations for both inference and learning, we obtain a family of models that are much more powerful than those that are normally used for modeling sequential data. The empirical question is whether our approximations are good enough to allow us to exploit the power of this family for modeling real sequences in which each time-frame is high-dimensional and the past has high-bandwidth non-linear effects on the future.

3.2 The Temporal Restricted Boltzmann Machine

Figure 3.1 shows an RBM that has been augmented by directed connections from previous states of the visible and hidden units. We call this a Temporal Restricted Boltzmann Machine (TRBM). The resulting sequence model is defined as a product of standard RBMs conditioned on the previous states of the hidden and the visible variables. As a result of this definition, the log probability of a sequence decouples into a sum, where each term is learned separately and efficiently by CD; approximate filtering

[2] Products of linear dynamical systems are linear dynamical systems and mixtures of hidden Markov models are hidden Markov models.

[3] The number of parameters is only quadratic, so there are strong limitations on how the exponentially large state space can be used, but for sequences in which there are several independent things going on at once, it is easy to use different subsets of the hidden units to model different components of the sequential structure.

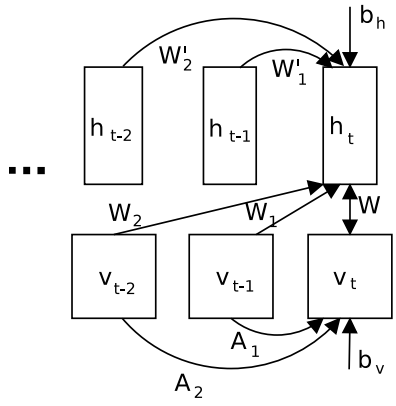


Figure 3.1: A schematic diagram of the TRBM. It is a directed model with an RBM at each timestep.

is easy (see subsection 3.2.1); and finally, it is straightforward to introduce more hidden layers to the model, as it is for the RBM.

The TRBM defines a joint distribution over (v_t, h_t) that is conditional on earlier hidden and visible states. The effect of these earlier states is to dynamically adjust the effective biases of the visible and hidden units at time t:

$$P(v_t, h_t | v_{t-m}^{t-1}, h_{t-m}^{t-1}) = \frac{\exp\left(h_t^\top W v_t + B_v(v_{t-m}^{t-1})^\top v_t + B_h(v_{t-m}^{t-1}, h_{t-m}^{t-1})^\top h_t\right)}{Z(v_{t-m}^{t-1}, h_{t-m}^{t-1})} \tag{3.1}$$

where v_t, h_t denote the state of the variables at time t and the notation v_{t-m}^{t-1} stands for $v_{t-1}, \ldots, v_{t-m}$, and likewise, h_{t-m}^{t-1}. The functions $B_v(v_{t-m}^{t-1})$ and $B_h(v_{t-m}^{t-1}, h_{t-m}^{t-1})$ represent dynamic biases that provide the TRBM with its dynamic biases, and are defined by the equations

$$\begin{aligned} B_v\left(v_{t-m}^{t-1}\right) &= A_1 v_{t-1} + \cdots + A_m v_{t-m} + b_v \\ B_h\left(v_{t-m}^{t-1}, h_{t-m}^{t-1}\right) &= W'_1 h_{t-1} + \cdots + W'_m h_{t-m} + W_1 v_{t-1} + \cdots + W_m v_{t-m} + b_h \end{aligned} \tag{3.2}$$

Consequently, Z depends on the states of $v_{t-m}^{t-1}, h_{t-m}^{t-1}$, as well as on the weight matrices W, $\{W_j\}_{j \le m}$, $\{W'_j\}_{j \le m}$, $\{A_j\}_{j \le m}$, and b_v, b_h that parameterize the bias functions. Thus, the TRBM is a standard RBM with W as its weight matrix and $B_h(\cdot,\cdot)$, $B_v(\cdot)$ as its biases; it is through the bias functions that the TRBM can model sequential structure. It is easy introduce additional connections to the model that make $B_v(\cdot)$ depend on h_{t-m}^{t-1} as well. Whenever the TRBM needs to use a value of v_τ or h_τ where τ is less than 1, we use learned "initial" values that depend on τ.

We model the probability of a whole sequence using a product of the distributions defined by a separate TRBM for each time step, with all of the TRBMs sharing the same parameters:

$$P(v_1^T, h_1^T) = \prod_{t=1}^{T} P(v_t, h_t | v_{t-m}^{t-1}, h_{t-m}^{t-1}) \tag{3.3}$$

Alg. 1 samples from the TRBM. It is fairly expensive, because it computes samples from an RBM at every timestep.

We chose the TRBM as the building block for our sequence model for three reasons. The first is that given v_1^t and h_1^{t-1} the distribution over h_t is factorial as long as the future variables v_{t+1}^T, h_{t+1}^T are unknown. This helps to ensure that a factorial approximation to the filtering distribution $P(h_t | v_1^t)$ is

Algorithm 1 Sampling from the TRBM

1: **for** $1 \leq t \leq T$ **do**
2: sample $v_t \sim P(v_t|h_{t-m}^{t-1}, v_{t-m}^{t-1})$
3: sample $h_t \sim P(h_t|v_{t-m}^{t}, h_{t-m}^{t-1})$
4: **end for**
5: **output** (v_1^T, h_1^T)

reasonably accurate. The second reason is that parameter estimation for an individual TRBM can be done efficiently using CD if we know the previous hidden and visible states, so by using the filtering distribution to approximate the posterior distribution over the hidden variables we can reduce the problem of modeling whole sequences to the problem of modeling the distribution of the current time frame of data given the previous data and the previous hidden states computed by the filtering distribution.

The third reason is that the TRBM model can easily be extended to include additional hidden layers (see section 3.4) and by adding more hidden layers we get a better representation and a better generative model.

3.2.1 Approximate Filtering

Our model is designed to make it easy to approximate the filtering distribution $P(h_t|v_1^t)$. Let $P_{\text{approx}}(h_{t,i} = 1|v_1^t)$ be the probability that the i^{th} hidden unit is one in the factorial approximation to the filtering distribution. For each time t we maintain a vector $p_t \in [0,1]^{N_h}$ such that $p_{t,i} = P_{\text{approx}}(h_{t,i} = 1|v_1^t)$. We show how to compute p_t, which clearly shows how to immediately obtain P_{approx}.

We derive our factorial approximation from the following observation. Suppose that v_1^t and h_1^{t-1} are known with certainty. In that case, the filtering distribution is truly factorial and is given by

$$P(h_{t,i} = 1|v_1^t, h_1^{t-1}) = \text{sigmoid}\left((Wv_t)_i + B_h\left(v_{t-m}^{t-1}, h_{t-m}^{t-1}\right)_i\right), \tag{3.4}$$

In the general case, we assume that v_1^t is given by the data with certainty but h_1^{t-1} is unknown and its uncertainty is represented by a factorial distribution P_{approx} (that is summarized by p). We use the mean-field equations (Peterson and Anderson, 1987) to compute p_t from p_1^{t-1} and v_1^t. The resulting equation is very similar to equation 3.4, except that we replace the values of the variables h_t with their probabilities p_t, thus getting the equation

$$p_{t,i} = \text{sigmoid}\left((Wv_t)_i + B_h\left(v_{t-m}^{t-1}, p_{t-m}^{t-1}\right)_i\right), \tag{3.5}$$

where we note that the definition of the bias function B_h is valid for real-valued arguments.

3.2.2 Learning

To allow online learning, we ignore the effect of future data on the inferred distribution over h_t and use the approximate filtering distribution as an approximate posterior (i.e., our distribution over h_t is a function of v_1^t and not v_1^T). Once no hidden variables remain, learning is done by a CD update for each time step separately.

Consider the standard lower bound to the log likelihood (Jordan et al., 1999):

$$\log P(v_1^T) \geq \mathbf{E}_{P_{\text{approx}}}\left[\log P\left(v_1^T, h_1^T\right)\right] + \mathbb{H}\left(P_{\text{approx}}\right), \tag{3.6}$$

where $\mathbb{H}$ is the entropy of a distribution, and $P_{\text{approx}}(h_1^T|v_1^T)$ is the approximate filtering distribution. We would like to maximize this lower bound with respect to P and P_{approx}; doing so enables us to obtain

the weight updates necessary for learning. Maximizing this lower bound with respect to P amounts precisely to learning each TRBM separately using the factorial hidden distribution provided by P_{approx}, but as a result of this maximization with respect to P, the distribution P_{approx} changes as well, and can possibly reduce the value of the bound. The fact that the learning works in practice suggests that this ignored effect is not too serious at least in some situations. Even though the lower bound is described with respect to one vector v_1^T, we maximize the average of these lower bounds over the training set, thus maximizing a lower bound to the average log likelihood. It is, in principle, possible to compute the correct derivatives of the bound w.r.t. the parameters of P_{approx}, but the variance of the gradient estimate will be infeasibly large, so we do not do so.

Learning a TRBM when the hidden states are known is simple. It is just an RBM with dynamic biases which can be learned in the same way as normal biases. In the equation below we write the weight update for a single TRBM. There are T such TRBMs, and the sum of their weight updates constitutes the full weight update. To simplify the notation we assume that there is only one training sequence in which case the weight update for time step t is

$$\Delta W \propto \mathbf{E}_{Q_1}\left[h_t {v_t}^\top\right] - \mathbf{E}_{Q_2^t}\left[h_t {v_t}^\top\right] \tag{3.7}$$

$$\Delta W_n \propto \mathbf{E}_{Q_1}\left[(h_t - \mathbf{E}_{Q_2^t}[h_t]) {v_{t-n}}^\top\right] \tag{3.8}$$

$$\Delta W'_n \propto \mathbf{E}_{Q_1}\left[(h_t - \mathbf{E}_{Q_2^t}[h_t]) {h_{t-n}}^\top\right] \tag{3.9}$$

$$\Delta A_n \propto \mathbf{E}_{Q_1}\left[(v_t - \mathbf{E}_{Q_2^t}[v_t]) {v_{t-n}}^\top\right] \tag{3.10}$$

The distribution Q_1 is the filtering distribution obtained given the single training sequence v_1^T, and the distribution Q_2^t is identical to Q_1 for frames $1, \ldots, t-1$, but for timestep t it is the TRBM distribution over (v_t, h_t) conditioned on the previous states h_1^{t-1} and v_1^{t-1}, namely

$$Q_2^t(v_1^t, h_1^t) = P(v_t, h_t | h_{t-m}^{t-1}, v_{t-m}^{t-1}) Q_1(v_1^{t-1}, h_1^{t-1}) \tag{3.11}$$

(the distribution of Q_2^t over the variables h_{t+1}^T, v_{t+1}^T is irrelevant). Note that even though the values of h_1^{t-1} are uncertain and are averaged over by Q_1 (which is also P_{approx}), in practice we substitute the values of each coordinate of h_1^{t-1} by p_1^{t-1}, the vector of probabilities of each coordinate being 1 under the filtering distribution Q_1 of v_1^T. This makes the biases to the TRBM deterministic and eases learning. We also cannot evaluate the expectations with respect to the TRBM distribution, so we use CD by replacing $Q_2^t(v_t, h_t)$ by the distribution obtained from running Gibbs sampling in the TRBM at time t for one step starting at v_t, exactly as for an RBM (sec. 2.7). We assumed that there was only one datapoint in the training set in the above description, but actually the datapoint is sampled from the training set, so the gradients are averaged by the empirical data distribution.

3.3 Experiments with a single layer model

To demonstrate that our learning procedure works we used it to learn synthetic video sequences composed of 20×20 grey-scale pixel time-frames of two balls bouncing in a box. The first row in figure 2 shows a sample from the training data. A movie can be viewed at the URL `www.cs.utoronto.ca/~ilya/aistats2007_filter/index.html`

In the pixel space, the dynamics are highly non-linear. Even if we could extract the positions and velocities of the centers of both balls, the dynamics would be highly non-linear when the balls bounce off the walls or off each other. Also, the underlying coordinates are related to the pixel intensities in a very

non-linear way. For all these reasons, modeling the raw sequence of pixel intensities is a challenging task which is made even more difficult if the model class cannot handle componential structure efficiently. An HMM, for example, would need about 10^4 hidden states to distinguish 10 values of the x and y positions and velocities of one ball, and 10^8 states for both balls.

We used several different TRBM models that had 400 visible units, 200 hidden units, and direct access to the hidden and visible states of the 4 previous time steps (i.e., $m = 4$). The full TRBM has 3 kinds of connections: connections between the hidden variables (HH), connections between the visible variables (VV) and connections between the visible and hidden variables (VH). In addition to trying the full TRBM we also tried leaving out each set of connections in turn. We call these special cases TRBM-VV, TRBM-HH, and TRBM-VH where the last part of the name indicates which connections are omitted. TRBM-VV, for example, has no visible-to-visible connections. Despite its name, TRBM-VH retains the undirected connections between the current instantiations of V and H.

The TRBM-HH model is an interesting special case because the lack of hidden-to-hidden connections makes exact inference possible. This model is particularly well suited for hierarchical learning, as we will show in section 3.4.

Each model was trained on 10,000 sequences of length 100. The weights were updated at the end of each sequence, with an initial learning rate of 0.00005 and momentum of 0.9. In addition, we double the learning rate at iteration 100, 200, 500 and 1000. This increase in the speed of learning proved crucial: without it, learning takes more than an order of magnitude more time, and even then it results in worse generative models. All four variations of the TRBM learned quite good generative models that could continue an initial segment of a video (see the URL above for examples of sequences generated by these models). The models could also be used for online denoising of sequences by performing approximate filtering and then reconstructing the visible state from the approximate filtering distribution. Figure 3.2 shows a typical image sequence and the same sequence corrupted by noise. The noise is correlated in both time and space which makes denoising much more difficult. All four variations of the TRBM denoise the sequence quite well, even though they were trained on noiseless data. Figure 3.2 shows the denoised sequence produced by the TRBM-VV which must use the hidden states to combine information across frames. When an extra hidden layer is added to any of the TRBMs (as described in the next section), there is a noticeable improvement in the denoising (see fig. 3.2), as well as in the generation (see the URL). To denoise with two hidden layers we first compute the approximate filtering distribution for the second hidden layer and then reconstruct each frame of the data from the second hidden layer.

Our models denoise much better than a simple RBM which cannot make use of previous frames. They are not as good as a linear autoregressive model that has been trained to predict the clean image from the four previous noisy ones, but our model is not trained with noise so it can denoise without requiring training data that contains both the noisy and the noisy-free sequence.

The biggest disadvantage of our models is that, before Tesla GPUs were available, they took 20 hours to train and even then the training was not complete. We also tried training a full TRBM with 400 hidden units for two weeks after which it had a model that generated extremely well.

3.4 Multilayer TRBMs

We straightforwardly generalize the idea (and use the notation) of sec. 2.7.1 to our sequence model. First we learn a TRBM, and then learn another TRBM that learns to model the hidden states of the first TRBM, which is precisely analogous to the way the RBM was augmented.

Denote by $P(v_1^T, h_1^T)$ the distribution defined by the first TRBM. The posterior $P(h_1^T|v_1^T)$ is not factorial, so we crudely approximate it by the filtering distribution, $P_{\text{approx}}(h_1^T|v_1^T)$. Let $Q(h_1^T, u_1^T)$ be a

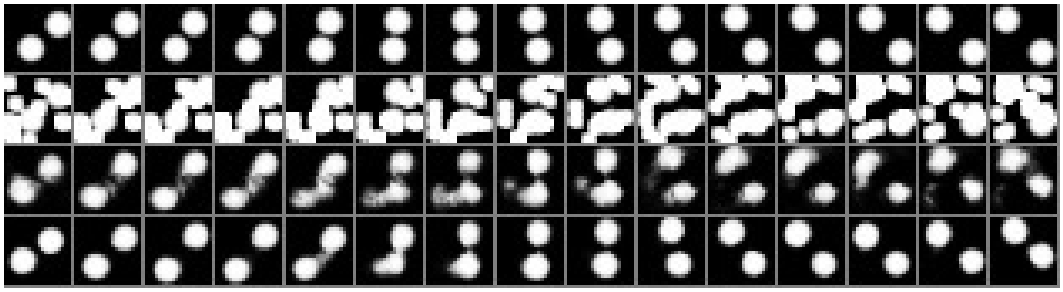


Figure 3.2: Top row: An image sequence. Second row: The same sequence corrupted by noise that is highly correlated in space and time. The noise consists of images of balls of a smaller radius whose position is fixed that appear and disappear at random times. Third row: Denoising by a TRBM-VV using a single hidden layer. Bottom row: Denoising by a TRBM-VV with two hidden layers (see sec. 3.4).

TRBM that we use to learn the aggregated approximate filtering distribution $\sum_{v_1^T} P_{\text{approx}}(h_1^T|v_1^T)S(v_1^T)$, where u_1^T is the sequence of the hidden variables of the TRBM Q. The approximate posterior of u_1^T, given by $Q_{\text{approx}}(u_1^T|h_1^T)P_{\text{approx}}(h_1^T|v_1^T)$ with h_1^T marginalized, allows u_1^T to represent higher-level features that can be computed on-line, since neither P_{approx} nor Q_{approx} make use of future frames. The resulting augmented generative model, $M_{PQ}(v_1^T)$, is one where we first sample from $Q(h_1^T)$, then from $P(v_1^T|h_1^T)$, so, as with the RBMs, $M_{PQ}(v_1^T) = \sum_{h_1^T} P(v_1^T|h_1^T)Q(h_1^T)$. If we can initialize Q so that $Q(h_1^T) = P(h_1^T)$, then the augmented model is identical to P and has the same likelihood. By making Q learn the distribution of the hidden states of P, which is $\sum_v P_{\text{approx}}(h_1^T|v_1^T)S(v_1^T)$, we maximize the lower bound

$$\mathcal{L} \geq \mathbf{E}_{P_{\text{approx}}}\left[\log Q\left(h_1^T\right) P\left(v_1^T|h_1^T\right)\right] + \mathbb{H}\left(P_{\text{approx}}\right), \tag{3.12}$$

with respect to Q. This is very similar to the bound in equation 2.35, except for the use of the approximate posterior. At the beginning of the optimization this bound is strictly *less* than $\log P(v_1^T)$ even when $Q(h_1^T) = P(h_1^T)$, because an approximate posterior is used. It could, therefore, remain less than $\log P(v_1^T)$. This is not the case for RBMs, since the exact posterior $P(h|v)$ is easily computable, so the lower bound is equal to $\log P(v)$ at the beginning of the optimization.

Although our learning procedure maximizes a lower bound that is initially smaller than $\log P(v_1^T)$, it is very likely that by the end of learning the bound will exceed $\log P(v_1^T)$. In addition, since we use an approximate posterior during the learning of $P(v_1^T)$ (recall that inference is intractable in our TRBM model), we are performing approximate maximization of a lower bound on $\log P(v_1^T)$ as well (this is also equation 3.6; see subsection 3.2.2):

$$\log P(v_1^T) \geq \mathbf{E}_{P_{\text{approx}}}\left[P\left(h_1^T\right) P\left(v_1^T|h_1^T\right)\right] + \mathbb{H}\left(P_{\text{approx}}\right), \tag{3.13}$$

(the maximization is approximate in that we ignore the effect that changing the approximate posterior has on the bound), so by introducing Q the new lower bound of equation 3.12 will be equal to the bound in equation 3.13 if Q is properly initialized. Therefore, the lower bound in eq. 3.12 will be greater than the lower bound in eq. 3.13.

In order to initialize Q such that $Q(h_1^T) = P(h_1^T)$, it is necessary for Q to have directed connections between its visible variables (the variables h_1^T) so that $Q(h_1^T)$ can represent every distribution $P(h_1^T)$ can. For RBMs, learning one hidden layer at a time works well even if $Q(h)$ is not initialized to be equal to $P(h)$ (Hinton et al., 2006), so in our experiments (see section 3.4.1), we did not initialize $Q(h_1^T) = P(h_1^T)$.

We can also add further hidden layers in the same way as is done for RBMs and each time another layer is added we should get a better generative model.

Notice that for the model TRBM-HH, for which $P(h_1^T|v_1^T)$ is exactly factorial, the situation is significantly better. Not only does it have an exact learning procedure (if we ignore the approximations introduced by contrastive divergence), but its augmented model *always* has a greater likelihood since the lower bound (eq. 3.13) is equal to the log likelihood if $Q(h_1^T) = P(h_1^T)$, because $P_{\text{approx}}(h_1^T|v_1^T) = P(h_1^T|v_1^T)$.

A drawback of TRBMs with direct connections between their hidden units is that $P(v_1^T|h_1^T)$ is not factorial. This makes it intractable to generate unbiased samples from the multilayered models, so further approximations must be used.

3.4.1 Results for multilevel models

We conducted experiments to determine whether adding an extra hidden layer improves the quality of generative models. For each of TRBM, TRBM-VV, TRBM-VH, TRBM-HH, we used the same type of TRBM with 400 hidden units (and 200 visibles) to learn the aggregated posterior distribution of the hidden units in the first-level model. The learning parameters of all these models were the same as those for the original TRBMs and training lasted for 10,000 updates. All of the generative models improved and they all became better at denoising (see figure 3.2 for a typical denoising example, or the URL for many movies of denoising and generation).

Despite the improved performance, we cannot generate exactly from the improved multilevel models if they have visible-to-visible connections. To generate a sample (see sec. 2.7), we first need to use $Q(h_1^T, u_1^T)$ to sample the activities of h_1^T and then we need to sample from $P(v_1^T|h_1^T)$, which is the distribution over sequences of visible frames given a sequence of hidden frames. This distribution is intractable for the same reasons inference is intractable in our models, and we approximate it in a similar spirit. However, if they do not have visible-to-visible connections, then it is possible to draw samples from the multilayered model.

3.5 The Recurrent Temporal Restricted Boltzmann Machine

The TRBM, while powerful and expressive, is unappealing because of the crude approximations that are required to compute a parameter update. In the remainder of this chapter we introduce the Recurrent TRBM (RTRBM), which is a model very similar to the TRBM that is just as expressive. But despite their similarity, the exact inference in RTRBMs is trivial and it is feasible to compute the gradient of the log likelihood up to the error introduced by the use of Contrastive Divergence. We demonstrate that the RTRBM generates more realistic samples than an equivalent TRBM for motion capture and for the pixels of videos of bouncing balls. The RTRBMs performance is better than the TRBM mainly because it learns to convey more information through its hidden-to-hidden connections. The first RTRBM was described by Sutskever et al. (2008) and later extended by Boulanger-Lewandowski et al. (2011, 2012).

3.6 Simplified TRBM

We will use a simplified definition of the TRBM for the reminder of the chapter. The simplified TRBM $P(v_1^T, h_1^T)$ is defined as follows:

$$P(v_1^T, h_1^T) = \prod_{t=1}^{T} P(v_t, h_t|h_{t-1}) \tag{3.14}$$

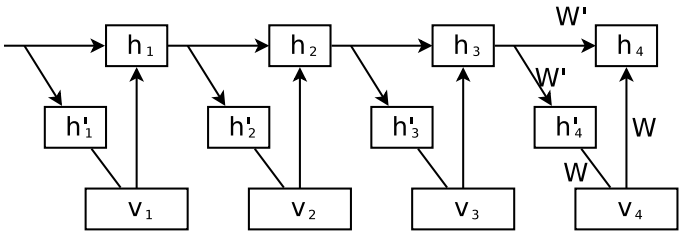


Figure 3.3: The graphical structure of the RTRBM. The variables h_t are real valued while the variables h'_t are binary. The conditional distribution $\hat{P}(v_t, h'_t|h_{t-1})$ is given by the equation $\hat{P}(v_t, h'_t|h_{t-1}) = \exp\left(h'^{\top}_t W v_t + v_t^{\top} b_v + h'_t(b_h + W' h_{t-1})\right)/Z(h_{t-1})$, which is essentially the same as the TRBMs conditional distribution P from equation 3.15. We will always integrate out h'_t and will work directly with the distribution $\hat{P}(v_t|h_{t-1})$. Notice that when v_1 is observed, h'_1 does not affect h_1.

The conditional distribution $P(v_t, h_t|h_{t-1})$ is an RBM whose biases for h_t are a function of h_{t-1}:

$$P(v_t, h_t|h_{t-1}) = \exp\left(v_t^{\top} b_v + h_t^{\top} W v_t + h_t^{\top}(b_h + W' h_{t-1})\right)/Z(h_{t-1}) \tag{3.15}$$

where b_v, b_h and W are as in eq. 2.24, and W' is the weight matrix of the connections from h_{t-1} to h_t, making $b_h + W' h_{t-1}$ be the bias of RBM at time t. In the above equations, h_0 is a parameter vector of the very first hidden state.

Algorithm 2 Sampling from the simplified TRBM

1: **for** $1 \leq t \leq T$ **do**
2: sample $v_t \sim P(v_t|h_{t-1})$
3: sample $h_t \sim P(h_t|v_t, h_{t-1})$
4: **end for**
5: **output** (v_1^T, h_1^T)

As in most probabilistic models, the weight update is computed by solving the inference problem and computing the weight update as if the inferred variables were observed. The main computational difficulty of learning TRBMs is in obtaining samples from a distribution approximating the posterior $P(h_1^T|v_1^T)$. But as we have seen earlier in the chapter, the inference problem is harder than that of a typical undirected graphical model, because even computing the probability $P(h_t^{(j)} = 1|$ everything else$)$ involves evaluating the exact ratio of two RBM partition functions.

3.7 Model Definition

Consider an arbitrary factorial distribution $P'(h)$. The statement $h \sim P'(h)$ means that h is sampled from the factorial distribution $P'(h)$, so each $h^{(j)}$ is set to 1 with probability $P'(h^{(j)} = 1)$, and 0 otherwise. We let the statement $h \leftarrow P'(h)$ mean that each $h^{(j)}$ is independently set to the real value $P'(h^{(j)} = 1)$, so this is a "mean-field" update (Peterson and Anderson, 1987; Wainwright and Jordan, 2003). Observe that the distribution $P(h_t|v_t, h_{t-1})$ can be represented by the vector $\mathrm{sigmoid}(W v_t + W' h_{t-1} + b_h)$ when P is a TRBM.

The RTRBM $\hat{P}(v_1^T, h_1^T)$ is given by an equation of the same structure as a TRBM:

$$\begin{aligned}\hat{P}(v_1^T, h_1^T) &= \prod_{t=1}^{T} \sum_{h_t'} \hat{P}(v_t, h_t'|h_{t-1}) \hat{P}(h_t|v_t, h_{t-1}) \\ &= \prod_{t=1}^{T} \hat{P}(v_t|h_{t-1}) \hat{P}(h_t|v_t, h_{t-1}) \end{aligned} \tag{3.16}$$

Alg. 2 draws a sample from the simplified TRBM with a forward pass that samples the conditional RBM at every timestep where step 2 requires sampling from the marginals of a Boltzmann Machine (by integrating out h_t), which involves running a Markov chain.

RTRBMs and TRBMs are parameterized in the same way, so P and $\hat{P}$ have identical parameters, which are W, W', b_v, b_h, and h_0. Then the following algorithm defines the distribution of the RRTBM:

Algorithm 3 Sampling from the RTRBM

1: **for** $1 \leq t \leq T$ **do**
2: sample $v_t \sim P(v_t|h_{t-1})$
3: set $h_t \leftarrow P(h_t|v_t, h_{t-1})$
4: **end for**
5: **output** (v_1^T, h_1^T)

We can infer that $\hat{P}(v_t|h_{t-1}) = P(v_t|h_{t-1})$ because of step 2 in Algorithm 3, which is also consistent with the equation given in figure 3.3 where h_t' is integrated out. The only difference between Algorithm 2 and Algorithm 3 is in step 3. The difference may seem minor, since the operations $h_t \sim P(h_t|v_t, h_{t-1})$ and $h_t \leftarrow P(h_t|v_t, h_{t-1})$ are superficially similar. However, this difference significantly alters the inference and learning procedures of the RTRBM.

3.8 Inference in RTRBMs

RTRBMs allow for trivial posterior inference (i.e., computing $\hat{P}(h_1^T|v_1^T)$), which might be surprising in light of its similarity to the TRBM whose posterior inference is completely intractable. The reason inference is easy is similar to the reason inference in square ICAs is easy (Bell and Sejnowski, 1995): there is a unique and easily computable value of the hidden variables that has a nonzero posterior probability. Suppose, for example, that the value of v_1 is given, which means that v_1 was produced at the end of step 2 in Algorithm 3. Since step 3, the deterministic operation $h_1 \leftarrow P(h_1|v_1, h_0)$, has been executed, the only value h_1 can take is the value assigned by the operation $\cdot \leftarrow P(h_1|v_1, h_0)$. Any other value for h_1 is never produced by a generative process that outputs v_1 and thus has posterior probability 0. In addition, by executing this operation, we can recover h_1. Thus, $\hat{P}(h_1|v_1) = \delta_{\text{sigmoid}(Wv_1+b_h+W'h_0)}(h_1)$. Note that h_1's value is unaffected by v_2^T.

Once h_1 is known, we can consider the generative process that produced v_2. As before, since v_2 was produced at the end of step 2, then the fact that step 3 has been executed implies that h_2 is given by $h_2 \leftarrow P(h_2|v_2, h_1)$ (recall that at this point h_1 is known with absolute certainty). If the same reasoning is repeated t times, then all of h_1^t is uniquely determined and is easily computed when v_1^t is known. There is no need for smoothing because v_t and h_{t-1} influence h_t with such strength that the knowledge of v_{t+1}^T cannot alter the model's belief about h_t. This is because $\hat{P}(h_t|v_t, h_{t-1}) = \delta_{\text{sigmoid}(Wv_t+b_h+W'h_{t-1})}(h_t)$.

The resulting inference algorithm is simple:

Algorithm 4 Inference in an RTRBM

1: **for all** $1 \leq t \leq T$ **do**
2: $\quad h_t \leftarrow P(h_t|v_t, h_{t-1})$
3: **end for**

Let $h(v)_1^T$ denote the output of the inference algorithm on input v_1^T, in which case the posterior is described by

$$\hat{P}(h_1^T|v_1^T) = \delta_{h(v)_1^T}(h_1^T) \tag{3.17}$$

Inference is simple in the RTRBM because future observations have no influence on the posterior of a given hidden state. However, this simplicity makes the true posterior of RTRBM less expressive than the true posterior of the TRBM because the latter is affected by future observations. However, it is not a major disadvantage in practice, because it is very difficult to incorporate future observations into the TRBM's posterior.

3.9 Learning in RTRBMs

Learning in RTRBMs may seem easy once inference is solved, since the main difficulty in learning TRBMs is the inference problem. However, it is not sufficient to infer the hidden states of the RTRBM for learning because the equation $\nabla \log \hat{P}(v_1^T) = \mathbf{E}_{h_1^T \sim \hat{P}(h_1^T|v_1^T)} \left[\nabla \log \hat{P}(v_1^T, h_1^T)\right]$ is not meaningful: the gradient $\nabla \log \hat{P}(v_1^T, h_1^T)$ is undefined because $\delta_{\text{sigmoid}(W' h_{t-1} + b_h + W^T v_t)}(h_t)$ is not a continuous function of the parameters. Thus, the gradient has to be computed differently.

Notice that the RTRBMs log probability satisfies $\log \hat{P}(v_1^T) = \sum_{t=1}^T \log \hat{P}(v_t|v_1^{t-1})$, so we could try computing the sum $\nabla \sum_{t=1}^T \log \hat{P}(v_t|v_1^{t-1})$. The key observation that makes the computation feasible is the equation

$$\hat{P}(v_t|v_1^{t-1}) = \hat{P}(v_t|h(v)_{t-1}) \tag{3.18}$$

where $h(v)_{t-1}$ is the value computed by the RTRBM inference algorithm with inputs v_1^t at time t. This equation holds because

$$\hat{P}(v_t|v_1^{t-1}) = \int_{h'_{t-1}} \hat{P}(v_t|h'_{t-1}) \hat{P}(h'_{t-1}|v_1^{t-1}) dh'_{t-1} = \hat{P}(v_t|h(v)_{t-1}) \tag{3.19}$$

as the posterior distribution $\hat{P}(h'_{t-1}|v_1^{t-1}) = \delta_{h(v)_{t-1}}(h'_{t-1})$ is a point-mass at $h(v)_{t-1}$, which follows from eq. 3.17.

The equality $\hat{P}(v_t|v_1^{t-1}) = \hat{P}(v_t|h(v)_{t-1})$ lets us define a recurrent neural network whose parameters are identical to those of the RTRBM, and whose cost function is equal to the log likelihood of the RTRBM. This is useful because it is easy to compute gradients with respect to the RNN's parameters using BPTT. The RNN has a pair of variables at each timestep, $\{(v_t, r_t)\}_{t=1}^T$, where v_t are the input variables and r_t are the RNN's hidden variables (all of which are deterministic). The hiddens r_1^T are computed by the equation

$$r_t = \text{sigmoid}(W v_t + b_h + W' r_{t-1}) \tag{3.20}$$

This definition was chosen so that the equation $r_1^T = h(v)_1^T$ would hold. The RNN attempts to probabilistically predict the next timestep from its history using the marginal distribution of the RBM $\hat{P}(v_{t+1}|r_t)$, so its objective function at time t is defined to be $\log \hat{P}(v_{t+1}|r_t)$, where $\hat{P}$ depends on the RNN's parameters in the same way it depends on the RTRBMs parameters (the two sets of parameters

being identical). Thus the RTRBM is an RNN generative model (sec. 2.5.2). This is a valid definition of an RNN whose cumulative objective for the sequence v_1^T is

$$L = \sum_{t=1}^{T} \log \hat{P}(v_t|r_{t-1}) \tag{3.21}$$

But since r_t as computed in equation 3.20 on input v_1^T is identical to $h(v)_t$, the equality $\log \hat{P}(v_t|r_{t-1}) = \log \hat{P}(v_t|v_1^{t-1})$ holds. Substituting this identity into eq. 3.21 yields

$$L = \sum_{t=1}^{T} \log \hat{P}(v_t|r_{t-1}) = \sum_{t=1}^{T} \log \hat{P}(v_t|v_1^{t-1}) = \log \hat{P}(v_1^T) \tag{3.22}$$

which is the log probability of the corresponding RTRBM.

This means that $\nabla L = \nabla \log \hat{P}(v_1^T)$ can be computed with the backpropagation through time algorithm (Rumelhart et al., 1986), where the contribution of the gradient from each timestep is computed with Contrastive Divergence.

3.10 Details of Backpropagation Through Time

We now describe the application of the backpropagation through time algorithm to the RTRBM. It maintains a term $\partial L/\partial r_t$ which is computed from $\partial L/\partial r_{t+1}$ and $\partial \log \hat{P}(v_{t+1}|r_t)/\partial r_t$ using the chain rule, by the equation

$$\partial L/\partial r_t = W'^{\top}(r_{t+1} \odot (1 - r_{t+1}) \odot \partial L/\partial r_{t+1}) + W'^{\top} \partial \log \hat{P}(v_t|r_{t-1})/\partial b_h \tag{3.23}$$

where $a \odot b$ denotes component-wise multiplication, the term $r_t \odot (1 - r_t)$ arises from the derivative of the logistic function $\text{sigmoid}'(x) = \text{sigmoid}(x) \odot (1 - \text{sigmoid}(x))$, and $\partial \log \hat{P}(v_{t+1}|r_t)/\partial b_h$ is computed by CD. Once $\partial L/\partial r_t$ is computed for all t, the gradients of the parameters can be computed using the following equations:

$$\frac{\partial L}{\partial W'} = \sum_{t=2}^{T} \left(r_t \odot (1 - r_t) \odot \frac{\partial L}{\partial r_t} \right) r_{t-1}^{\top} \tag{3.24}$$

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T-1} \left(W'^{\top}(r_{t+1} \odot (1 - r_{t+1}) \odot \frac{\partial L}{\partial r_{t+1}}) \right) v_t^{\top} + \sum_{t=1}^{T} \frac{\partial \log \hat{P}(v_t|r_{t-1})}{\partial W} \tag{3.25}$$

The first summation in eq. 3.25 corresponds to the use of W for computing r_t, and the second summation arises from the use of W as RBM parameters for $\log \hat{P}(v_t|r_{t-1})$, so each $\partial \log \hat{P}(v_{t+1}|r_t)/\partial W$ term is computed with CD. Computing $\partial L/\partial r_t$ is done most conveniently with a single backward pass through the sequence. It is also seen that the gradient of the RTRBM would be computed exactly if CD were replaced with the derivatives of the RBM's log probability.

3.11 Experiments

We report the results of experiments comparing an RTRBM to a TRBM. The results presented in the first half of this chapter, were obtained using TRBMs that had several delay-taps, which means that each hidden unit could directly observe several previous timesteps. To demonstrate that the RTRBM learns to use the hidden units to store information, we did not use delay-taps for the RTRBM nor the TRBM,

which causes the results to be worse (but not much) than the results of both Taylor et al. (2007) and the first half of this chapter. If delay-taps are allowed, then there is little benefit from the hidden-to-hidden connections, when modelling data with only short-term direct dependencies, which makes the comparison between the RTRBM and the TRBM uninteresting.

In all experiments, the RTRBM and the TRBM had the same number of hidden units, their parameters were initialized in the same manner, and they were trained for the same number of weight updates. When sampling from the TRBM, we would use the sampling procedure of the RTRBM using the TRBMs parameters to eliminate the additional noise from its hidden units. If this is not done, the samples produced by the TRBM are noiser and are less visually appealing, which arguably suggses that they are of poorer quality (although their noise might be properly representing uncertainty).

3.11.1 Videos of bouncing balls

We used a dataset consisting of videos of 3 balls bouncing in a box. The videos are of length 100 and of resolution 30×30. Each training example is synthetically generated, so no training sequence is seen twice by the model, which means that overfitting is not an issues with our models. The task is to learn to generate videos at the pixel level. This problem is high-dimensional, having 900 dimensions per timestep, and the RTRBM and the TRBM are given no prior knowledge about the nature of the task (e.g., by convolutional weight matrices). Both the RTRBM and the TRBM had 400 hidden units. Samples from these models are provided as videos 1,2 (RTRBM) and videos 3,4 (TRBM). A sample training sequence is given in video 5. All the samples can be found in `www.cs.utoronto.ca/~ilya/pubs/2008/rtrbm_`
The real values in the videos are the conditional probabilities of the pixels (Sutskever and Hinton, 2007). The RTRBMs samples are noticeably better than those of the TRBM. A difference between these samples is that the balls produced by the TRBM move in a random walk, while those produced by the RTRBM move in a noticeably persistent direction. An examination of the visible-to-hidden connections of the RTRBM reveals a number of hidden units that are not connected to any visible units. These units have the most active hidden to hidden connections, which is likely used to propagate information through time. In particular, these units are the only units that do not have a strong self connection (i.e., $W_{i,i}$ is not large; see figure 3.4). No such separation of units is found in the TRBM and all its hidden units have large visible to hidden connections. We can also use the TRBM and the RTRBM to predict the next timestep of the sequence in order to acquire some quantitative measure of the model's performance. The mean squared prediction error per pixel per timestep is 0.007 for the RTRBM and is 0.04 for the TRBM.

3.11.2 Motion capture data

We used a dataset that represents human motion capture data by sequences of joint angles, translations, and rotations of the base of the spine. The data was preprocessed to be invariant to isometries (Taylor et al., 2007). The total number of timesteps in the dataset set was 3826, from which the model learned on subsequences of length 50. Each timestep has 49 dimensions, and both models have 200 hidden units. The data is real-valued, so the TRBM and the RTRBM were adapted to have Gaussian visible variables using eq. 2.33. The samples produced by the RTRBM exhibit less sticking and foot-skate than those produced by the TRBM. Samples from these models are provided as videos 6 and 7 (RTRBM), and videos 8 and 9 (TRBM), and video 10 is a sample training sequence. Part of the Gaussian noise was removed in a manner described by Sutskever and Hinton (2007) in both models. The mean squared prediction error per dimension per timestep for the RTRBM is 0.42 and is 0.47 for the TRBM.

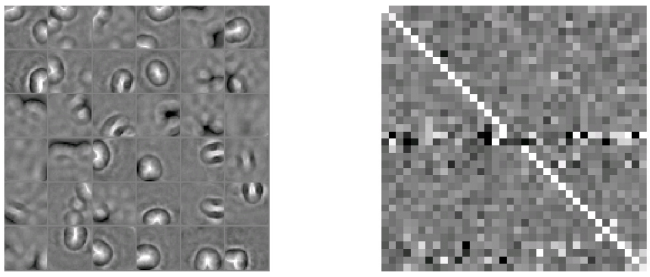


Figure 3.4: This figure shows the receptive fields of the first 36 hidden units of the RTRBM on the left, and the corresponding hidden-to-hidden weights between these units on the right: the ith row on the right corresponds to the ith receptive field on the left, when counted left-to-right. Hidden units 18 and 19 exhibit unusually strong hidden-to-hidden connections; they are also the ones with the weakest visible-hidden connections, which effectively makes them belong to another hidden layer.

3.11.3 Details of the learning procedures

Each problem was trained for 100,000 weight updates, with a momentum of 0.9. The the gradient was divided by the length of the sequence. The weights are updated after computing the gradient on a single sequence. The learning starts with CD_{10} for the first 1000 weight updates, which is then switched to CD_{25}. The visible-to-hidden weights, W, were initialized with static CD_5 (without using the (R)TRBM learning rules) on 30 sequences (which resulted in 30 weight updates) with learning rate of 0.01 and momentum 0.9. These weights were then given to the (R)TRBM learning procedure, where the learning rate was linearly reduced towards 0. The weights W' and the biases were initialized with a sample from spherical Gaussian of standard-deviation 0.005. For the bouncing balls problem the initial learning rate was 0.01, and for the motion capture data it was 0.005.

3.12 Conclusions

In this chapter we introduced the RTRBM, a probabilistic model as powerful as the intractable TRBM that has an exact inference and an almost exact learning procedure. The main disadvantage of the RTRBM is that it is a recurrent neural network, which is relatively difficult to train. However, this disadvantage is common to many other probabilistic sequence models, and it can be partially alleviated using techniques such as the long short term memory RNN (Hochreiter and Schmidhuber, 1997), as well as with the initialization techniques that are described in Chap. 7.

Chapter 4

Training RNNs with Hessian-Free Optimization

In this chapter, we show that recent advances in the Hessian-free (HF) optimization approach of Martens (2010) together with a novel damping scheme can successfully train RNNs on problems with pathological long range temporal dependencies. Prior to the work described in this chapter, standard RNNs (with the exception of the LSTM) were widely believed to impossible to train on such problems.

4.1 Motivation

Recently, a carefully designed and improved version of the Hessian-Free (HF) optimization approach, which is a practical large scale implementation of Newton's method, was successfully applied to learning deep neural networks from random initializations (Martens, 2010), which was considered to be infeasible with other optimization approaches until very recently (Glorot and Bengio, 2010, and Chap. 7).

We were inspired by the success of the HF approach on deep neural networks and were compelled to revisit the basic problem of RNN training. Our results show that HF, augmented with a novel "structural-damping" which we develop, can effectively train RNNs on the long-term dependency problems (adapted directly from Hochreiter and Schmidhuber (1997)) that span over 100 timesteps, thus overcoming the main objection made against using RNNs. From there we go on to address the question of whether these advances generalize to real-world sequence modeling problems by considering both a high-dimensional motion video prediction task, a MIDI-music modeling task, and a speech modelling task. We find that RNNs trained using our method are highly effective on these tasks and significantly outperform similarly sized LSTMs.

Other contributions we make include the development of the aforementioned structural damping scheme which, as we demonstrate through experiments, significantly improves the robustness of the HF optimizer in the setting of RNN training, and a new interpretation of the generalized Gauss-Newton matrix of Schraudolph (2002) which forms a key component of the HF approach of Martens (2010).

4.2 Hessian-Free Optimization

Hessian-Free optimization is concerned with the minimization of an objective $f : \mathbb{R}^N \to \mathbb{R}$, with respect to N-dimensional input vector θ. Its operation is viewed as the iterative minimization of simpler sub-objectives based on local approximations to $f(\theta)$. Specifically, given a parameter setting θ_n, the

next one, θ_{n+1}, is found by partially optimizing the sub-objective

$$q_{\theta_n}(\theta) \equiv M_{\theta_n}(\theta) + \lambda R_{\theta_n}(\theta), \tag{4.1}$$

In this equation, $M_{\theta_n}(\theta)$ is a θ_n-dependent "local" quadratic approximation to $f(\theta)$ given by

$$M_{\theta_n}(\theta) = f(\theta_n) + f'(\theta_n)^\top \delta_n + \frac{\delta_n^\top \mathrm{C}_n \delta_n}{2}, \tag{4.2}$$

where C_n is an approximation to the curvature of f at θ_n, the term δ_n is given by $\delta_n = \theta - \theta_n$, and $R_{\theta_n}(\theta)$ is a damping function that penalizes the solution according to the difference between θ and θ_n, thus encouraging θ to remain close to θ_n. The use of a damping function is generally necessary because the accuracy of the local approximation $M_{\theta_n}(\theta)$ degrades as θ moves further from θ_n.

In standard Hessian-free optimization (or 'truncated-Newton', as it is also known), M_{θ_n} is chosen to be the same quadratic as the one optimized in Newton's method: the full 2nd-order Taylor series approximation to f, which is obtained by taking $\mathrm{C}_n = f''(\theta_n)$ in eq. 4.2, and the damping function $R_{\theta_n}(\theta)$ is chosen to be $\|\delta_n\|^2/2$. Unlike with quasi-Newton approaches like L-BFGS there is no low-rank or diagonal approximation involved, and as a consequence, fully optimizing the sub-objective q_{θ_n} can require a matrix inversion or some similarly expensive operation. The HF approach circumvents this problem by performing a much cheaper partial optimization of q_{θ_n} using the linear conjugate gradient algorithm (CG).

CG is a non-stationary iterative algorithm for minimizing quadratic functions of the form $\phi(z) = z^\top \mathrm{B} z/2 + b^\top z$ given an arbitrary starting point $z = z_0$. It is invoked within HF approaches by setting the initial vector z_0 to δ_{n-1}, the linear term b to $-f'(\theta_n)$, and the curvature matrix B to $\mathrm{C}_n + \lambda I$. While CG will in general require N steps to converge, it has very strong partial convergence properties and in practice tends to make substantial progress on the optimization of q_{θ_n} in a number of iterations far less than N. Moreover, CG does not require direct access to the curvature matrix B, and instead only needs a black-box function which can compute matrix-vector products $\mathrm{B}v$ for arbitrary v. Fortunately, there is an efficient, general and numerically-stable algorithm for computing the Hessian-vector products (Nocedal and Wright, 1999, ch. 7) which was first applied to machine learning models like RNNs by Pearlmutter (1994).

By utilizing the complete curvature information given by B, CG running within HF can take large steps in directions of low reduction and curvature which are effectively invisible to gradient descent while respecting the constraints imposed by the directions of high curvature.

Although the HF approach has been known and studied for decades within the optimization literature, the shortcomings of existing versions of the approach made them impractical or even completely ineffective for neural network training (Martens, 2010). The version of HF developed by Martens (2010) makes a series of important modifications and design choices to the basic approach, which include (among others): using the positive semi-definite Gauss-Newton curvature matrix in place of the possibly indefinite Hessian, using a quadratic damping function for $R_{\theta_n}(\theta) = \|\delta_n\|^2/2$ with the damping parameter λ adjusted by Levenberg-Marquardt style heuristics (More, 1978), using a criterion based directly on the value of q_{θ_n} in order to terminate CG (as opposed to the usual residual-error $\|\mathrm{B}z - b\|^2/2$ based criterion), and computing the curvature-matrix products $\mathrm{B}v$ using mini-batches (as opposed to the whole training set) and the gradients with much larger mini-batches.

In applying HF to RNNs we deviate from the approach of Martens (2010) in two significant ways. First, we use a different damping function for R (developed in section 4.2.3), which we found improves the overall robustness of the HF algorithm on RNNs by more accurately identifying regions where M_{θ_n} is likely to deviate significantly from f. Second, we do not use diagonal preconditioning because we found that it gave no substantive benefit for RNNs, the likely explanation for this being that RNNs do not exhibit obvious "axis-aligned" parameter scaling issues.

Algorithm 5 The Hessian-free optimization method (simplified version)

for each n, assign curvature and gradient mini-batches $\{U_n\}$ and $\{V_n\}$ (resp.)
$\delta_0 \leftarrow 0$
initialize λ
for all $n = 1, 2, \ldots$ **do**
  define the function to compute the matrix-vector product $\mathrm{B}v$ as $\frac{1}{|U_n|} \sum_{(x,y) \in U_n} \mathrm{G}_f((x,y);\theta_n)v + \lambda v$
  compute $b = -\frac{1}{|V_n|} \sum_{(x,y) \in V_n} f'((x,y);\theta_n)$
  find δ_n by partially minimizing $z^\top \mathrm{B}z/2 + b^\top z$ via CG with $z_0 = \delta_{n-1}$
  adjust λ using standard Levenburg-Marquardt heuristics (see More (1978))
  compute α via a standard back-tracking line search (see Nocedal and Wright (1999))
  $\theta_{n+1} \leftarrow \theta_n + \alpha\delta_n$
end for

4.2.1 The Levenberg-Marquardt Heuristic

The Levenberg-Marquardt heuristic (e.g., More (1978)) updates λ by monitoring the accuracy of the quadratic approximation. Specifically, given a parameter setting θ_n and a parameter update δ_n computed by CG with damping λI, the reduction ratio ρ is defined as

$$\rho = \frac{f(\theta_n + \delta_n) - f(\theta_n)}{q_{\theta_n}(\delta_n)} \tag{4.3}$$

Once ρ is calculated, λ is updated as follows (possibly with slightly different constants):

1. If $\rho > 3/4$ then $\lambda \leftarrow 2/3\lambda$
2. If $\rho < 1/4$ then $\lambda \leftarrow 3/2\lambda$

Thus the Levenberg-Marquardt heuristic reduces λ if the quadratic approximation q_{θ_n} accurately predicts the reduction in the objective, and increases λ otherwise.

4.2.2 Multiplication by the Generalized Gauss-Newton Matrix

We now define the aforementioned Gauss-Newton matrix G_f, discuss its properties, and describe the algorithm for computing the curvature matrix-vector product $\mathrm{G}_f v$ (which was first applied to neural networks by Schraudolph (2002), extending the work of Pearlmutter (1994)). We will use the notation from sec. 2.5 which describes the forward pass of an RNN:

input: v
for t **from** 1 **to** T **do**
  $u_t \leftarrow W_{hv} v_t + W_{hh} h_{t-1} + b_h$
  $h_t \leftarrow e(u_t)$
  $o_t \leftarrow W_{oh} h_t + b_o$
  $z_t \leftarrow g(o_t)$
end for
output: z

where an index-less variable (e.g., o) is the concatenation of its values through time (e.g., $\mathrm{concat}(o_1, \ldots, o_T)$).

The Gauss-Newton matrix for a single training example (v, y) is given by

$$\mathrm{G}_f \equiv \mathrm{J}_{o,\theta}^\top (L \circ g)'' \mathrm{J}_{o,\theta} \Big|_{\theta=\theta_n} \tag{4.4}$$

where the $\mathrm{J}_{o,\theta}$ is the Jacobian of o w.r.t. θ, and $(L \circ g)''$ is the Hessian of the function $L(g(o); y)$ with respect to o. To compute the curvature matrix-vector product $\mathrm{G}_f v$ we first compute $\mathrm{J}_{o,\theta} v = \mathrm{R}o$ products using the "$R\{\}$-method" given in Pearlmutter (1994) and sec. 2.3. Next, we run standard backpropagation through time with the vector $(L \circ g)'' \mathrm{J}_{o,\theta} v$ which accomplishes the multiplication by $\mathrm{J}_{o,\theta}^\top$:

$$\mathrm{J}_{o,\theta}^\top ((L \circ g)'' \mathrm{J}_{o,\theta} v)\Big|_{\theta=\theta_n} = (\mathrm{J}_{o,\theta}^\top (L \circ g)'' \mathrm{J}_{o,\theta})\Big|_{\theta=\theta_n} v = \mathrm{G}_f v$$

If o and the hidden state sequence h is precomputed then the total cost of this operation is the same as a standard forward pass and back-propagation through time operation — i.e., it is essentially linear in the number of parameters.

A very important property of the Gauss-Newton matrix is that it is positive semi-definite when $(L \circ g)''$ is, which happens precisely when $L(z(o); y)$ is convex w.r.t. o. The usual view of the Gauss-Newton matrix is that it is simply a positive semi-definite approximation to the Hessian. There is an alternative view, which is usually seen in the context of non-linear least-squares problems, which we now extended to encompass Schraudolph's generalized Gauss-Newton formulation. Consider the approximation $\tilde{f}$ of f obtained by "linearizing" the the network up to the activations of the output units o, after which the final nonlinearity g and the usual error function are applied. To be precise, we replace $o(\theta)$ with $o(\theta_n) + \mathrm{J}_{o,\theta}|_{\theta=\theta_n} \delta_n$ (recall $\delta_n \equiv \theta - \theta_n$), where $\mathrm{J}_{o,\theta}$ is the Jacobian of o w.r.t. to θ, giving raise to the approximation

$$\tilde{f}_{\theta_n}(\theta) = L\left(g(\, o(\theta_n) + \mathrm{J}_{o,\theta}|_{\theta=\theta_n} \delta_n \,) \;;\; y \right) \tag{4.5}$$

Because the function $\mathrm{J}_{o,\theta}\delta_n$ is affine in θ, it follows that $\tilde{f}$ is convex when $L \circ g$ is (since it will be the composition of a convex and an affine function), and so its Hessian will be positive semi-definite. A sufficient condition for $L \circ g$ being convex is when the output non-linearity function g 'matches' L (Schraudolph, 2002), as is the case when g is the logistic function and L the cross-entropy error. Note that the choice to linearize up to o is arbitrary in a sense, and we could instead linearize up to z (for example), as long as L is convex w.r.t. z.[1] However, it makes sense to linearize as close to the parameters θ as possible to achieve the most faithful approximation to f, as linearizing at θ recovers the original function f.

In the case of non-linear least-squares, once we linearize the network, $L \circ g$ is simply squared L_2 norm and so $\tilde{f}$ is already quadratic in θ. However, in the more general setting we only require that $L \circ g$ be convex and twice differentiable. To obtain a (convex) quadratic we can simply compute the 2nd-order Taylor series approximation of $\tilde{f}$.

The gradient of $\tilde{f}(\theta)$ is given by

$$\nabla_o L(g(o(\theta_n) + \mathrm{J}_{o,\theta}|_{\theta=\theta_n} \delta_n); y)\ \mathrm{J}_{o,\theta}|_{\theta=\theta_n} \tag{4.6}$$

The 1st-order term in the quadratic approximation of $\tilde{f}$ centered at θ_n is the gradient of $\tilde{f}$ as computed above, evaluated at $\theta = \theta_n$ (i.e. at $\delta_n = 0$), and is

$$\nabla_o L(g(o(\theta_n)); y)\ \mathrm{J}_{o,\theta}|_{\theta=\theta_n}$$

which is equal to the gradient of the exact f evaluated at $\theta = \theta_n$. Thus the 1st-order terms of the quadratic approximations to f and $\hat{f}$ are identical.

[1] It is also possible to linearize up to L, which causes the Gauss-Newton matrix to be equal to the Empirical Fisher Information $|U_n|^{-1} \sum_{(x,y)\in U_n} \left[\nabla f'((x,y),\theta_n) \nabla f'((x,y),\theta_n)^\top\right]$, but this matrix was shown to yield inferior performance (Martens, 2010)

Computing the Hessian of $\tilde{f}$ we obtain:

$$\mathrm{J}_{o,\theta}^{\top}\Big|_{\theta=\theta_n} (L \circ g)'' \left(o(\theta_n) + \mathrm{J}_{o,\theta}|_{\theta=\theta_n} \delta_n \right) \mathrm{J}_{o,\theta}|_{\theta=\theta_n}$$

The 2nd-order term of the Taylor-series approximation of $\tilde{f}$ is this quantity evaluated at $\theta = \theta_n$, i.e., at $\delta_n = 0$. Unlike the first-order term, it is not the same as the corresponding term in the Taylor series of f, but is in fact the Gauss-Newton matrix G_f of f. Thus we showed that the quadratic model of f which uses the Gauss-Newton matrix in place of the Hessian is in fact the Taylor-series approximation to $\tilde{f}$.

Martens (2010) found that using the Gauss-Newton matrix G_f instead of H within the quadratic model for f was highly preferable for several reasons. Firstly, because H is in general indefinite, the sub-objective q_{θ_n} may not even be bounded below when $\mathrm{B} = \mathrm{H} + \lambda \mathrm{I}$. In particular, infinite reduction will be possible along any directions of negative curvature that have a non-zero inner product with the gradient. While this problem can be remedied by choosing λ to be larger in magnitude than the most-negative eigenvalue of H, there is no efficient way to compute this in general, and moreover, using large values of λ can have a highly detrimental effect on the performance of HF as it effectively masks out the important low curvature directions. Secondly, Martens (2010) made the qualitative observation that, even when putting the issues of negative curvature aside, the parameter updates produced by using G_f in place of H performed much better in practice in the context of neural network learning.

4.2.3 Structural Damping

While we have found that applying the HF approach of Martens (2010) to RNNs achieves robust performance for *most* of the pathological long-term dependency problems (section 4.3) without any significant modification, for truly robust performance on all of these problems for 100 timesteps and beyond we found that is was necessary to incorporate an additional idea which we call "structural damping" (named so because it is a damping strategy that makes use of the specific structure of the objective function).

In general, 2nd-order optimization algorithms, and HF in particular, perform poorly when the quadratic approximation M_{θ_n} becomes highly inaccurate as δ_n approaches the optimum of the quadratic sub-objective q_{θ_n}. Damping, which is critical to the success of HF, encourages the optimal point to be close to $\delta_n = 0$, where the quadratic approximation becomes exact but trivial. The basic damping strategy used by Martens (2010) is the classic Tikhonov regularization, which penalizes δ_n by $\lambda R(\delta_n)$ for $R(\delta_n) = \frac{1}{2}\|\delta_n\|^2$. This is accomplished by adding $\lambda \mathrm{I}$ to the curvature matrix B.

Our initial experience training RNNs with HF on long-term dependency tasks was that when λ was small there would be a rapid decrease in the accuracy of the quadratic approximation M_{θ_n} as δ_n was driven by CG towards the optimum of q_{θ_n}. This would cause the Levenburg-Marquardt heuristics to (rightly) compensate by adjusting λ to be much larger. Unfortunately, we found that such a scenario was associated with poor training outcomes on the more difficult long-term dependency tasks of Hochreiter and Schmidhuber (1997). In particular, we saw that the parameters seemed to approach a bad local minimum where little-to-no long-term information was being propagated through the hidden states.

One way to explain this observation is that for large values of λ any Tikhonov-damped 2nd-order optimization approach (such as HF) will behave similarly to a 1st-order approach[2] and crucial low-curvature directions whose associated curvature is significantly smaller than λ will be effectively masked-out. Martens (2010) found that the Tikhonov damping strategy was very effective for training deep auto-encoders, presumably because during any point in the optimization, there was a value of λ which would allow M_{θ_n} to remain an accurate approximation of f (at q_{θ_n}'s optimum) without being so high

[2] it can be easily shown that as $\lambda \to \infty$, minimizing q_{θ_n} is equivalent to a small step of gradient descent.

as to reduce the method to mostly 1st-order approach. Unfortunately our experience seems to indicate that is less the case with RNNs.

Our hypothesis is that for certain small changes in θ there can be large and highly non-linear changes in the hidden state sequence h (given that W_{hh} is applied iteratively to potentially hundreds of temporal 'layers') and these will not be accurately approximated by M_{θ_n}. Furthermore, preventing these large changes may also be helpful in situations where, while the changes do not negatively impact the objective, they nonetheless cause damage to the beneficial initial hidden-hidden dynamics determined by the random initial parameters. The most obvious example is when the hidden-output connections are weak and the hidden-hidden dynamics are not being strongly used. In such situations, the optimizer may be too aggressive and make a change to the parameters which cause the dynamics to become trivial, or completely chaotic. In some sense, we would prefer the RNN to behave a bit like an echo-state-network (Jaeger and Haas, 2004) during the initial stages of optimization. These ideas motivate the development of a damping function R that can penalize directions in parameter space that despite not being large in magnitude, can nonetheless lead to large changes in the hidden-state sequence. With such a damping scheme we can penalize said directions without requiring λ to be so high as to prevent the strong pursuit of other low-curvature directions whose effect on f are modeled more accurately. To this end we define the "structural damping function" as

$$R_{\theta_n}(\theta) = \frac{1}{2}\|\delta_n\|^2 + \mu S_{\theta_n}(\theta) \tag{4.7}$$

where $\mu > 0$ is a weighting constant and $S_{\theta_n}(\theta)$ is a function which quantifies change in the hidden units as a function of the change in parameters. Recalling that $h(\theta)$ is the hidden state sequnece of an RNN with parameters θ, the function S is given by

$$S_{\theta_n}(\theta) \equiv D(\, h(\theta)\, ;\, h(\theta_n)\,),$$

where D is a distance function similar to the loss L. Note that we still include the usual Tikhonov term within this new damping function and so just as before, if λ becomes too high, all of the low curvature directions become essentially "masked out". However, with the inclusion of the $\mu S_{\theta_n}(\theta)$ term we hope that the quadratic model will tend to be more accurate at its optimum even for smaller values of λ. Fortunately this is what we observed in practice, along with improved performance on the pathological long-term dependency problems.

When the damping function $R_{\theta_n}(\theta)$ is given by $\|\delta_n\|^2/2$, the function q_{θ_n} is a quadratic with respect to δ_n and thus CG can be directly applied with $\mathrm{B} = \mathrm{G}_f + \lambda I$ in order to minimize it. However, CG will not work with the structural damping function because S is not generally quadratic in θ. Thus, just as we approximated f by the quadratic function M, so too must we approximate S.

One option is to use the standard Taylor-series approximation:

$$S(\theta_n) + (\theta - \theta_n)^\top \left.\frac{\partial S}{\partial \theta}\right|_{\theta=\theta_n} + \frac{1}{2}(\theta - \theta_n)^\top \mathrm{H}_S(\theta - \theta_n)$$

where H_S is the Hessian of S at θ_n. Note that because D is a distance function, $D(h(\theta), h(\theta_n))$ is minimized at $\theta = \theta_n$ with value 0. This implies that

$$D(h(\theta), h(\theta_n)) = 0 \quad \text{and} \quad \frac{\partial D(h(\theta), h(\theta_n))}{\partial h(\theta)} = 0$$

Using these facts we can eliminate the constant and linear terms from the quadratic approximation of S since $S(\theta_n) = D(h(\theta_n), h(\theta_n)) = 0$ and

$$\left.\frac{\partial S}{\partial \theta}\right|_{\theta=\theta_n} = \left.\frac{\partial D(h(\theta), h(\theta_n))}{\partial \theta}\right|_{\theta=\theta_n} = \left.\frac{\partial D(h(\theta), h(\theta_n))}{\partial h(\theta)}\frac{\partial h(\theta)}{\partial \theta}\right|_{\theta=\theta_n} = 0$$

The fact that the gradient of the damping term is zero is important since otherwise including it with q_{θ_n} would result in a biased optimization process that could no longer be said to be optimizing the objective f.

Moreover, just as we did when creating a quadratic approximation of f, we may use the Gauss-Newton matrix G_S of S in place of the true Hessian. In particular, we define:

$$\mathrm{G}_S \equiv \mathrm{J}_{x,\theta}^\top (D \circ e)'' \mathrm{J}_{x,\theta} \Big|_{\theta=\theta_n}$$

where $\mathrm{J}_{x,\theta}$ is the Jacobian of $x(\theta)$ w.r.t. θ (recall sec. 2.5: $h_i = e(u_i)$ and $(D \circ e)''$ is the Hessian of $D(e(x); h(\theta_n))$ w.r.t. x). This matrix will be positive semi-definite if the distance function D matches the non-linearity e in the same way that we required L to match g in order to obtain the Gauss-Newton matrix for f.

And just as with G_f we may view the Gauss-Newton matrix for S as resulting from the Taylor-series approximation of the following convex approximation $\tilde{S}$ to S

$$\tilde{S} = D(\, e(x(\theta_n) + \mathrm{J}_{x,\theta}|_{\theta=\theta_n} \delta_n) \,;\, h(\theta_n) \,)$$

What remains is to find an efficient method for computing the contribution of the Gauss-Newton matrix for S to the usual curvature matrix-vector product to be used within the HF optimizer. While it almost certainly would require a significant amount of additional work to compute both $\mathrm{G}_f v$ and $\lambda\mu \mathrm{G}_S v$ separately, it turns out that we can compute their sum $\mathrm{G}_f v + \lambda\mu \mathrm{G}_S v = (\mathrm{G}_f + \lambda\mu \mathrm{G}_S)v$, which is what we actually need, for essentially the same cost as just computing $\mathrm{G}_f v$ by itself.

We can formally include $\lambda\mu S$ as part of the objective, treating $h = e(x)$ as an additional output of the network so that we linearize up to x in addition to o. This gives the combined Gauss-Newton matrix:

$$\mathrm{J}_{(o,x),\theta}^\top \Big|_{\theta=\theta_n} \begin{pmatrix} L \circ g & 0 \\ 0 & D \circ e \end{pmatrix}'' \mathrm{J}_{(o,x),\theta} \Big|_{\theta=\theta_n}$$

Because the linearized prediction of x is already computed as an intermediate step when computing the linearized prediction of o, the required multiplication by the Jacobian $\mathrm{J}_{(o,x),\theta}|_{\theta=\theta_n}$ requires no extra work. Moreover, as long as we perform backpropagation starting from both o and x together (as we would when applying reverse-mode automatic differentiation) the required multiplication by the transpose Jacobian term does not either. The only extra computation required will be the multiplication by $(D \circ e)''$, but this is very cheap compared to the weight-matrix multiplication.

Equivalently, we may just straightforwardly apply Pearlmutter's technique for computing Hessian-vector products directly to any code which computes $\tilde{f} + \lambda\mu\tilde{S}$, and as long as said code does not perform an unnecessary double computation of x, the result will require essentially no more work than if it were just applied to $\tilde{f}$. And of course said code can itself be generated by applying the R-operator to the augmented objective $f + \lambda\mu S$, stopping short of linearizing past x (see Appendix).

The result of applying either of these techniques is given as Algorithm 6 (see appendix) for specific choices of neural activation functions and error functions. Note that the required modification to the usual algorithm for computing the curvature matrix-vector product is simple and elegant and requires virtually no additional computation. However, it does require the extended storage of the $\mathrm{R}u_i$'s, which could otherwise be discarded immediately after they are used to compute $\mathrm{R}h_i$.

While we have derived the method of 'structural damping' specifically for the hidden unit inputs x in the RNN model, it is easy to see that the idea and derivation generalize to any intermediate quantity x in any parameterized function we wish to learn. Moreover, the required modification to the associated matrix-vector product algorithm for computing the damped Gauss-Newton matrix would similarly be

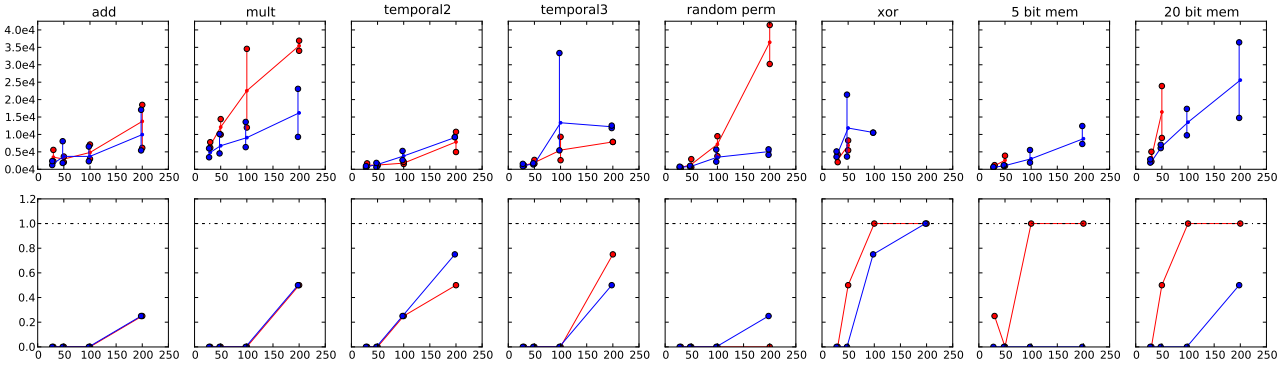


Figure 4.1: For each problem, the plots in the top row show the mean, min, and max number of minibatches processed (y-axis) required to solve the problem for various values of T (x-axis), where a minibatch is a set of 1000 sequences of length T that may be used for evaluating the gradient or a curvature matrix-vector product. A run using over 50,000 minibatches is considered a failure. The failure rates are shown in the bottom row. The red plots correspond to regular damping and the green plots to structural damping.

to add $\lambda\mu(D \circ e)'' Rx$ to $R\{\partial\tilde{k}/\partial x\}$ (see Algorithm 6) where D is the distance function that penalizes change in z and $\tilde{k}$ is the damped objective.

Because S is a highly non-linear function of δ_n it could be the case that the quadratic model will occasionally underestimate (or overestimate) the amount of change, perhaps severely. One might even be tempted to think that this technique would be useless, since if the linear model for the hidden units h is accurate for a particular δ_n then we would not need to penalize change in h, and if it is not accurate then structural damping may not help because it uses the same linearized model for h. While we cannot completely overcome these objections, we can argue why they might not be too important in practice. Firstly, even when the hidden states (which are high-dimensional vectors) are not accurately predicted by the model it could still be the case that the overall measure of their change, which is only a scalar, still will be. Secondly, while the amount of change may be over or underestimated for any particular training case, we are averaging our computations over many (usually 1000s) training cases and thus there may be some cancellation. Thirdly, while a large value for the quadratic approximation of S is not a necessary condition for the linear model of h to be inaccurate, it is at least a sufficient condition.

4.3 Experiments

In our first set of experiments we trained RNNs, using our HF-based approach, on various pathological synthetic problems that for a long time were widely believed to be hopelessly impossible for gradient descent Hochreiter et al. (2001). These problems were taken directly from the original LSTM paper of Hochreiter and Schmidhuber (1997).

In our second set of experiments we trained both RNNs (using HF) and LSTMs (using backprop-through-time) on more realistic high-dimensional time-series prediction tasks. In doing so we examine the hypothesis that the specialized LSTM approach, while being effective at tasks where the main difficulty is the need for long-term memorization and recall, may not be fully exploiting the power of recurrent computation in RNNs, and that our more general HF approach can do better.

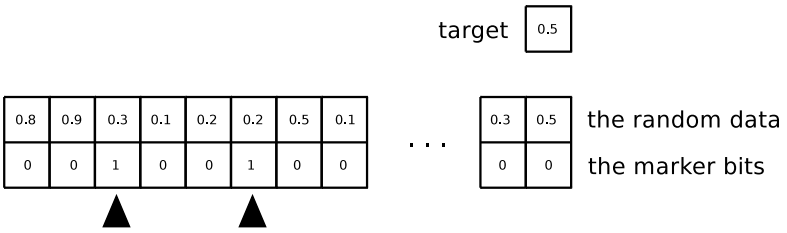


Figure 4.2: An illustration of the addition problem, a typical problem with pathological long-term dependencies. The target is the sum of the marked numbers and is independent of the unmarked ones. The ellipsis "..." represents a large number of timesteps.

4.3.1 Pathological synthetic problems

In this section, we train RNNs on seven problems that exhibit pathological long-term dependencies. All of these problems have a similar form, where the inputs are long sequences whose start (and possibly middle) are relevant, and the goal is to output a particular function of the relevant inputs at the final few time-steps. The irrelevant part of the sequence is usually random, which makes the problem harder. A typical problem is illustrated in figure 4.2. The difficulty of these problems increases with T (the sequence length), since longer sequences exhibit longer range dependencies. Problems 1-6 are taken from Hochreiter and Schmidhuber (1997), so we will move their description to the appendix.

In our experiments, we trained RNNs with 100 hidden units using HF, with and without structural damping, on problems 1-7 for $T = 30, 50, 100, 200$. Our goal is to determine the amount of work the optimizer requires to train the RNN to solve each problem. To decide if the optimizer has succeeded we use the same criterion as Hochreiter and Schmidhuber (1997).

The addition problem: In this problem, the input to the RNN is a sequence of random numbers, and its target output, which is located at the end of the sequence, is the sum of the two "marked" numbers. The marked numbers are far from the sequence's end, their position varies, and they are distant from each other. We refer to Hochreiter and Schmidhuber (1997) for a formal description of the addition problem, and to fig. 4.2 for an intuitive illustration.

This problem and others like it are challenging not only because of their obvious long-term dependencies, but also because the RNN's hidden state, which is exposed to both the relevant and irrelevant inputs in the same manner (since they share the same input unit v_t), must learn to accurately remember one and ignore the other. These two conditions are so contradictory as to appear unimplementable by a standard RNN, which, unlike an LSTM, does not have a built-in means to protect its hidden state from irrelevant inputs. Yet, as we will see, our methods can solve this and related tasks for T as large as 200.

The multiplication problem: The multiplication problem is precisely analogous to the addition problem except for the different operation.

The Xor problem: This problem is similar to the addition problem, but the noisy inputs are binary and the operation is the Xor. This task is difficult for both our method and for LSTMs (Hochreiter and Schmidhuber, 1997, sec. 5.6) because the RNN cannot obtain a partial solution by discovering a relationship between one of the marked inputs and the target.

The temporal order problem: See Hochreiter and Schmidhuber (1997), task 6a.

The 3-bit temporal order problem: See Hochreiter and Schmidhuber (1997), task 6b.

The random permutation problem: See Hochreiter and Schmidhuber (1997), task 2b.

Noiseless memorization: The goal of this problem is to learn to memorize and reproduce long sequences of bits. The input sequence starts with a string of 5 bits followed by T occurrences of a

constant input. There is a target in every timestep, which is constant, except in the last 5 timesteps, which are the original 5 bits. There is also a special input in the 5th timestep before last signaling that the output needs to be presented. We also experimented with a harder variant of this problem, where the goal is to memorize and sequentially reproduce 10 random integers from $\{1, \ldots, 5\}$ which contain over 20 bits of information.

4.3.2 Results and discussion

The results from our experiments with the pathological synthetic problems are shown in figure 4.1. From these it is clear that the HF algorithm is capable of training RNNs to solve problems with very long-term dependencies. Moreover, the inclusion of structural damping adds some additional speed and robustness (especially in the 5 and 20 bit memorization tasks, where it appears to be required when the minimal time-lag is larger than 50 steps). For some random seeds the exploding/vanishing gradient problem seemed to be too extreme for the HF optimizer to handle and the parameters would converge to a bad local minimum (these are reported as failures in figure 4.1). This is a problem that can likely be addressed with a more careful random initialization than the one we used.

It should be noted that the total amount of computation required by our HF approach in order to successfully train an RNN is considerably higher than what is required to train the LSTM using its specialized variant of stochastic gradient descent. However, the LSTM architecture was designed specifically so that long-term memorization and recall can occur naturally via the "memory units", so representationally it is a model ideally suited for such tasks. In contrast, the problem of training a standard RNN to succeed at these tasks using only its general-purpose units, which lack the convenient "gating neurons" (a type of 3-way neural connection), is a *much* harder one, so the extra computation required by the HF approach seems justified.

4.3.3 The effect of structural damping

While structural damping was only slightly beneficial for problems 1-6, it was essential for the 5-bit and 20-bit memorization problems. We believe this is the case partly because of the additional work the hidden state is required to do. Specifically, to succeed on the 5-bit memorization problem, it is not enough for the hidden state to simply carry all the information about the input (which, incidentally, is trivial to arrange with an ESN-like initialization that forces the hidden state to be different for every (input, timestep) pair due to the noiseless nature of the problem). In addition to carrying the information about the input, the hidden state must also change rapidly and precisely, when making the predictions in the end of the sequence, to enable the hidden-to-output connections to read them out. Rapidly changing hidden states can be seen in the video in `www.cs.utoronto.ca/~ilya/2011/HF-RNN_video.zip`, which shows the evolution of the hidden state sequences as learning progresses for 6 inputs of this problem. Dynamics of this kind, where the hidden state changes on its own, without being driven to do so by the inputs, are likely less important for problems 1-6, because all these problems make their important prediction in precisely one timestep per sequence.

4.3.4 Natural problems

For these problems, the output sequence is equal to the input sequence shifted forward by one time-step, so the goal of training is to have the model predict the next timestep by either minimizing the squared error, a KL-divergence, or by maximizing the log likelihood. We trained an RNN with HF and an LSTM with backpropagation-through-time (verified for correctness by numerical differentiation). The RNN had 300 hidden units, and the LSTM had 30 "memory blocks" (each of which has 10 "memory

Dataset (measure)	RNN+HF	LSTM+GD
Bouncing balls (error)	22	35
MIDI (log-likelihood)	-569	-683
Speech (error)	22	41

Table 4.1: Average test-set performance for HF-trained RNN and the LSTM on the three natural sequence modeling problems.

cells"); this results in nearly identically-sized parameterizations for both models (the LSTM had slightly more). Full details of the datasets and of the training are given in the supplementary material.

Bouncing ball motion video prediction: The bouncing balls problem consists of synthetic low-resolution (15×15) videos of three balls bouncing in a box. We trained the models on sequences of length 30.

Music prediction: We downloaded a large number of MIDI files and converted each into a "piano-roll". A piano-roll is a sequence of 128-dimensional binary vectors, each describing the set of notes played in the current timestep (which is 0.05 seconds long). The models were trained on sequences of length 200 to model the significant long-term dependencies exhibited by music.

Speech prediction: We used a speech recognition dataset to obtain sequences of 39-dimensional vectors representing the underlying speech signal, and used sequences of length 100.

Results

Results are shown in Table 4.1. The HF-trained RNNs outperform the LSTMs trained with gradient descent by a large margin, and the results are similar when rerun.

Appendix

4.4 Details of the Pathological Synthetic Problems

We begin by describing the experimental setup that was used in all the pathological problems. In every experiment, the RNN had 100 hidden units and a little over 10,000 parameters. It was initialized with a sparse initialization (Martens, 2010): each weight matrix (W_{hv}, W_{hh}, and W_{ho}) is sparsely initialized so that each unit is connected to 15 other units. The nonzero connections of W_{hh}, of W_{ho}, and the biases are sampled independently from a Gaussian with mean 0 and variance $\frac{1}{15}$, while the nonzero connections of W_{hv} are independently sampled from a Gaussian with mean 0 and a variance of 1. These constants were chosen so that the initial gradient would not vanish or explode too strongly for the optimizer to cope with. The occasional failures for some random seeds of the HF approach on a few of the harder synthetic problems was likely due to a deficiency in this initialization scheme.

The gradient was computed on 10,000 sequences, of which 1,000 were used to evaluate the curvature matrix-vector products (except for the 5-bit version of problem 7, where there are only 32 possible distinct sequences, so the gradient and the curvature matrix-vector products were computed using all the data). Every HF iteration used a fresh set of sequences and the test set also consists of 10,000 sequences. We allowed for at most 300 steps of CG at each run of HF, but we would stop the CG run sooner if the magnitude of the reduction in the objective function on the curvature minibatch is less than 90% of its maximum during the CG run. The damping coefficient λ was initialized to 0.1 if structural damping was used in which case μ was set to $1/30$, and to 0.3 otherwise.

The criteria for success and failure we used were taken directly from Hochreiter and Schmidhuber (1997). In particular, a run is defined to be successful if less than 1% of the test sequences are misclassified. While the notion of sequence misclassification is trivially defined for binary or discrete problems, for continuous problems (such as the addition or the multiplication problem), we say that a sequence is misclassified if the absolute prediction error exceeds 0.04.

In some of the problems, the inputs or the outputs are symbols. Input symbols are simply represented with their 1-of-N encoding. To predict a symbol, the RNN uses the softmax output nonlinearity with the cross entropy loss function, whose composition is "matching" and therefore the Gauss-Newton matrix can be used.

In the following subsections, we let $U[a, b]$ denote a uniformly random real number from the interval $[a, b]$ and $i[a, b]$ denote a uniformly random integer from the same interval. The descriptions also assume that a desired problem length T has been selected.

4.4.1 The addition, multiplication, and XOR problem

The addition problem is described in Figure 2 of the chapter. The inputs consist of sequences of random numbers, and the target output, which is located in the end of the sequence, is the sum of the two "marked" numbers. The marked numbers are far from the sequence's end, their position varies, and they are distant from each other.

To generate a single training sequence for the addition problem, let the length of the sequence T' be a sample from $i[T, 11T/10]$ (so that $T' > T$). Let I and J be the positions of the marked inputs, which are drawn from $i[1, T'/10]$ and let $i[T'/10, T'/2]$. The input x will consist of the pairs (u, v) where the u_t's are independently drawn from $U[0, 1]$ and the "markers" u'_t will be 0, except at $t = I$ and J, where $u'_I = u'_J = 1$. Finally, let the target output at time T' be the normalized sum $(u_I + u_J)/2$. Thus the u_t component of the input $v_t = (u_t, u'_t)$ is irrelevant when t is not I or J.

The multiplication and the XOR problem are completely analogous.

4.4.2 The temporal order problem

In this task (Hochreiter and Schmidhuber, 1997, task 6a), the input sequences consist of random symbols in $\{1, 2, ..., 6\}$ which are represented with 1-of-6 encodings. All of the inputs are irrelevant and are the encodings of randomly chosen integers in $\{3, 4, 5, 6\}$ except for 2 special ones which are encodings of randomly chosen integers in $\{1, 2\}$. The target output is a 1-of-4 encoding of the ordered-pair of the two special symbols.

Formally, let I and J be the positions of the special inputs which are drawn from $i[T/10, 2T/10]$ and $i[5T/10, 6T/10]$ respectively. We let $c_1, \ldots, c_T$ be independent draws from $i[3, 6]$, and let a, b, the special symbols, be independently drawn from $i[1, 2]$. Let e_j denote the 1-of-6 encoding of j. Then we set the input v_t to e_{c_t} for $t \neq I$ or J, and set v_I to e_a and v_J to e_b. The target output at time T is the pair (a, b) which is represented with a 1-of-4 encoding (because there are four possibilities for (a, b)).

4.4.3 The 3-bit temporal order problem

This problem is similar to the above, except that there are 3 special symbols in the input sequence, and thus 8 possible outputs (Hochreiter and Schmidhuber, 1997, task 6b).

The formal setup is nearly identical to the previous one. The positions I, J, and K of the three special symbols are drawn from $i[T/10, 2T/10]$, $i[3T/10, 4T/10]$, and $i[6T/10, 7T/10]$, respectively. The values of the special symbols a, b, c are independently drawn from $i[1, 2]$. Finally, $v_I = e_a, v_J = e_b, v_K = e_c$, and the target is (a, b, c) represented with a 1-of-8 encoding.

4.4.4 The random permutation problem

The inputs consist of sequences of symbols from $\{1, \ldots, 100\}$. The first and the last symbol are identical and their value is drawn from $i[1, 2]$, while the rest of the symbols are independently drawn from $i[3, 100]$. At each timestep, the target output is equal to the input symbol of the next timestep, and as a result only the first and the last symbol are predictable. This problem is difficult not only for its long-term dependencies, but also because the loss incurred at the last timestep (the one we use to measure success) is small compared to the loss associated with the other timesteps, which may cause the optimizer to focus on an incorrect aspect of the problem.

In this task, we initialized λ to $0.1 \cdot T$ because there are many more outputs which make the scale of the objective is considerably larger.

4.4.5 Noiseless memorization

In this problem, the input sequence starts with a string of 5 bits followed by T occurrences of a constant input. The target output is constant up until the last 5 timesteps, which are the original 5 input bits. There is also a special input in the 5th-last timestep signaling that the output needs to be presented.

In the following, inputs and the outputs are represented by 1-of-K vector encodings but for brevity we describe them as symbols. More formally, let $v_1, \ldots, v_{T+10}$ be the input symbols and $y_1, \ldots, y_{T+10}$ be the target symbols. The sequence begins with $v_1, \ldots, v_5$ which are drawn from $i[1, 2]$ and are thus binary. The subsequent inputs are constant except at $t = T + 5$, which is a "trigger" symbol (set to 4) which signals that the memorized symbols should be produced as output. The target outputs are obtained by setting $y_j = 3$ for $j \leq T + 5$, and $y_{j+T+5} = v_j$ for $1 \leq j \leq 5$. In particular, most of the predictions (up to timestep $T + 5$) are irrelevant.

We straightforwardly adapt the problem of memorizing 10 symbols from $\{1, \ldots, 5\}$.

4.5 Details of the Natural Problems

In all these problems, the output sequence is equal to the input sequence shifted forward by one timestep, so the goal of training is to have the model predict the next timestep by either minimizing the squared error, a KL-divergence, or by maximizing the log likelihood.

We now describe the details of the natural datasets and the settings of the optimizer used.

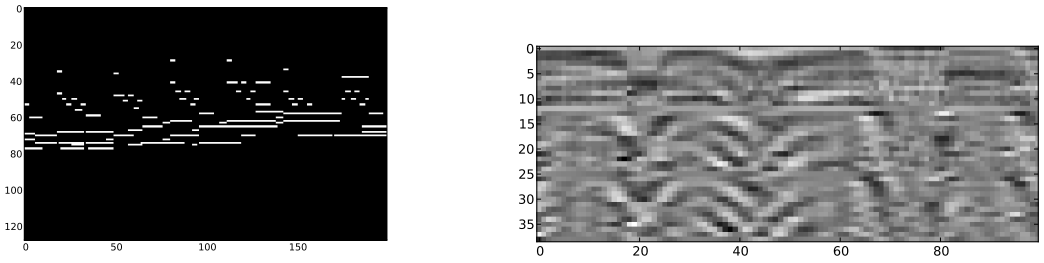
4.5.1 The bouncing balls problem

This dataset consists of synthetically generated videos of three balls bouncing in a box. Given that the sequences can be perfectly predicted using only a small number of previous timesteps, we trained both models on sequences of length 30. The resolution of the videos is 18×18.

The pixels are real values between 0 and 1, so the RNN and the LSTM are trained to minimize the per-pixel KL divergence $\sum_i q_i \log q_i/p_i + (1 - q_i) \log (1 - q_i)/(1 - p_i)$, where q_i is the ith pixel and p_i is the ith prediction (which is the output of the sigmoid function). This also is the per-sequence error measure we reported in Table 1 in the chapter.

For training of the RNN with the HF approach we used $4 \cdot 10^4$ sequences for computing the gradient and $4 \cdot 10^3$ for computing the curvature matrix-vector products. We stopped training after a total of $100 \cdot 10^6$ sequences (not necessarily distinct) had been processed. In this experiment, μ was set to 1.0 and λ was initialized to 1.

The LSTM was trained with stochastic gradient descent with a learning rate of 0.001 and a momentum of 0.9 for a total of 10^5 weight updates, where each gradient was computed as the average over a



(a) An example of a midi sequence

(b) An expamle of a speech sequence

Figure 4.3: **Top:** An example of a training sequence from the MIDI music dataset. The sequence is 200 timestep long, and each timestep is a 128-dimensional binary vector. Typical sequences from this dataset exhibit a substantial amount of nontrivial long range temporal dependencies. **Bottom:**An example of a training sequence from the speech dataset. The sequence is 100 timestep long, and each timestep is a 39-dimensional real valued vector.

minibatch of 50 training sequences. It was initialized with an analogous sparse initialization, where the nonzero entries were sampled independently from a Gaussian with mean 0 and variance 0.01. We also added 3.0 to the biases of the "forget gates" which we found was beneficial to the optimization.

4.5.2 The MIDI dataset

To construct this dataset we downloaded 3364 MIDI files from an online database and created a training set consisting of 100,000 sequences of length 200, where each timestep corresponds to the set of tones present in a 0.05-second long interval (represented as a 128-dimensional binary vector). Thus each sequence corresponds to exactly 10 seconds of music. We increased the size of the dataset by translating the tones in each sequence by an offset sampled from $i[-10, 10]$. Figure 4.3(b) depicts a typical sequence from this dataset.

Due to the binary nature of the data, we trained the RNN and LSTM to maximize the log probability of the sequences using the cross-entropy objective $\sum_i t_i \log p_i + (1 - t_i) \log(1 - p_i)$, where t_i ranges over the dimensions of the timestep and p_i is the set of predictions. For training of the RNN with the HF approach we used $2 \cdot 10^4$ sequences for computing the gradient and $2 \cdot 10^3$ for computing the curvature matrix-vector products. We stopped training after a total of $60 \cdot 10^6$ sequences had been processed. We initialized λ to 100 and set μ to 0.1. The LSTM was trained for $5 \cdot 10^4$ gradient steps with a learning rate of 0.001, where each gradient minibatch was computed on 50 sequences, and was otherwise equivalent to the previous LSTM.

4.5.3 The speech dataset

We used the TIMIT dataset (Triefenbach et al., 2010) to construct the speech timeseries. The training sequences are 100 timesteps long, where each timestep is a 39-dimensional vector of the Mel-frequency cepstral coefficients (MFCC) representation of the signal (see fig. 4.3(a) for an example sequence). As the data is real-valued, we trained the RNN and LSTM to minimize the squared error.

For training of the RNN with the HF approach we used $2 \cdot 10^4$ sequences for computing the gradient and $2 \cdot 10^3$ for computing the curvature matrix-vector products. We stopped training after a total of $3 \cdot 10^6$

had been processed. We initialized λ to 50 and set μ to 0.1. The LSTM was trained for 10^5 gradient steps with a learning rate of 0.001, where each gradient minibatch was computed on 50 sequences, and was initialized as in the other experiments.

4.6 Pseudo-code for the Damped Gauss-Newton Vector Product

Algorithm 6 Computation of the matrix-vector product of the structurally-damped Gauss-Newton matrix with the vector v, for the case when e is the $\tanh$ non-linearity, g the logistic sigmoid, D and L are the corresponding matching loss functions. The notation reflects the "convex approximation" interpretation of the GN matrix so that we are applying the R-operator to the forwards-backwards pass through the linearized and structurally damped objective $\tilde{k}$, and the desired matrix-vector product is given by $\mathrm{R}\frac{\partial \tilde{k}}{\partial \theta}$. All derivatives are implicitly evaluated at $\theta = \theta_n$. The previously defined parameter symbols W_{oh}, W_{hv}, W_{hh}, b_h, b_o, h_0 will correspond to the parameter vector θ_n if they have no super-script and to the input parameter vector v if they have the 'v' superscript. The $\mathrm{R}x$ notation follows Pearlmutter (1994), and for the purposes of reading the pseudo-code can be interpreted as merely defining a new symbol. We assume that intermediate quantities of the network (e.g., h_i) have already been computed (from θ_n). The operator $\odot$ is coordinate-wise multiplication. Line 13 (underlined) is responsible for structural damping.

1: **for** $i = 1$ to T **do**
2: $\quad \mathrm{R}u_i \leftarrow b_h^v + W_{hv}^v v_i + W_{hh}^v h_{i-1} + W_{hh}\mathrm{R}h_{i-1}$
3: $\quad \mathrm{R}h_i \leftarrow (1 + h_i) \odot (1 - h_i) \odot \mathrm{R}u_i$
4: $\quad \mathrm{R}o_i \leftarrow b_p^v + W_{oh}^v h_i + W_{oh}\mathrm{R}h_i$
5: $\quad \mathrm{R}z_i \leftarrow z_i \odot (1 - z_i) \odot \mathrm{R}o_i$
6: **end for**
7: $\mathrm{R}\frac{\partial \tilde{k}}{\partial \theta} \leftarrow 0$
8: $\mathrm{R}\frac{\partial \tilde{k}}{\partial t_{T+1}} \leftarrow 0$
9: **for** $i = T$ down to 0 **do**
10: $\quad \mathrm{R}\frac{\partial \tilde{k}}{\partial o_i} \leftarrow \mathrm{R}z_i$
11: $\quad \mathrm{R}\frac{\partial \tilde{k}}{\partial h_i} \leftarrow W_{hh}^\top \mathrm{R}\frac{\partial \tilde{k}}{\partial t_{i+1}} + W_{oh}^\top \mathrm{R}\frac{\partial \tilde{k}}{\partial o_i}$
12: $\quad \mathrm{R}\frac{\partial \tilde{k}}{\partial u_i} \leftarrow (1 + h_i) \odot (1 - h_i) \odot \mathrm{R}\frac{\partial \tilde{k}}{\partial h_i}$
13: $\quad \underline{\mathrm{R}\frac{\partial \tilde{k}}{\partial u_i} \leftarrow \mathrm{R}\frac{\partial \tilde{k}}{\partial u_i} + \lambda\mu(1 + h_i) \odot (1 - h_i) \odot \mathrm{R}u_i}$
14: $\quad \mathrm{R}\frac{\partial \tilde{k}}{\partial W_{oh}} \leftarrow \mathrm{R}\frac{\partial \tilde{k}}{\partial W_{oh}} + \mathrm{R}z_i h_i^\top$
15: $\quad \mathrm{R}\frac{\partial \tilde{k}}{\partial W_{hh}} \leftarrow \mathrm{R}\frac{\partial \tilde{k}}{\partial W_{hh}} + \mathrm{R}\frac{\partial \tilde{k}}{\partial t_{i+1}} h_i^\top$
16: $\quad \mathrm{R}\frac{\partial \tilde{k}}{\partial W_{hv}} \leftarrow \mathrm{R}\frac{\partial \tilde{k}}{\partial W_{hv}} + \mathrm{R}\frac{\partial \tilde{k}}{\partial u_i} v_i^\top$
17: $\quad \mathrm{R}\frac{\partial \tilde{k}}{\partial b_h} \leftarrow \mathrm{R}\frac{\partial \tilde{k}}{\partial b_h} + \mathrm{R}\frac{\partial \tilde{k}}{\partial u_i}$
18: $\quad \mathrm{R}\frac{\partial \tilde{k}}{\partial b_p} \leftarrow \mathrm{R}\frac{\partial \tilde{k}}{\partial b_p} + \mathrm{R}z_i$
19: **end for**

Chapter 5

Language Modelling with RNNs

5.1 Introduction

In this chapter, we use Hessian-Free optimization to train large RNNs on the problem of predicting the next character in natural text. It is an important problem, because better language models improve the compression of text files (Rissanen and Langdon, 1979), the quality of speech recognition (Dahl et al., 2012) and machine translation (Papineni et al., 2002), and make it easier for people with physical disabilities to interact with computers (Ward et al., 2000). More speculatively, achieving the asymptotic limit in text compression requires an understanding that is "equivalent to intelligence" (Hutter, 2006). Good compression can be achieved by exploiting simple regularities such as the vocabulary and the syntax of the relevant languages and the shallow associations exemplified by the fact that the word "milk" often occurs soon after the word "cow", but beyond a certain point any improvement in performance must result from a deeper understanding of the text's meaning.

Prior to our work, Recurrent Neural Networks were applied to work-level language modelling and trained using truncated backpropagation through time (Mikolov et al., 2010, 2011, 2009). They achieved real improvements over state-of-the-art word-level language models, which were substantial when the RNNs were averaged with an N-gram word-level language model.

Although standard RNNs are very expressive, we found that achieving competitive results on character-level language modeling required the development of a different type of RNN that was better suited to our application. This new "MRNN" architecture uses multiplicative connections to allow the current input character to determine the hidden-to-hidden weight matrix. We trained MRNNs on over a hundred megabytes of text for several days, using 8 Graphics Processing Units in parallel, to perform significantly better than one of the best word-agnostic character-level language models: the sequence memoizer (Wood et al., 2009; Gasthaus et al., 2010), which is a hierarchical nonparametric Bayesian method.

While our method performs at the state of the art for pure character-level models, its compression performance falls short of the best models which have explicit knowledge of words, the most powerful of these being PAQ8hp12 (Mahoney, 2005). PAQ is a mixture model of a large number of well-chosen context models whose mixing proportions are computed by a neural network. Its weights are a function of the current context and its predictions are further combined with a neural-network like model. Unlike standard compression techniques, some of PAQ's context models not only consider contiguous contexts but also contexts with "gaps", allowing it to capture some types of longer range structures cheaply. More significantly, PAQ is not word-agnostic, because it uses a combination of character-level and word-level models. PAQ also preprocesses the data with a dictionary of common English words which we disabled, because it gave PAQ an unfair advantage over models that do not use such task-specific

(and indeed, English-specific) explicit prior knowledge. The numerous mixture components of PAQ were chosen because they improved performance on a development set, so in this respect PAQ is similar in model complexity to the winning entry of the Netflix prize (Bell et al., 2007).

Finally, language models can be used to "generate" language, and to our surprise, the text generated by the MRNNs we trained exhibited a significant amount of interesting and high-level linguistic structure, featuring a large vocabulary, a considerable amount of grammatical structure, and a wide variety of highly plausible proper names that were not in the training set. Mastering the vocabulary of English did not seem to be a problem for the MRNN: it generated very few uncapitalized non-words, and those that it did generate were often very plausible, like "homosomalist" or "un-ameliary". Of particular interest was the fact that the MRNN learned to balance parentheses and quotes over long distances (e.g., 30 characters). A character-level N-gram language model could only do this by modeling 31-grams, and neither Memoizer nor PAQ are representationally capable of balancing parentheses because of their need for exact context matches. In contrast, the MRNN's nonlinear dynamics enables it to extract higher level "knowledge" from the text, and there are no obvious limits to its representational power because of the ability of its hidden states to perform general computation.

5.2 The Multiplicative RNN

Having applied a modestly-sized standard RNN architecture to the character-level language modeling problem (where the target output at each time step is defined as the the input character at the next time-step), we found the performance somewhat unsatisfactory, and that while increasing the dimensionality of the hidden state did help, the per-parameter gain in test performance was not sufficient to allow the method to be both practical and competitive with state-of-the-art approaches. We address this problem by proposing a new RNN architecture called the Multiplicative RNN (MRNN) which we will argue is better suited to the language modeling task.

5.2.1 The Tensor RNN

The dynamics of the RNN's hidden states depend on the hidden-to-hidden matrix and on the inputs. In a standard RNN (as defined in sec. 2.5) the current input v_t is first transformed via the visible-to-hidden weight matrix W_{hv} and then contributes additively to the input for the current hidden state. A more powerful way for the current input character to affect the hidden state dynamics would be to determine the entire hidden-to-hidden matrix (which defines the non-linear dynamics) in addition to providing an additive bias.

One motivation for this approach came from viewing an RNN as a model of an unbounded tree in which each node is a hidden state vector and each edge is labelled by a character that determines how the parent node gives rise to the child node. This view emphasizes the resemblance of an RNN to a Markov model that stores familiar strings of characters in a tree, and it also makes it clear that the RNN tree is potentially much more powerful than the Markov model because the distributed representation of a node allows different nodes to share knowledge. For example, the character string "ing" is quite probable after "fix" and also quite probable after "break". If the hidden state vectors that represent the two histories "fix" and "break" share a common representation of the fact that this could be the stem of a verb, then this common representation can be acted upon by the character "i" to produce a hidden state that predicts an "n". For this to be a good prediction we require the *conjunction* of the verb-stem representation in the previous hidden state and the character "i". One or other of these alone does not provide half as much evidence for predicting an "n": it is their conjunction that is important. This strongly suggests that we need a multiplicative interaction. To achieve this goal we modify the RNN so

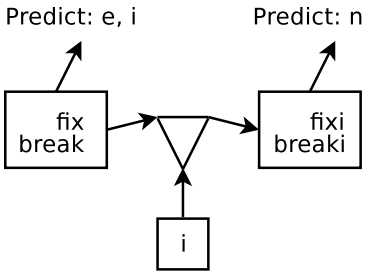


Figure 5.1: An illustration of the significance of the multiplicative connections (the product is depicted by a triangle). The presence of the multiplicative connections enables the RNN to be sensitive to conjunctions of context and character, allowing different contexts to respond in a qualitatively different manner to the same input character. Even though standard RNNs can also be sensitive to the conjunction of the hidden state and the input, it is more natural with multiplicative connections.

that its hidden-to-hidden weight matrix is a (learned) function of the current input v_t:

$$h_t = \tanh\left(W_{hv}v_t + W_{hh}^{(v_t)}h_{t-1} + b_h\right) \tag{5.1}$$

$$o_t = W_{oh}h_t + b_o \tag{5.2}$$

These are identical to the equations in sec. 2.5, except that W_{hh} is replaced with $W_{hh}^{(v_t)}$, so each character to specifies a different hidden-to-hidden weight matrix.

It is natural to define $W_{hh}^{(v_t)}$ using a tensor. If we store M matrices, $W_{hh}^{(1)}, \ldots, W_{hh}^{(M)}$, where M is the number of dimensions of v_t, we could define $W_{hh}^{(v_t)}$ by the equation

$$W_{hh}^{(v_t)} = \sum_{m=1}^{M} v_t^{(m)} W_{hh}^{(m)} \tag{5.3}$$

where $v_t^{(m)}$ is the m-th coordinate of v_t. When the input v_t is a 1-of-M encoding of a character, it is easily seen that every character has an associated weight matrix and $W_{hh}^{(v_t)}$ is the matrix assigned to the character represented by v_t. [1]

5.2.2 The Multiplicative RNN

The above scheme, while appealing, has a major drawback: fully general 3-way tensors are not practical because of their size. In particular, if we want to use RNNs with a large number of hidden units (say, 1000) and if the dimensionality of v_t is even moderately large, then the storage required for the tensor $W_{hh}^{(v_t)}$ becomes prohibitive.

It turns out we can remedy the above problem by factoring the tensor $W_{hh}^{(v)}$ (Taylor and Hinton, 2009). This is done by introducing the three matrices W_{fv}, W_{hf}, and W_{fh}, and reparameterizing the matrix $W_{hh}^{(v_t)}$ by the equation

$$W_{hh}^{(v_t)} \equiv W_{hf} \cdot \mathrm{diag}(W_{fv}v_t) \cdot W_{fh} \tag{5.4}$$

[1] The above model, applied to discrete inputs represented with their 1-of-M encodings, is the nonlinear version of the Observable Operator Model (Jaeger, 2000) whose linear nature makes it closely related to an HMM in terms of expressive power.

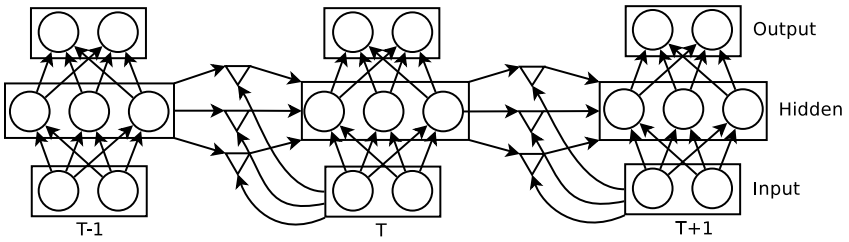


Figure 5.2: The Multiplicative Recurrent Neural Network "gates" the recurrent weight matrix with the input symbol. Each triangle symbol represents a factor that applies a learned linear filter at each of its two inputs. The product of the outputs of these two linear filters is then sent, via weighted connections, to all the units connected to the output of the triangle. Consequently every input vector can synthesize its own hidden-to-hidden weight matrix by determining the gains on all of the factors, each of which represents a rank one hidden-to-hidden weight matrix defined by the outer-product of its incoming and outgoing weight-vectors to the hidden units. The synthesized weight matrices share "structure" because they are all formed by blending the same set of rank one matrices. In contrast, an unconstrained tensor model provides each input with a completely separate weight matrix.

where $\mathrm{diag}(v)$ is a diagonal matrix whose diagonal is given by v. If the dimensionality of the vector $W_{fv}v_t$, denoted by F, is sufficiently large, then the factorization is as expressive as the original tensor. Smaller values of F require fewer parameters while hopefully retaining a significant fraction of the tensor's expressive power.

The Multiplicative RNN (MRNN) is the result of factorizing the Tensor RNN by expanding eq. 5.4 within eq. 5.1. The MRNN computes the hidden state sequence $(h_1, \ldots, h_T)$, an additional "factor state sequence" $(f_1, \ldots, f_T)$, and the output sequence $(o_1, \ldots, o_T)$ by iterating the equations

$$f_t = \mathrm{diag}(W_{fv}v_t) \cdot W_{fh}h_{t-1} \tag{5.5}$$

$$h_t = \tanh(W_{hf}f_t + W_{hv}v_t) \tag{5.6}$$

$$o_t = W_{oh}h_t + b_o \tag{5.7}$$

which implement the neural network in fig. 5.2. The tensor factorization of eq. 5.4 has the interpretation of an additional layer of multiplicative units between each pair of consecutive layers (i.e., the triangles in fig. 5.2), so the MRNN actually has two steps of nonlinear processing in its hidden states for every input timestep. Each of the multiplicative units outputs the value f_t of eq. 5.5 which is the product of the outputs of the two linear filters connecting the multiplicative unit to the previous hidden states and to the inputs.

We experimentally verified the advantage of the MRNN over the RNN when the two have the same number of parameters. We trained an RNN with 500 hidden units and an MRNN with 350 hidden units and 350 factors (so the RNN has slightly more parameters) on the "machine learning" dataset (dataset 3 in the experimental section). After extensive training, the MRNN achieved 1.56 bits per character and the RNN achieved 1.65 bits per character on the test set.

5.3 The Objective Function

The goal of character-level language modeling is to predict the next character in a sequence. More formally, given a training sequence $(v_1, \ldots, v_T)$, the RNN uses the sequence of its output vectors $(o_1, \ldots, o_T)$ to obtain a sequence of predictive distributions $P(v_{t+1}|v_{\leq t}) = \mathrm{softmax}(o_t)$, where the

softmax distribution is defined by $P(\text{softmax}(o_t) = j) = \exp(o_t^{(j)})/\sum_k \exp(o_t^{(k)})$. The language modeling objective is to maximize the total log probability of the training sequence $\sum_{t=0}^{T-1} \log P(v_{t+1}|v_{\leq t})$, which implies that the RNN learns a probability distribution over sequences. Even though the hidden units are deterministic, we can sample from an MRNN stochastically because the states of its output units define the conditional distribution $P(v_{t+1}|v_{\leq t})$. We can sample from this conditional distribution to get the next character in a generated string and provide it as the next input to the RNN. This means that the RNN is a directed non-Markov model and, in this respect, it resembles the sequence memoizer (Wood et al., 2009). This point is also discussed in sec. 2.5.2.

5.4 Experiments

The goal of our experiments is to demonstrate that the MRNN, when trained by HF, learns high-quality language models. We demonstrate this by comparing the MRNN to the sequence memoizer and to PAQ on three real-world language datasets. After splitting each dataset into a training and test set, we trained a large MRNN, a sequence memoizer[2], and PAQ, and report the bits per character (bpc) each model achieves on the test set.

Owing to its nonparametric nature and the nature of the data-structures it uses, the sequence memoizer is very memory intensive, so it can only be applied to training datasets of roughly 130MB or less on a machine with 32GB of RAM. In contrast, the MRNN can be applied to datasets of unlimited size although it typically requires considerably more total FLOPS to achieve good performance (but, unlike the memoizer, it is easily parallelized). However, to make the experimental comparison fair, we train the MRNN, the memoizer, and PAQ on datasets of the same size.

5.4.1 Datasets

We now describe the datasets. Each dataset is a long string of characters from an 86-character alphabet of about 100MB that includes digits and punctuation, together with a special symbol which indicates that the character in the original text was not one of the other 85 characters in our alphabet. The last 10 million characters of each dataset are used as a test set.

1. The first dataset is a sequence of characters from the English Wikipedia. We removed the XML and the Wikipedia markup to clean the dataset. We randomly permuted its articles before partitioning it into a train and a test set.

2. The second dataset is a collection of articles from the New York Times (Sandhaus, 2008).

3. The third dataset is a corpus of machine learning papers. To construct this dataset we downloaded every NIPS and JMLR paper, and converted them to plain text using the pdftotext utility. We then translated a large number of special characters to their ascii equivalents (including non-ascii punctuation, greek letters, and the "fi" and "if" symbols), and removed most of the unstructured text by using only sentences consisting of at least 70% alphanumeric characters. Finally, we randomly permuted the papers.

The first two corpora are subsets of larger corpora (over 1GB large).

5.4.2 Training details

To compute the exact gradient of the log probability of the training set (eq. 5.3), the MRNN needs to process the entire training set sequentially and store the hidden state sequence in order to apply

[2]Which has no hyper-parameters and strictly speaking is not 'trained' but rather conditioned on the training set.

DATA SET	MEMOIZER	PAQ	MRNN	MRNN (FULL SET)
WIKI	1.66	1.51	1.60 (1.53)	1.55 (1.54)
NYT	1.49	1.38	1.48 (1.44)	1.47 (1.46)
ML	1.33	1.22	1.31 (1.27)	

Table 5.1: This table shows the test bits per character for each experiment, with the training bits in brackets (where available). The MRNN achieves lower bits per character than the sequence memoizer but higher than PAQ on each of the three datasets. The MRNN (full set) column refers to MRNNs trained on the larger (1GB) training corpora (except for the ML dataset which is not a subset of a larger corpus). Note, also, that the improvement resulting from larger dataset is modest, implying that the an MRNN with 1500 units and factors is fairly well-trained with 100MB of text.

backpropagation-through-time. This is infeasible due to the size of the training set but it is also unnecessary: training the MRNN on many shorter sequences is just as effective, provided they are several hundred characters or more long. If the sequences are too short, we fail to utilize the ability of the HF optimizer to capture long-term dependencies spanning hundreds of timesteps.

An advantage of using a large number of relatively short sequences over using a single long sequence is that the former is much easier to parallelize. This is essential, since our preliminary experiments suggested that HF applied to MRNNs works best when the gradient is computed using millions of characters and the curvature-matrix vector products are computed using hundreds of thousands of characters. Using a highly parallel system (consisting of 8 high-end GPUs with 4GB of RAM each), we computed the gradient on 160·300=48000 sequences of length 250, of which 8·300=2400 sequences were used to compute the curvature-matrix vector products that are needed for the HF optimizer (Martens and Sutskever, 2011) (so each GPU processes 300 sequences at a time).

The first few characters of any sequence are much harder to predict because they do not have a sufficiently large context, so it is not beneficial to have the MRNN spend neural resources predicting these characters. We take this effect into account by having the MRNN predict only the last 200 timesteps of the 250-long training sequences, thus providing every prediction with at least 50 characters of context.

The Hessian-Free optimizer (Martens, 2010) and its RNN-specialized variant (Martens and Sutskever, 2011) have a small number of meta-parameters that must be specified. We imposed a maximum of 150 steps to be used by each run of CG within HF, set the structural damping coefficient μ to 0.1, and initialized λ to 10 (see Chapter 4 for a description of these meta-parameters). Our HF implementation uses a different subset of the training data at every iteration, so at a coarse temporal scale it is essentially online. In this setup, training lasted roughly 5 days for each dataset.

We found that a total of 120·150 weight updates was sufficient to adequately train an MRNN. More specifically, we used 120 steps of HF, with each of these steps using a maximum of 150 conjugate gradient iterations to approach the minimum of the quadratic Gauss-Newton-based approximation to the objective function, which remains fixed during the conjugate gradient iterations. The small number of weight updates, each requiring a massive amount of computation, makes the HF optimizer much easier to parallelize than stochastic gradient descent.

In all our experiments we use MRNNs with 1500 hidden units and 1500 factors, giving them 4,900,000 parameters. The MRNNs were initialized with sparse connections: each unit starts out with 15 nonzero connections to other units (Martens and Sutskever (2011) and Chap. 4). Note that if we unroll the MRNN in time (as in fig. 5.2) we obtain a neural network with 500 layers of size 1500 if we view the multiplicative units f_t as layers. This is among the deepest neural network ever trained.

5.4.3 Results

The main experimental results are shown in table 5.1. We see that the MRNN predicts the test set more accurately than the sequence memoizer but less accurately than the dictionary-free PAQ on the three datasets.

5.4.4 Debagging

It is easy to convert a sentence into a bag of words, but it is much harder to convert a bag of words into a meaningful sentence. We name the latter the debagging problem. We perform an experiment where a character-level language model evaluates every possible ordering of the words in the bag, and returns and the ordering it deems best. To make the experiment tractable, we only considered bags of 7 words, giving a search space of size 5040.

For our experiment, we used the MRNN and the memoizer[3] to debag 500 bags of randomly chosen words from "Ana Karenina". We use 11 words for each bag, where the first two and the last two words are used as context to aid debagging the middle seven words.

We say that the model correctly debags a sentence if the correct ordering is assigned the highest log probability. We found that the wikipedia-trained MRNN recovered the correct ordering 34% of the time, and the wikipedia-trained memoizer did so 27% of the time. Given that the problem is "word-level", utilizing large character contexts is essential to achieve good performance.

5.5 Qualitative experiments

In this section we qualitatively investigate the nature of the models learned by the MRNN.

5.5.1 Samples from the models

The simplest qualitative experiment is to inspect the samples generated by the three MRNNs. The most salient characteristic of the samples is the richness of their vocabularies. Further inspection reveals that the text is mostly grammatical, and that parentheses are usually balanced over many characters. The unusual artifacts in the generated text, such as consecutive commas or quotes, are the result of the data preprocessing and are frequently found in the training set.

Samples from the Wikipedia model

We now present a sample from the Wikipedia model. We use ? to indicate the "unknown" character. The sample below was obtained by running the MRNN less than 10 times and selecting the most intriguing sample. The MRNN was initialized with the phrase "The meaning of life is":

The meaning of life is the tradition of the ancient human reproduction: it is less favorable to the good boy for when to remove her bigger. In the show's agreement unanimously resurfaced. The wild pasteured with consistent street forests were incorporated by the 15th century BE. In 1996 the primary rapford undergoes an effort that the reserve conditioning, written into Jewish cities, sleepers to incorporate the .St Eurasia that activates the population. Mar??a Nationale, Kelli, Zedlat-Dukastoe, Florendon, Ptu's thought is. To adapt in most parts of North America, the dynamic fairy Dan please believes, the free speech are much related to the serpents envisioned as they are considered the duo versus

[3] We were unable to modify the implementation of PAQ to make debagging feasible.

Samples from the NYT model

Below is a sample from the model trained on the full NYT dataset, where the MRNN was initialized with a single space. The spaces surrounding the punctuation are an artifact of the preprocessing.

while he was giving attention to the second advantage of school building a 2-for-2 stool killed by the Cultures saddled with a half-suit defending the Bharatiya Fernall 's office . Ms . Claire Parters will also have a history temple for him to raise jobs until naked Prodiena to paint baseball partners , provided people to ride both of Manhattan in 1978 , but what was largely directed to China in 1946 , focusing on the trademark period is the sailboat yesterday and comments on whom they obtain overheard within the 120th anniversary , where many civil rights defined , officials said early that forms , " said Bernard J. Marco Jr. of Pennsylvania , was monitoring New York City 's most armed forces accused of remaining law and order to the Japanese history .

Samples from the ML model

Finally, we generate text from an MRNN trained on the ML corpus conditioned on the string "Recurrent". This MRNN is also able to balance parentheses (e.g., the second line of the sample):

Recurrent network with the Stiefel information for logistic regression methods Along with either of the algorithms previously (two or more skewprecision) is more similar to the model with the same average mismatched graph. Though this task is to be studied under the reward transform, such as (c) and (C) from the training set, based on target activities for articles a ? 2(6) and (4.3). The PHDPic (PDB) matrix of cav'va using the three relevant information contains for tieming measurements. Moreover, because of the therap tor, the aim is to improve the score to the best patch randomly, but for each initially four data sets. As shown in Figure 11, it is more than 100 steps, we used ?? \to \infty with 1000 samples) of problem varying \gamma.

5.5.2 Structured sentence completion

In this section, we investigate the MRNN's response in various situations by sampling from the MRNN's distribution conditioned on a prefix. The goal is to see whether the MRNN is able to generate "plausible" continuations to the initial strings. In our first experiment, we use the Wikipedia MRNN to complete the string "England, Spain, France, Germany,":

England, Spain, France, Germany, and Massachusetts.
England, Spain, France, Germany, cars, and direct schools
England, Spain, France, Germany, , or New Orleans and Uganda.
England, Spain, France, Germany, , Westchester,
Jet State, Springfield, Athleaves and Sorvinhee

In the above completions, the MRNN correctly interpreted the string to be a list of locations, so the generated text was also a part of a list.

Next, we performed a similar experiment using the ML model and the pair of strings "(ABC et al" and "ABC et al". The system has never seen the string "(ABC et al" in its training set (simply because there is no machine learning author named ABC, and its capitalization is particularly uncommon for a citation), so the MRNN needed to generalize over an entirely new author name:

(ABC et al., 2003), ?13?, and for a supervised Mann-¡Whitnaguing
(ABC et al., 2002), based on Lebanon and Haussler, 1995b)
ABC et al. (2003b), or Penalization of Information
ABC et al. (2008) can be evaluated and motivated by
providing optimal estimate

This example shows that the MRNN is sensitive to the initial bracket before "ABC", illustrating its representational power. The above effect is extremely robust. In contrast, both N-gram models and the sequence memoizer cannot make such predictions unless these exact strings (e.g., "(ABC et al., 2003)") occur in the training set, which cannot be counted on. In fact, *any* method which is based on precise context matches is *fundamentally incapable* of utilizing long contexts, because the probability that a

long context occurs more than once is vanishingly small. We experimentally verified that neither the sequence memoizer nor PAQ are sensitive to the initial bracket.

5.6 Discussion

Modeling language at the character level seems unnecessarily difficult because we already know that morphemes are the appropriate units for making semantic and syntactic predictions. Converting large databases into sequences of morphemes, however, is non-trivial compared with treating them as character strings. Also, learning which character strings make words is a relatively easy task compared with discovering the subtleties of semantic and syntactic structure.

All our experiments show that an MRNN finds it easy to learn words. With the exception of proper names, the generated text contains very few non-words. At the same time, the MRNN also assigns probability to (and occasionally generates) plausible words that do not appear in the training set (e.g., "cryptoliation", "homosomalist", or "un-ameliary"). This is a desirable property, which enabled the MRNN to gracefully deal with real words that it nonetheless did not see in the training set. Predicting the next word by making a sequence of character predictions avoids having to use a huge softmax over all known words and this is so advantageous that some word-level language models actually make up binary "spellings" of words so that they can predict them one bit at a time (Mnih and Hinton, 2009).

MRNNs already learn surprisingly good language models using only 1500 hidden units, and unlike other approaches such as the sequence memoizer and PAQ, they are easy to extend along various dimensions. If we could train much bigger MRNNs with millions of units and billions of connections, it is possible that brute force alone would be sufficient to achieve an even higher standard of performance. But this will of course require considerably more computational power.

Chapter 6

Learning Control Laws with Recurrent Neural Networks

6.1 Introduction

Control is the problem of computing motor commands that produce a desired motion. A controller that can rapidly generate motion is useful for building robots, so designing such controllers is an important problem. In this chapter we show how to use Hessian-Free optimization (Martens, 2010) to learn Recurrent Neural Networks that can control simulated arms in conditions of delayed feedback, disturbances, and actuator noise. To successfully reach a target under these conditions, the RNN controller needs to estimate the ongoing disturbance and look into the future to account for the delayed feedback. Learning RNN controllers is also interesting from a neuroscience perspective (Todorov, 2004; Scott, 2004) — the way in which neural circuits compute optimal sensorimotor control is not well-understood, and it is not clear how other promising engineering approaches to control (such as online trajectory optimization) might be implemented neurally.

A simple way to control noiseless plants[1] that are easy to simulate and whose derivatives are readily available is to directly optimize the command sequence, which is stored in a table, to produce a satisfactory motion (also known as "trajectory optimization"). Such methods, for example the iterative Linear Quadratic Regulator (iLQR; Li and Todorov, 2004), are effective when applicable, but are generally difficult to apply for two reasons. First, plants often have contact forces and that makes them non-differentiable. Second, realistic plants cannot be optimally controlled with a precomputed sequence of commands due to the unpredictable events that occur during the motion. For example, a plant may have actuator and/or sensor noise, unpredictable disturbances (e.g., one of the arm's joints develops friction due to an unexpected temperature change; the arm picks up an object, thus altering the movement dynamics; or an opponent pushes on the arm), and delayed feedback (i.e., if the system is to operate very quickly, it may observe the plant's state with some delay, which is especially relevant to neural systems). In these situations, the controller must use feedback, memory, and foresight to identify and then counteract the unpredictable disturbance. The delayed feedback in particular complicates planning because a controller cannot rely on the current observation alone when choosing an action: such a controller would realize that it missed the target only after the fact. To be accurate the controller needs to infer that the target has been reached before it has observed that it is so.

The above issues can be partially addressed by online trajectory optimization (OTO), which uses an iLQR-like method to recompute the sequence of commands after every timestep of the simulation,

[1] A *plant* is a general term denoting any system that can be influenced via control commands.

allowing OTO to react to a disturbance by replanning at each timestep. It is a highly effective method that operates well in all regions of the state space for which a detailed model of the plant exists. However, the method has several weaknesses. For example, it ignores disturbances when it recomputes trajectories, which causes it to undercompensate. If an OTO method needs to use a robotic arm to lift a heavy object, it will use too little force unless it explicitly models the forces created by the object.

In this chapter we describe a method that, for a given known differentiable plant, produces a Recurrent Neural Network controller that uses feedback to bring the plant to any state in the presence of the obstacles mentioned above. Our method derives its strength from the RNN's ability to represent a controller that estimates and counteracts a disturbance on the fly, and takes the delayed feedback into account. Like iLQR, we rely on second order nonlinear optimization, but we aim to learn a globally applicable policy. Although optimizing global policies is considerably more challenging than optimizing individual trajectories, RNNs have several advantages over trajectory-optimization methods. Key advantages are flexibility and the ability to specify more general notions of optimality. For example, it is trivial to formulate an RNN control objective that includes delayed feedback and disturbances spanning long periods of time, allowing the RNN (at least in principle) to compute optimal trajectories in these situations. In contrast, there is no obvious way to use such objectives in a trajectory optimization setting. Another aspect of the RNN's flexibility is the ease of incorporating sensory information. For example, it is straightforward to connect an RNN controller to the output of a neural network that performs visual perception and to adapt the control with visual cues.

RNN controllers are extremely difficult to optimize because at their heart lies a difficult RNN optimization problem (Bengio et al., 1994), which is reflected by the absence of successful RNN controllers in the literature. We have chosen to engage this difficult problem because of the recent advance in RNN training methods that was described in Chapter 4. So while previous work on RNN controllers was not particularly successful, we suspected that HF would be powerful enough to optimize RNN controllers. And indeed, we found the HF optimizer of Martens and Sutskever (2011) to be capable of learning controllers that work in the presence of unknown disturbances and delays. We relied on a new variant of HF called Augmented-HF (or AHF) which was found to be more robust and had a significantly lower rate of optimization failure.

As far as we are aware, our work is the first that successfully incorporates disturbances and delayed feedback into the control objective. Indeed, while approaches like online trajectory optimization can deal with disturbances, their trajectories will likely not be optimal, because their objective does not take the disturbances into consideration, and are fundamentally incapable of correctly dealing with delayed feedback. As a result, they may use excessive force when recovering from a disturbance, and oscillate around the target because of the delayed feedback. In contrast, by incorporating the delayed feedback and the disturbances into the control objective, our approach implicitly approximates the best possible trajectories subject to delayed feedback and disturbances.

6.2 Augmented Hessian-Free Optimization

This work applies Hessian-Free optimization to the problem of learning recurrent neural networks that are robust controllers, so we briefly review Hessian-Free optimization (Martens, 2010). HF is fundamentally based on the repeated optimization of rich local quadratic approximations

$$q(x) = \frac{x^\top A x}{2} - b^\top x \tag{6.1}$$

to the nonlinear objective f. It is distinct from other second order methods, such as L-BFGS, because it uses a very high-rank curvature matrix which is accessible only via a function that multiplies it by

vectors. HF then optimizes this quadratic using linear conjugate gradient (CG), which is a powerful method for minimizing quadratics that exploits directions of low curvature. CG is powerful because it maintains the global minimizer of q over the i-th Krylov subspace

$$K_i(A, b) \equiv \text{span}\{b, Ab, A^2b, \ldots, A^{i-1}b\} \tag{6.2}$$

at the i-th iteration, which immediately implies that CG converges in at most N iterations (where N is the number of problem dimensions). CG's agressive pursuit of low-curvature directions is beneficial, because it contributes to the global optimization of our problems and causes the solutions to be of higher quality. Overall, the effectiveness of CG and a rich curvature make HF an extremely powerful optimization technique.

Augmented-HF (or AHF), like HF, is a particular flavor of truncated Newton optimization algorithm that uses sophisticated damping strategies in order to achieve practical performance for the large and the most challenging optimizations that appear in machine learning. Here, "damping" refers to various modifications of the quadratic approximation q, or the optimization of it, with the intent to produce smaller and less aggressive update proposals that are more likely to lie in regions of the parameter space where q remains a reasonable approximation to f. Martens (2010) used the well-known approach of adding λI to the curvature matrix (sometimes referred to as Tikhonov regularization) which encourages updates to have a small magnitude (measure in the default parameter space), with λ being dynamically adapted according to standard heuristics, along early truncations of the CG optimization. Later, Martens and Sutskever (2011) and chap. 4 introduced an additional "structural damping" technique which added terms to the curvature matrix, which encouraged updates to have a low magnitude of change in the hidden unit activations by penalizing a quadratic approximation of this quantity. With AHF, instead of relying on explicit modifications of the quadratic, a similar approach to Krylov Subspace Descent (KSD; Vinyals and Povey, 2011) it taken, and direct optimization of the objective function is performed over the Krylov subspace which contains all of the solutions that would be found by CG for any choice of λ and any level of CG truncation. But while Krylov Subspace Descent optimizes the coefficient over an orthonormal basis of a fixed-dimension subspace, and optimizes this with a standard non-linear optimizer like BFGS, AHF performs greedy sequential optimization of the coefficients α_i over an ever expanding conjugate basis $\{c_1, c_2, ...\}$ of CG's Krylov subspace. To justify this, we note that in terms of the optimization of the quadratic approximation q, the optimal choices for the coefficients of the conjugate vectors do not depend on the values of the coefficients for the other vectors, and so since q locally approximates the objective, greedy independent optimization of these coefficients ought to be reasonably effective for the true objective, and we have found this to be true in practice. To see that the coefficients can be independently optimized in the quadratic setting, note that

$$\begin{aligned} q\left(\sum_i \alpha_i c_i\right) &= \frac{1}{2}\sum_i \alpha c_i^\top A \sum_i \alpha c_i + b^\top \sum_i \alpha c_i \\ &= \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j c_i^\top A c_j + \sum_i \alpha b^\top c_i \\ &= \frac{1}{2}\sum_i \alpha_i^2 c_i^\top A c_i + \sum_i \alpha b^\top c_i = \sum_i q(\alpha_i c_i) \end{aligned}$$

where we have used the defining property of a conjugate basis, that $c_i^\top A c_j = 0$ for $i \neq j$.

The type of damping achieved by AHF via directly searching over Krylov subspaces generalizes and subsumes both Tikhonov regularization and CG truncation (although not structural damping) used in the original HF formulation, and we have found it to be more powerful than these approaches in general. Finally, in order to allow for the "hot-starts" of CG used in the original HF formulation, a

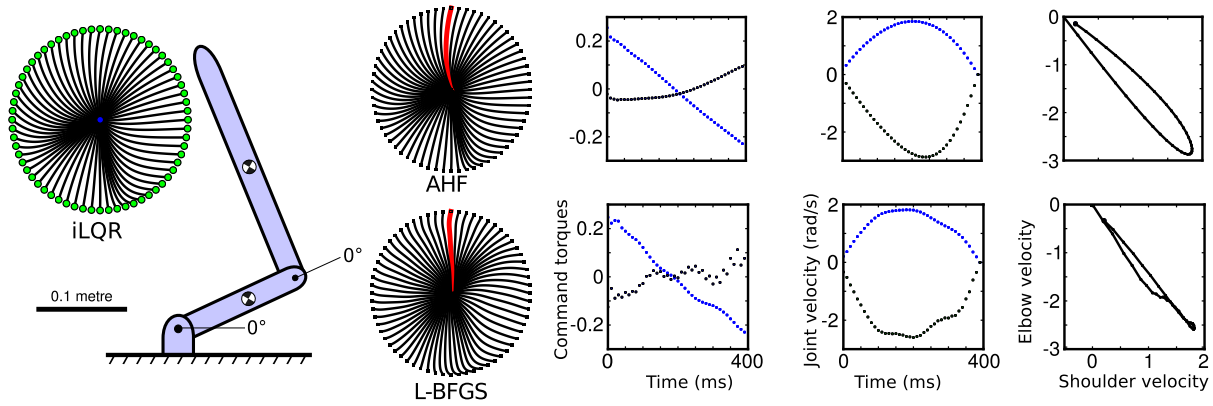


Figure 6.1: The figure displays the artificial limb with and its joint angles, and the center-out task with the iLQR trajectories in the correct scale relative to the limb. The figure also presents the trajectories of two RNNs trained on the center-out task. The four top panels show the trajectories and the command torques of RNNs trained by AHF and the bottom by L-BFGS. Although AHF's energy expenditure is only slightly lower than L-BFGS's, the trajectories and the command torques (that correspond to the trajectory colored red) found by AHF are much nicer and closer to iLQR's. Where applicable, blue traces denote the shoulder joint torque/velocity, and black denotes elbow torque/velocity.

conjugate basis for "augmented" Krylov subspace optimized over using an augmented CG/Lanczos algorithm of our own design. The name of the approach, as well as most of its complexity and per-iteration performance advantages over Krylov Subspace Descent derive mostly from this last element. AHF has several other important practical advantages over Krylov Subspace Descent. Most important among these is that, unlike Krylov Subspace Descent, it does not need to store an entire basis for the Krylov subspace in memory at any one time, and the subspace dimension need not be fixed ahead of time.

6.3 Experiments: Tasks

In this chapter we have examined three tasks requiring control of a two-joint limb in 2D-space. The first task (figure 6.1) is a straightforward reaching task where the controller is required to move the limb from an initial configuration x to some desired configuration x^* at some final time T while minimizing energy expenditure. Here we examined the solution to a series of reaches arranged in a circle around a central starting point; variants of this task are well known in the neuroscience motor control literature (e.g., Scott et al., 2001), and we will call it the *center-out* task.

For the second task (inspired by a similar task used in monkey neurophysiology; Sergio and Kalaska, 2003), we looked at a postural task where the controller is perturbed away from one of 9 desired postures, x^*_{1-9}, by one of 8 constant forces applied to the limb. The controller is required to resist the ongoing perturbation and remain close to the original starting position. Importantly, the perturbation on any given trial is unknown to the controller and must be inferred from the observations via feedback. The feedback is delayed – in this case the information is received 100ms (10 time steps in our simulations) after it had occurred. This renders the task substantially more difficult because the network is required to "plan" for the future: i.e., even if the network manages to infer the force which is being applied to the hand in a given trial, it must then resist and push back just enough to bring the hand back to the target without overshooting. Controllers which are less "intelligent" will tend to oscillate (or "ring") around the desired position since they only become aware that they have overshot the mark when the

delayed feedback instructs them so – with a delay – and this may happen repeatedly and often leads to instability.

For the third and most difficult task, the controller was required to move the limb from a random starting position in the human/monkey workspace to a random target in the workspace. This task was further complicated in the following ways: (a) Sensory feedback was delayed by 70ms (7 time steps), (b) Constant perturbations drawn from a Gaussian distribution were applied to the shoulder and elbow joints (c) Signal dependent Gaussian noise (i.e., noise whose magnitude grows with the magnitude of the signal as in real neurons and muscles) was applied to the torque commands issued by the network. We call it the *disturbance-delays-noise* (DDN) task. In order to successfully reach and remain at the target the network must dynamically infer the perturbation applied to the limb from the delayed and noisy sensory feedback. As well, it must integrate sensory information over time to tease apart whether the perturbations from an expected trajectory are due to an external perturbation or merely caused by motor noise.

Our plant is based on a continuous time physics model which we discretize with $\Delta t = 0.01$s steps and integrate with simple Euler integration. The center-out task is given 400ms, and the DDN task 600ms, while the postural task is given 1000ms (i.e., 40, 60, and 100 timesteps). Thus the motions were all fairly brief. The objective for the center-out task measures the the squared difference between the target and final state plus $\alpha = 10^{-2} \cdot \Delta t$ times the sum of the squares of the command torques. The objective for the DDN task is identical to the center-out's objective, except that $\alpha = 10^{-4} \cdot \Delta t$. The objective for the postural task is to minimize the squared distance between the actual and target joint angles at each timestep (as opposed to the final state), and for the second half of each trial, to zero the joint velocities as well (to quench oscillation). In this task we used an energy cost on the squared commands of $\alpha = 10^{-6} \cdot \Delta t$.

6.4 Network Details

We tried a number of configurations of network structure and nonlinearities and found a single network type which we found had enough capacity to perform reasonably well for all the tasks described. We used a 2-hidden layer network of fully connected $\tanh(\cdot)$ units. There were 100 units in each layer and each layer was fully recurrently connected within itself, though not to the other layer. In addition to the usual connections between the network inputs and the first layer, there were "skip" connections from the inputs to the second layer. The second layer of the network was fully connected to two linear output units which were interpreted as the commanded joint torques at the shoulder and elbow joints, i.e., $u_t = [\tau_\theta, \tau_\phi]$. The network was chosen to have this size and architecture in order to enable it to represent solutions with low energy expenditure, as we found that smaller and simpler nets had little trouble reaching the targets, but their energy expenditure was noticeably greater.

At each timestep t the network received a (possibly delayed, depending on the task) copy of the state of the arm coded as the shoulder (θ) and elbow (ϕ) joint angles and velocities, i.e., $x_{t-\tau} = \left[\theta, \phi, \dot{\theta}, \dot{\phi}\right]$, where $\tau \geq 0$ is the delay in the feedback. As well, the network received a goal or desired final state, x^*, also coded in shoulder and elbow joint angles/velocities. Thus, for example, in the reaching experiments presented here the final desired velocity was 0. This is not necessary, and we could in principle examine problems where the desired velocity at some time point is specifically non-zero (as in the problem of swinging a bat to hit a baseball). While we have not examined tasks such as this, in principle our approach should have no trouble dealing with this extension.

6.5 Formal Problem Statement

We now formally define the optimal control problems and the recurrent neural networks which are the focus of this work, specifically for the center-out and the DDN tasks. The general form of the control problems studied here is given by the equation

$$x_{t+1} = f(x_t, u_t \odot (1 + \xi_t^u) + d_t, t) + \xi_t^f$$

where f is the deterministic differentiable function that describes the discrete time-evolution of the plant, x_t is the the state of the plant, u_t is a control command, d_t is a disturbance, ξ_t^u is the command noise, and ξ_t^f is plant noise. Also, $\odot$ denotes coordinate-wise multiplication to indicate that the command noise is multiplicative.

Next we describe a one-layer RNN controller, although we use a two-layered controller in our experiments. In the equations below, c_t is a context vector representing the RNN's target:

$$\begin{aligned} v_t &= [x_t; c_t] \\ h_t &= \tanh(W_{hv} v_t + W_{hh} h_{t-1} + b_z) \\ u_t &= \tanh(W_{hu} h_t + b_u) \end{aligned}$$

The controller parameters is the concatenation of the RNN's parameters, $\theta = [W_{hv}, W_{hh}, W_{hu}, b_h, b_u]$, and the loss for a single trajectory initialized at x_0 is given by

$$L_{x^*,x_0}(\theta) = \sum_{t=0}^{T} \ell_{x^*}(x_t, u_t, t)$$

In our setting, the loss function $\ell_{x^*}(x_t, u_t, t)$ measures the distance to the target x^* and the energy expenditure:

$$\ell_{x^*}(x_t, u_t, t) = \|x_t - x^*\|^2/2 \cdot \delta_{t,T} + \alpha \|u_t\|^2/2$$

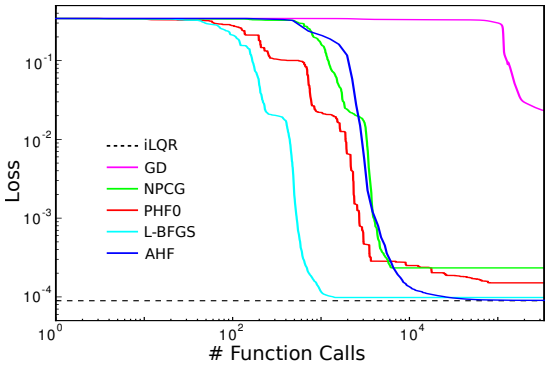
The objective is an average of losses with a range of starting positions, disturbances, and targets:

$$L(\theta) = E_{x^*,x_0}\left[L_{x^*,x_0}(\theta)\right]$$

6.6 Details of the Plant

Throughout this work we have employed a 2-joint revolute arm for the plant model which is intended to resemble the human or monkey upper limb constrained to move in the horizontal plane (Figure 6.1). We used length, mass and inertial parameters based on those found empirically for the monkey upper limb (Cheng and Scott, 2000). This plant has the advantage of being well known in the literature (Stroeve, 1998; Li and Todorov, 2004) and is a sensible plant on which we test our RNN controllers.

In modelling the limb dynamics, we initially attempted to use the widely known physics formulation given by Li and Todorov (2004). However, this implementation of the physics equations appears to be prone to a known issue (Agrawal, 1991) where the inertia matrix becomes singular in some configurations of the limb, which was especially problematic when we used monkey-like limb parameters. Thus we instead derived our own equations of motion for the limb physics.



(**a**) Performance versus time

Method	Loss
iLQR	$8.95 \cdot 10^{-5}$
AHF	$8.96 \cdot 10^{-5}$
L-BFGS	$9.29 \cdot 10^{-5}$

(**b**) Final performance table.

Figure 6.2: (**a**) The performance of several different optimization methods on the center-out task versus the number of function calls is shown. The dashed line at the bottom displays iLQR's performance which is our gold standard. The figure shows that L-BFGS reduces the objective much faster than AHF. However, it quickly reaches a point beyond which it cannot improve. AHF takes longer, but it comes considerably closer to the best possible performance, which is also reflected in both figure 6.1 and the table above. In addition, AHF is the only method that does not slow down (get a "kink") at error of 10^{-2}, indicating additional robustness. (**b**) A table with the performance of AHF, L-BFGS, and iLQR on the centerout task.

6.7 Experiments: Description of Results

6.7.1 The center-out task

Our results on the first task (Figures 6.1,6.2) demonstrate several important things. First and foremost, it shows that while the problem of finding a good setting of the parameters for the network such that it can make all of the 64 reaches around the circle is much more difficult than the problem of find purely local solutions, our approach (AHF) is able to find a solution which almost identically matches the local solutions produced by the iterative Local Quadratic Regulator (iLQR) approach. Secondly, examining the performance of a variety of optimization approaches, we found substantial variance in their ability to solve this problem. We looked at the mean loss across the 64 reaches and we treat the local solutions produced by iLQR as the gold standard (dashed black line in Figure 6.2).

We looked at the performance of standard gradient descent (GD) with a fixed learning rate tuned by hand, and three of the 2nd order methods provided by the `minFunc` package: the Nonlinear Preconditioned Conjugate Gradient (NPCG; `pcg`), the Preconditioned Hessian-Free optimizer (PHF0; `pnewton0`, and the Limited-Memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS; `lbfgs` with $L = 100$) optimizer. We also examined the performance of the `minimize.m` implementation of NPCG (results not shown as we found its performance inferior to those produced by the `minFunc` variant). Finally, we used the Hessian-Free optimizer with the structural damping as described by Martens and Sutskever (2011) and the Augmented Hessian-Free (AHF) approach which is more robust than HF. In all of the experiments presented, we used the identical network structure and network parameter initializations.

Our simulations were written using the symbolic library Theano (Bergstra et al., 2010) in Python, which made it easy to obtain the gradient and the R-operators. While not strictly necessary for solution of the problems we study, Theano is practically essential because manual computation of the derivatives for control problems are effort-intensive and error prone. In order to effectively make use of the well known `minFunc` package (Schmidt, 2009) we called our Python functions using Matlab's Mex

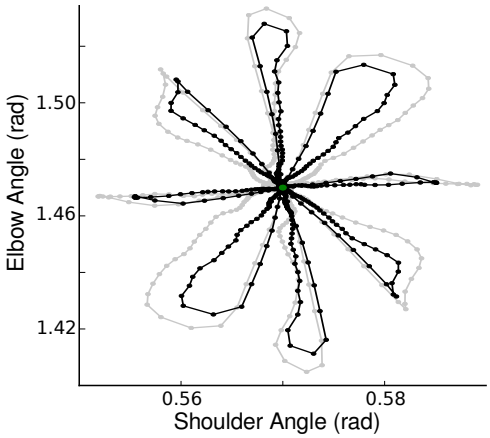


Figure 6.3: The figure shows the trajectories of an RNN trained on the postural task from a single starting position. First the limb is pushed from its starting point due to the force of the disturbance, but it quickly returns to the target and stays still, perfectly counteracting the disturbance.

functionality. As well, the results for the center-out task demonstrate that we achieved very good results with at least one of the minFunc optimizers, allowing us to be confident that our Matlab/Mex/Python interface code was working properly and that `minimize.m` and the `minFunc.m` optimizers were not disadvantaged in our setup.

Several previous works have looked at training RNNs to control simulated limbs to make reaches (Stroeve, 1998; Huh and Todorov, 2009; Shimansky et al., 2004). However, in all of the cases we are aware of, the authors used either a gradient descent variant (e.g., Stroeve, 1998) or a variant of NPCG (e.g. Huh and Todorov (2009) used `minimize.m`), and made no attempt to compare the solutions discovered to those attainable via local methods such as iLQR. Our results show that the choice of the optimization algorithm is extremely important when learning control tasks with an RNN. Only L-BFGS and AHF find solutions which are comparable to the solution discovered by iLQR, and only AHF was ever able to find a solution with near identical loss.

A robust finding was that using any of the other optimizers, i.e., GD, NPCG, PHF0, led to much worse behavioural performance even when the differences in the loss were not large. These optimizers may produce a network which will grossly move to the target, but the trajectories are obviously degraded from the optimal trajectories found by iLQR and our method. Thus, while previous research has claimed to produce good control for reaching with RNNs, these claims have simply not been evaluated with quantitative metrics as we have done here. As such, previous research has ignored the large gap between the solutions possible via GD/NPCG and those possible via local methods such as iLQR.

Why should this be the case? At first glance, the center out task appears easy, perhaps even trivial to optimize. It seems at first that all that is needed is to push on the two torque commands (for shoulder and elbow) until the arm arrives at the desired location. Perhaps, all that is required is to push harder to start and then less as one gets near to the target? For example, one could imagine pushing in a manner proportional to the distance remaining to the target. What makes the problem difficult is to arrive at the target with near-zero velocity and to do so while using as little energy as possible. In order to be successful in this regard a controller must be smart and plan for the future. In particular the problem requires that the controller begins with small velocities in the direction of the target, smoothly and slowly increase the velocity until approximately the middle of the movement (depending on the plant dynamics) and then smoothly and slowly decrease the velocity in just such a fashion that the target is arrived at the moment the velocity hits 0. Thus, the controller must decide to start slowing the motion

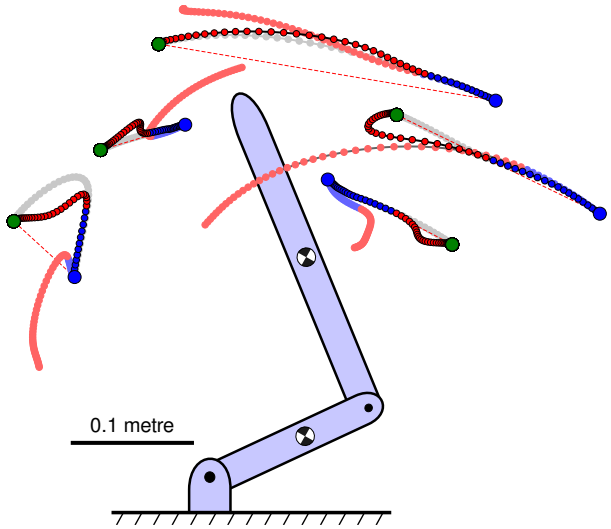


Figure 6.4: **(Best viewed in color)** The figure shows the trajectories produced by an RNN trained with delays and disturbances (the robust RNN) superimposed on top of the trajectories of an RNN trained without disturbances (but with delays). The colors of the trajectory indicate the state of the disturbance: the disturbance is inactive when the trajectory is blue, and is active when the trajectory is red. The figure also shows the trajectories of the robust RNN on the same reaching tasks with no disturbance whatsoever (the grey trajectories), allowing us to see the precise effect of the disturbance on the robust RNN's trajectory. We can see that the robust RNN takes about 7 timestep to detect and correct for the disturbance.

down in the middle of the trajectory, when the arm is still far from the target.

The results of the center-out experiment are summarized in figure 6.2.

6.7.2 The postural task

The postural task (Figure 6.3) was already described in detail in section 3. It is different from the center-out task in two ways. First, it has delayed feedback by 100ms or 10 timesteps, and second, the loss is the cumulative deviation from the target (which is equal to the starting point) over the 100 timestep of the trajectory, rather than the deviation at the last timestep. The real the objective is to identify the disturbance and to counteract it as quickly as possible, while avoiding oscillations and "ringing" around the target. On this task, both AHF (whose final loss is 2.2e-2) and L-BFGS (whose final loss is 2.3e-2) learned very high-quality solutions that quickly identify the disturbance and correct for it with no oscillations and no ringing. The postural task appears to be easier than center-out, because the RNN is given a target at every timestep, rather than at only the last one, which greatly diminishes the importance of long range temporal dependencies.

To solve the postural task we used batches of 72 sequences (which is the total number of starting point/disturbance combinations). AHF required 75,000 function evaluations, while L-BFGS needed 15,000 evaluations to solve the problem.

6.7.3 The DDN task

The Delay-Disturbance-Noise (DDN) is the hardest of our tasks. In this task, the RNN has to start at any position and reach any target within a fairly large radius (roughly the typical workspace of a human or monkey). A random disturbance is activated in the first half of the trajectory which stays unchanged

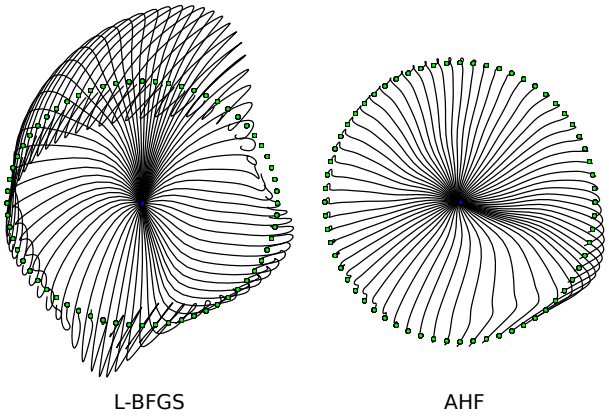


Figure 6.5: The figure shows two RNNs, trained AHF and with LBFGS on the the DDN task. Their trajectories are shown on the center-out task. These solutions exhibit a qualitative difference that goes beyond the numerical values of their objectives. The figure also shows that the value of the objective does not reveal the whole story, and that AHF consistently finds solutions that are "globally" better. We hypothesize that it is a direct consequence of its aggressive pursuit of low-curvature directions.

throughout the trajectory. The disturbance is continuous, so the RNN needs to estimate and represent the continuous disturbance in its hidden units for many timesteps. The actuator noise makes it harder for the RNN to estimate the disturbance because an apparent disturbance could be due to the noise. The task's difficulty is reflected by the relative difficulty of learning good solutions to this problem.

We ran experiments in several settings. We trained an RNN on delays alone, on delays and disturbances, and on delays, disturbances, and noise. The delay consist of 7 time steps, and the disturbances were set to Normal(0,0.2) for each joint torque. We found that longer delays and larger disturbances made the optimization problem more difficult.

Figure 6.4 shows the trajectories of an RNN trained with delays and disturbances (the robust RNN) superimposed on the trajectories of an RNN trained with delays but without disturbances (the non-robust RNN). The figure shows how disturbances make it impossible for the non-robust RNN to successfully complete the motion. It also shows how a disturbance affects the trajectory of the robust RNN for about 7 timesteps (due to the delayed feedback), after which the robust RNN identifies the disturbance and counteracts it to successfully complete the motion. A delay is essential for the disturbance to have a noticeable impact on the trajectories, because controllers with instant feedback can detect and counteract a disturbance immediately, before it made a noticeable effect on the trajectory.

We compared our solutions to L-BFGS. On the full DDN task AHF was able to get an error of $9 \cdot 10^{-3}$, while L-BFGS's error was $2 \cdot 10^{-2}$. For this task, we used batches of 400 starting points and targets for both optimizers, and refreshed L-BFGS's batch every 25 parameter updates. But just like with the center-out problem, L-BFGS's solutions are also qualitatively less good. Figure 6.5 shows the trajectories of the two RNNs on the center-out task, where the difference in the solution quality is seen clearly. To reach this performance, AHF used 100,000 function calls, while L-BFGS required 25,000 function calls, at which point it stopped making progress.

6.8 Discussion and Future Directions

Our results show that the AHF algorithm is capable of learning RNNs that can robustly control a simple plant subject to delayed feedback, disturbances, and noise. By matching iLQR's results on the center-out task, we demonstrated that the proposed method is capable of implementing near-optimal trajectories

in an RNN. As well, we expect that our method may work on more challenging plants since it is, in a sense, equivalent to iLQR: if we optimize a fixed command sequence using HF, we recover precisely the same trajectories as found with iLQR. This is because both iLQR and HF are second order methods that divide by the exact Gauss-Newton matrix. However, unlike iLQR, which is only applicable to a fixed command sequence for a single reach (it strongly relies on this fact to be able to exactly divide by the Gauss-Newton matrix), HF can be used to optimize arbitrary functions, including RNN controllers. And while optimizing RNN controllers with HF is substantially more challenging than applying iLQR, both methods are based on the same principle.

The most straightforward future direction consists of applying our technique to more complex plants. For example, we could easily apply it to 3-dimensional limbs and to arms actuated with nonlinear realistic muscles that can contract but not expand. In addition, we could train our plant to rely on a visual context rather than on a precise coordinate representation of the target.

In addition, we can also extend our method so that it works on unknown plants. We could learn a neural network approximation of the plant's forward dynamics, and use it to compute the approximate plant derivatives while using the correct forward trajectories. In principle, this technique allows us to apply our method to unknown and non-differentiable plants because of the differentiability of the approximating neural network. We have preliminary success with this approach and we hope to apply this technique to the control of a human body.

Chapter 7

Momentum Methods for Well-Initialized RNNs

In this chapter, we show that using carefully crafted (but fairly simple) random initializations, deep autoencoders and recurrent neural networks can be trained effectively using stochastic gradient descent, and that these results can be significantly improved through the use of aggressive momentum-based acceleration. For deep autoencoders we show that using any one of a variety of recently proposed random initialization schemes, deep autoencoders can be trained to a level of performance exceeding that reported by Hinton and Salakhutdinov (2006), and with the addition of Nesterov-type momentum, the results can be further improved to surpass those reported by Martens (2010). For RNNs we give a simple initialization scheme related to the one used for Echo State Networks and successfully train them on various datasets exhibiting long trem dependencies (spanning 50-200 timesteps, depending on the problem).

Our results suggest that previous attempts to train deep and recurrent neural networks from random initializations failed mostly due to poor choices of their initializations, and that the curvature issues which are present in the training objectives of deep and recurrent neural networks can be addressed through the use of aggressive momentum-based acceleration, without the need for second-order methods.

7.1 Motivation

7.1.1 Recent results for deep neural networks

Several recent results have appeared to disagree with the commonly held belief that first-order methods are incapable of learning deep models from random initializations. Glorot and Bengio (2010) and Mohamed et al. (2012) reported little difficulty training neural networks of depth 5 and 8 respectively from random initializations. Chapelle and Erhan (2011) used the random initialization of Glorot and Bengio (2010) and SGD to train the 11-layer autoencoder of Hinton and Salakhutdinov (2006). They were able to surpass the level of performance reported by Hinton and Salakhutdinov (2006) and approach the performance reported by Martens (2010) for HF.

7.1.2 Recent results for recurrent neural networks

Echo-State Networks (ESNs; Jaeger and Haas, 2004, also described in sec. 2.8.4) are standard RNNs that use an unusual learning procedure that only trains the hidden-to-output connections, and relies on a carefully crafted random initialization to create random but useful hidden dynamics which remain

fixed through training. Jaeger (2012b) successfully trained ESNs to very reliably achieve low test errors on a number of artificial datasets exhibiting pathological long-range dependencies (Hochreiter and Schmidhuber, 1997; Martens and Sutskever, 2011). It was far from obvious that ESNs could do well on these tasks, as they only learn the hidden-to-output weights, leaving the output-to-hidden and hidden-to-hidden weights fixed throughout the course of training. These results, while very impressive, are not directly comparable to those of Martens and Sutskever (2011) and Chapter 4, due to the ESNs' use of much higher dimensionality for the hidden states.

Despite the surprising power of ESNs on certain very difficult artificial problems, training the hidden dynamics still seems like a worthwhile pursuit, one that is necessary to fully exploit the theoretical representational power of RNNs. Moreover, since the number of free-parameters of an ESN grows only linearly with the hidden-state dimension, for rich and data-intensive sequence learning tasks that require millions of free parameters, an ESN-based approach would require the use of an infeasibly large hidden-state vector, although this problem could be mitigated, somewhat, by the use of a highly-optimized implementation of sparse multiplication.

Nevertheless, there is a lot to be learned from the success of ESNs, not the least of which is a good method for randomly initializing an RNN's parameters. In the ESN approach, the hidden-to-hidden weight matrices are drawn from a carefully chosen distribution. It is designed to encourage rich hidden dynamics that can convey information about the input, and various random transformations of it, across many time-steps. This suggests that ESN-based initializations may be useful for RNN training, especially for problems exhibiting long-range dependencies, where training has a tendency to get stuck in regions of the parameter space where no useful information is ever transmitted over a long range.

7.2 Momentum and Nesterov's Accelerated Gradient

A classical technique for accelerating stochastic gradient descent is the momentum method (abbrv. M; sec. 2.2), which accumulates a velocity vector in directions of persistent reduction in the objective (Plaut et al., 1986). This property allows momentum to accumulate velocity in directions of low curvature that persist across multiple iterations, leading to accelerated progress in such directions compared to gradient descent. Nesterov's accelerated gradient (abbrv. NAG; Nesterov, 1983) is a first-order optimization method which is proven to have a better convergence rate guarantee than gradient descent for general convex functions with Lipshitz-continuous derivatives ($O(1/T^2)$ versus $O(1/T)$). Upon close examination, Nesterov's accelerated gradient turns out to be very similar to standard momentum, differing only in two important aspects. Firstly, it prescribes a particular formula for the learning rate ε and momentum constant μ, which are necessary to achieve its theoretical guarantees for smooth convex functions. However, these turn out to be far too aggressive for our nonconvex objectives (especially with a stochastic gradient signal), and we have found that the problem of choosing ε and μ is better addressed by manual tuning (at least until a more robust automatic system is created). The second difference, which seems minor but turns out to be important, is the precise update mechanism for the velocity vector v.

Standard momentum is given by the following pair of equations:

$$v_t = \mu v_{t-1} - \varepsilon \nabla f(\theta_{t-1}) \tag{7.1}$$

$$\theta_t = \theta_{t-1} + v_t \tag{7.2}$$

where the variable $\varepsilon > 0$ is the learning rate, $\mu \in [0, 1]$ is the momentum constant (or $1 - \mu$ is the constant of friction), and $\nabla f(\theta_t)$ is an unbiased estimate of the gradient at θ_t. The momentum constant μ controls the "decay" of the velocity vector v, and values closer to 1 result in gradient information persisting across more iterations which generally leads to higher velocities.

Nesterov's accelerated gradient is an iterative algorithm that was originally derived for non-stochastic gradients. It is initialized by setting $k = 0$, $a_0 = 1$, $\theta_{-1} = y_0$, y_0 to an arbitrary parameter setting, z to an arbitrary parameter setting, and $\varepsilon_{-1} = \|y_0 - z\|/\|\nabla f(y_0) - \nabla f(z)\|$. It then repeatedly updates its parameters with the following equations:

$$\varepsilon_t = 2^{-i}\varepsilon_{t-1} \quad \text{(here } i \text{ is the smallest positive integer for which } f(y_t) - f(y_t - 2^{-i}\varepsilon_{t-1}\nabla f(y_t)) \geq 2^{-i}\varepsilon_{t-1}\frac{\|\nabla f(y_t)\|^2}{2}\text{)} \tag{7.3}$$

$$\theta_t = y_t - \varepsilon_t \nabla f(y_t) \tag{7.4}$$

$$a_{t+1} = \left(1 + \sqrt{4a_t^2 + 1}\right)/2 \tag{7.5}$$

$$y_{t+1} = \theta_t + (a_t - 1)(\theta_t - \theta_{t-1})/a_{t+1} \tag{7.6}$$

The above presentation is relatively opaque and could be difficult to understand, so we will rewrite these equations in a more intuitive manner.

The learning rate ε_t is adapted to always be smaller than the reciprocal of the "observed" Lipshitz coefficient of ∇f around the trajectory of the optimization. Alternatively, if the Lipshitz coefficient of the derivative is known to be equal to L, then setting $\varepsilon_t = 1/L$ for all t is sufficient to obtain the same theoretical guarantees. This method for choosing the learning rate assumes that f is not noisy, and will result in too-large learning rates if the objective is stochastic.

To understand the sequence a_t, we note that the function $x \to \left(1 + \sqrt{4x^2 + 1}\right)/2$ quickly approaches $x \to x + 0.5$ from below as $x \to \infty$, so $a_t \approx (t + 4)/2$ for large t, and thus $(a_t - 1)/a_{t+1}$ (from eq. 7.6) behaves like $1 - 3/(t + 5)$.

Finally, if we define

$$v_t \equiv \theta_t - \theta_{t-1} \tag{7.7}$$

$$\mu_t \equiv (a_t - 1)/a_{t+1} \tag{7.8}$$

then the combination of eqs. 7.6 and 7.8 implies:

$$y_t = \theta_{t-1} + \mu_{t-1}v_{t-1} \tag{7.9}$$

which can be used to rewrite eq. 7.4 as follows:

$$\theta_t = \theta_{t-1} + \mu_{t-1}v_{t-1} - \varepsilon_{t-1}\nabla f(\theta_{t-1} + \mu_{t-1}v_{t-1}) \tag{7.10}$$

$$v_t = \mu_{t-1}v_{t-1} - \varepsilon_{t-1}\nabla f(\theta_{t-1} + \mu_{t-1}v_{t-1}) \tag{7.11}$$

where eq. 7.11 is a consequence of eq. 7.7. Alternatively:

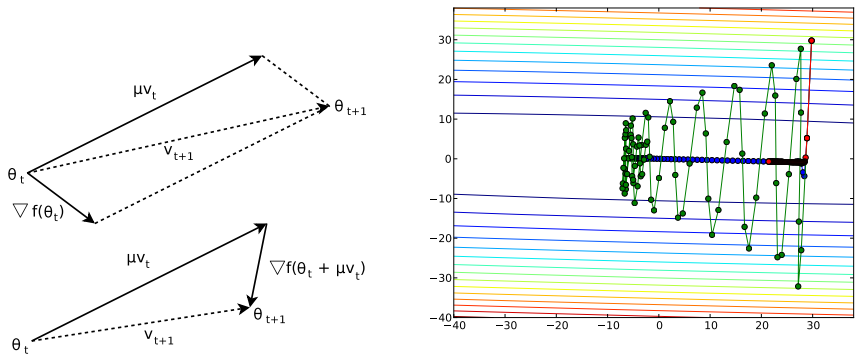
$$v_t = \mu_{t-1}v_{t-1} - \varepsilon_{t-1}\nabla f(\theta_{t-1} + \mu_{t-1}v_{t-1}) \tag{7.12}$$

$$\theta_t = \theta_{t-1} + v_t \tag{7.13}$$

where $\mu_t \approx 1 - 3/(t + 5)$. Nesterov (1983) shows that if f is a convex function with an L-Lipshitz continuous derivative, then the above method satisfies the following:

$$f(\theta_t) - f(\theta^*) \leq \frac{4L\|\theta_{-1} - \theta^*\|^2}{(t + 2)^2} \tag{7.14}$$

The quadratic dependence on the number of iterations can be intuitively understood by considering a one-dimensional linear function, where the magnitude of the i-th step would be proportional to i. Thus, only $O(\sqrt{N})$ of these steps are required to traverse N units of distance.



(a) **(top)** Momentum **(bottom)** Nesterov's accelerated gradient

(b) Nesterov and Momentum on a quadratic

Figure 7.1: **Left**: one step of Nesterov's accelerated gradient and of momentum. **Right**: the trajectories of GD (red), momentum (green), and Nesterov's accelerated gradient (blue). Both methods had μ set to 0.95. The global minimum of the quadratic is in the center of the figure, at $(0, 0)$.

When Nesterov's accelerated gradient is presented in eqs. 7.12-7.13, the similarity to eqs. 7.1-7.2 that define the standard momentum becomes apparent. If we ignore the proposed momentum schedule μ_t and the adaptive learning rate schedule, then the key difference between momentum and Nesterov's accelerated gradient is that momentum computes the gradient before applying the velocity, while Nesterov's accelerated gradient computes the gradient after doing so.

This benign-looking difference seems to allow Nesterov's accelerated gradient to change v in a quicker and more responsive way, letting it behave more stably than momentum in many situations, especially for higher values of μ. Indeed, consider the situation where the addition of μv_t to θ_t results in an undesirable increase in the objective f. In the case of Nesterov's accelerated gradient, the gradient correction to the velocity v_t is computed at point $\theta_t + \mu v_t$, and if μv_t is indeed a poor update, $\nabla f(\theta_t + \mu v_t)$ will point back towards θ_t more strongly than the gradient computed at θ_t (which is used by momentum), thus providing a larger and more timely correction to v. See fig. 7.1(a) for a diagram which illustrates this phenomenon geometrically. Now while each iteration of Nesterov's accelerated gradient may only be slightly more effective than momentum at correcting a large and inappropriate velocity, this difference in effectiveness may compound as the algorithms iterate. To demonstrate this we applied both Nesterov's accelerated gradient and momentum to a two-dimensional oblong quadratic objective, both with the same values for μ and ε (see fig. 7.1(b)). While the optimization path taken by momentum exhibits large oscillations along the high-curvature vertical direction, Nesterov's accelerated gradient is able to avoid these oscillations completely, confirming the intuition that it is much more effective than momentum at decelerating over the course of multiple iterations, and that this results in improved stability. Also note that there is little difference in the progress of both Nesterov's accelerated gradient and momentum along the horizontal direction of the quadratic, due to the fact that the gradient changes only very slowly along this direction (due to its low curvature).

We emphasize that this discussion does not prove that Nesterov's accelerated gradient will perform significantly better than standard momentum in all cases. However, due to its simplicity and close resemblance to standard momentum it is very easy to implement, and may provide a significant boost in performance, as will be evidenced in our experiments.

name	dim	size	objective	network architecture
CURVES	784	20,000	binary cross entropy	784-400-200-100-50-25-6
MNIST	784	60,000	binary cross entropy	784-1000-500-250-30
FACES	625	103,500	squared error	625-2000-1000-500-30

Table 7.1: A description of the networks' architectures and the sizes of the datasets.

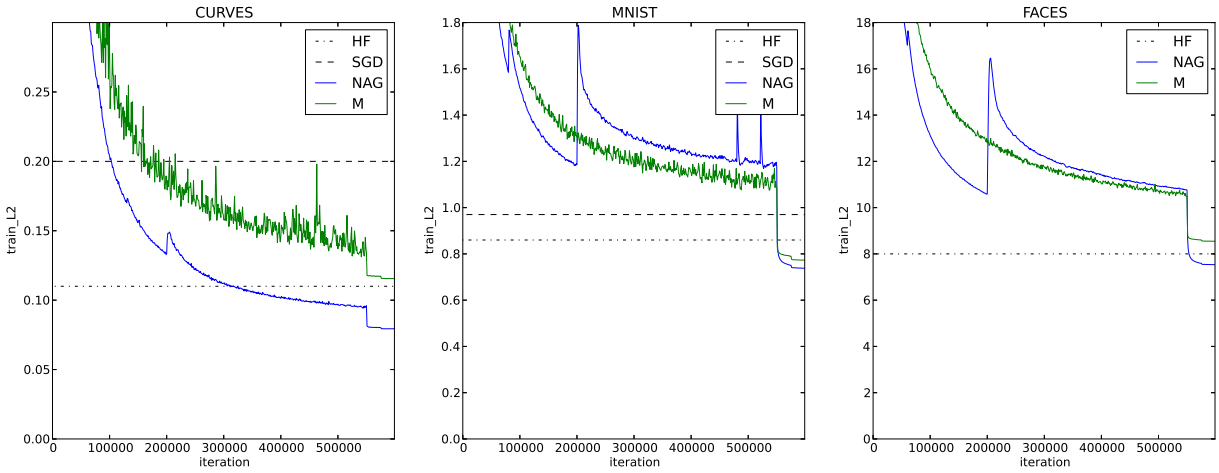


Figure 7.2: The figure shows the learning curves of Nesterov's accelerated gradient (NAG; blue) and of momentum (M; green) on the three autoencoder tasks. The μ-schedule of momentum is less aggressive than that of Nesterov's accelerated gradient. The "spikes" in the error correspond to the iterations at which the momentum is increased. Similarly, the rapid decrease in the objective near the end of the optimization is due to the reduction of both ε and μ. In these figures, the momentum experiments had the same μ-schedule as the corresponding Nesterov's accelerated gradient experiment, except that μ was capped at 0.99.

7.3 Deep Autoencoders

The task of the neural network autoencoder is to reconstruct its own input subject to the constraint that one of its hidden layers is restricted to be of low-dimension. This "bottleneck" layer acts as a low-dimensional code for the original input, similar to other dimensionality reduction techniques like Principle component Analysis (PCA). Hinton and Salakhutdinov (2006) trained very deep autoencoders on several datasets (as described in table 7.1), achieving results that greatly surpassed those of PCA both numerically and qualitatively. These autoencoders are the deepest neural networks with published results, ranging between 7 and 11 layers. Unlike Martens (2010), we did not use L2 regularization, so we rerun HF without L2 regularization as well.

Our main results are presented in figure 7.2 and table 7.2. The figure shows the learning curves of the deep autoencoders on the CURVES, MNIST, and FACES datasets. We examine only the training error in order to focus purely on the effectiveness of the optimization and to avoid the confounding factor of overfitting. Using Nesterov's accelerated gradient we achieved the lowest published errors for this set of problems, including those in Martens (2010). For Nesterov's accelerated gradient we used a more aggressive μ-schedule than for momentum because the more aggressive schedule caused momentum to

diverge, which further supports our claim that Nesterov's accelerated gradient is more robust to higher values of μ. The μ-schedule we used consists of the progression of values [0.8, 0.9, 0.95, 0.97, 0.99, 0.995, 0.997, 0.999] placed (roughly) at exponentially spaced iterations. We used fixed learning rates of 0.005 for CURVES and MNIST and 0.0005 for FACES and did not use weight decay. For each first-order method we used 600,000 iterations with randomly sampled minibatches of size 200. By 200,000 iterations μ reaches its maximum value of 0.999, and notably, the learning curve for Nesterov's accelerated gradient exhibits large spikes starting at precisely this iteration (for MNIST and FACES). It would seem that the sudden increase in μ destabilizes the optimization temporarily, possibly due to the "desire" of larger μ to have a wider minima.

We found it beneficial to reduce ε by a factor of 10 and μ down to 0.99 during the final 10% of the optimization. Doing so resulted in the following improvements in performance: 0.96 to 0.079 for CURVES, 1.20 to 0.73 for MNIST, and 10.83 to 7.52 for faces, and this effect is present for both Nesterov's accelerated gradient and for momentum (see fig. 7.2). It appears that reducing these constants allows for finer convergence to take place whereas otherwise the overly aggressive nature of the momentum method (especially with stochastic gradients) would prevent this. It may be tempting then to use lower values of ε and μ from the outset, or to reduce them immediately when progress in reducing the error appears to slow down. However, in our experiments we found that doing this was detrimental in terms of the final errors we could achieve, and that despite appearing to not make much progress, or even becoming significantly non-monotonic, the optimizers were doing something useful over these extended periods of time at higher values of ε and μ.

A speculative explanation as to why we see this behavior is as follows. With large constants the momentum methods are making useful progress along slow-changing directions of low-curvature even while appearing to not reduce the error very significantly, due to their failure to converge in the more turbulent high curvature directions. This progress takes them to new regions of the parameter space where "finer" and more careful optimization will allow convergence in the high-curvature directions and quickly zero-in on the nearest local solution (which because of the previous progress made in the low curvature directions will be of superior quality). Moving to this fine convergence regime by reducing the constants too early may make it difficult for the optimization to make significant progress along the low-curvature directions, since first-order methods, when lacking momentum or a large learning rate, are notoriously bad at this (which is one of the motivations for second-order methods like HF for deep learning). Another reason why progress along low-curvature directions may be hampered in this situation, which is particular to non-convex objectives, is that the finer and more careful optimization may more easily get stuck in small/narrow regions of the parameter space that are highly constrained by local high-curvature directions (appearing as small ripples or cracks in the error surface).

problem	NAG	M (0.99)	M (0.9)	SGD ($3 \cdot 10^6$ iters)	SGD (Glorot and Bengio, 2010)	HF
CURVES	0.078	0.110	0.220	0.250	0.160	0.110
MNIST	0.730	0.770	0.990	1.100	0.900	0.780
FACES	7.500	8.530	10.970	14.700	NA	8.000

Table 7.2: The results on the three autoencoders for the various methods. momentum (0.99) and momentum (0.9) refer to momentum where the value of the momentum is capped at 0.99 and 0.9, respectively. We have conducted an experiment with SGD for 3 million iterations with a minibatch of size 20 and a learning rate of size 0.02 or (0.002 for FACES). Even with six times more iterations SGD failed to reach the performance of Nesterov's accelerated gradient and momentum.

Unlike the CURVES and FACES autoencoder tasks, the MNIST one did not see much benefit from

the use of Nesterov's accelerated gradient over momentum. Thus we cannot infer from our results that Nesterov's accelerated gradient will likely always do better than momentum, but it seems unlikely that it would ever do much worse.

7.3.1 Random initializations

The above results were obtained with sigmoid neural networks that were initialized with the sparse initialization technique (abbrv. SI) described in Martens (2010). In this scheme, for each neural unit, 15 randomly chosen incoming connection weights are initialized by a unit Gaussian draw, while the remaining ones are set to zero. The intuitive justification is that the total amount of input to each unit will not depend on the size of the previous layer and hence they will not as easily saturate. When applying this method to networks with tanh units it is helpful to further rescale weights by 0.25, and shift the biases by -0.5 (so that the average output of each unit is closer to zero).

Glorot and Bengio (2010) designed an initialization method for deep networks with tanh units. In this scheme, each weight is drawn from $\sqrt{6/(n_1+n_2)} \cdot U[-1,1]$, where n_1 and n_2 are the number of incoming and outgoing units for that particular weight matrix, and $U[-1,1]$ is the uniform distribution over the interval $[-1,1]$. The scaling factor $\sqrt{6/(n_1+n_2)}$ is chosen so as to preserve variance of the activations of the units from layer to layer. Chapelle and Erhan (2011) used this initialziation method to achieve non-trivial performance on the CURVES and the MNIST autoencoder tasks using SGD optimization (these are reported in table 7.2).

We conducted a number of experiments on the CURVES problem training problem, comparing networks using tanh units against ones using sigmoids, as well as comparing SI with the variance preserving initialization method (VP). We found that the tanh networks required a learning rate 8 times smaller, or else learning would become too unstable with our aggressive μ-schedule. After 600,000 parameter updates, a tanh network initialized with SI achieved a training error of 0.086, while an identically defined tanh network initialized with VP achieved a training error of 0.098. Comparing to our results using SI (fig. 7.3), these results suggests that tanh networks are not significantly harder or easier to train than sigmoid networks, and that VP seems to perform worse than SI for our particular models and the optimizers we are using.

We also investigated the performance of the optimization as a function of the scale constant used in SI (which defaults to 1 for sigmoid units). We found that SI works reasonably well if it is rescaled by a factor of 2, but leads to noticeable (but not severe) slow down when scaled by a factor of 3. When we rescaled by factor of 1/2 or 5 we were unable to achieve sensible results. These findings are summarized in fig. 7.3.

We emphasize that the above random initializations, SI and VP, require no tuning and are easy to use with any sigmoid or tanh deep neural network, and that these results are typical for multiple random seeds.

7.3.2 Deeper autoencoders

We conducted an experiment with an autoencoder with 17 hidden layers on the CURVES autoencoder (versus the usual 11). The architecture was obtained by adding layers of size 200 so that the sizes of the layers of both the decoder and the encoder are monotonic. We have used the sparse initialization and precisely the same learning rate and μ-schedule as in our previous experiment. The deeper network has 30% more parameters, and it has achieved an error of 0.067 after 600,000 iterations (while the shallower 11-hidden layer autoencoder achieved an error of 0.078). See fig. 7.3 (rightmost figure).

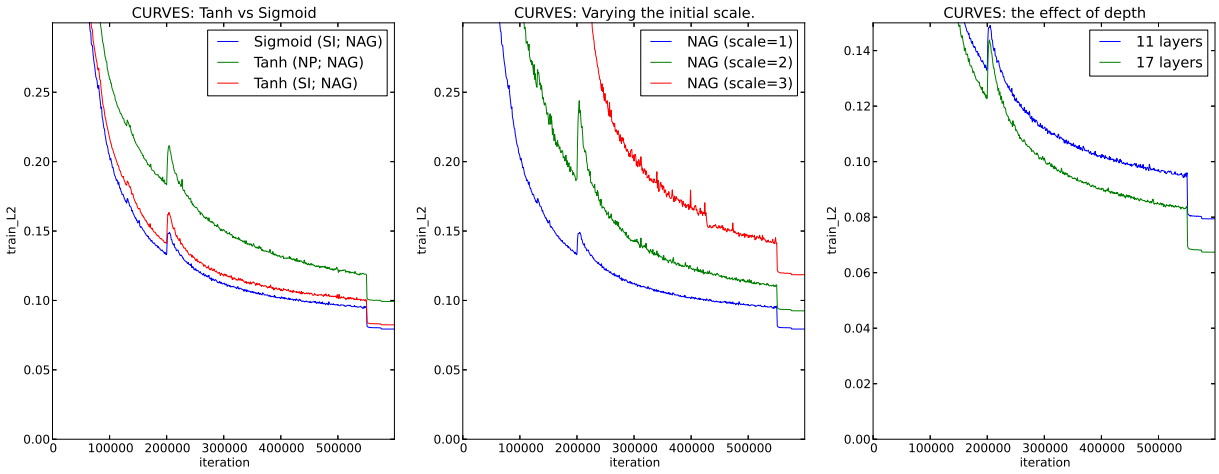


Figure 7.3: **Left**: An experiment comparing tanh to logistic sigmoid units. **Center**: The sensitivity to the initial scale. The initial scales of 0.5 and 5 resulted in error curves that were too large to be visible on this plot. **Right**: The learning curve of the 17-layer neural network.

7.4 Recurrent Neural Networks

problem	Nesterov's accelerated gradient	Momentum
add $T = 80$	0.011	0.250
mul $T = 80$	0.270	0.340
mem-5 $T = 200$	0.025	0.180
mem-20 $T = 50$	0.032	0.00006

Table 7.3: Results on the RNN toy problems with the pathological long range temporal dependencies. Every experiment was run 10 times with a different random initialization. Every single experiment resulted in a non-trivial solution that "established communication" between the inputs and the targets and achieved a relatively low prediction error. For each experiment, we compute the average zero-one loss on a test set (that measures the number of incorrectly remembered bits for the memorization problems, and the proportion of sequences whose prediction was further than 0.04 from the correct answer, for the addition and the multiplication problems).

7.4.1 The initialization

In Chapter 4 we found that a good initialization is essential for achieving nontrivial performance on the various artificial long-term dependency tasks from Hochreiter and Schmidhuber (1997). Without one, no information will be conveyed by the hidden units over a long enough distance, a situation which is unrecoverable for any local optimization method trying to train RNNs to solve these tasks.

We used an initialization inspired by the one used in ESNs for RNNs with tanh units, where the recurrent connections are initialized with random sparse connectivity (so 15 incoming weights to each

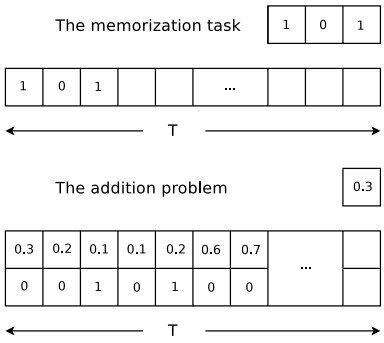


Figure 7.4: A diagram of the memorization and the addition problems.

unit are sampled from a unit Gaussian and the rest are set to 0), which are then rescaled so that the absolute value of the largest eigenvalue of the recurrent weight matrix is equal to 1.2. We found that nearby values of 1.1 and 1.3 resulted in less robust behavior of the optimization. The input-to-hidden connections also used the sparse initialization. However, we found that for the addition and the multiplication problem, the best results are obtained by rescaling these connections by 0.02, while the memorization problems worked best when the scale of these connections was set to 0.3. While learning is not sensitive to the precise values of the scale parameters of the input-to-hidden connections, it is sensitive to their order of magnitude.

As discussed in section 2.8.4, the value of the largest eigenvalue of the recurrent connection matrix has a considerable influence on the dynamics of the RNN's hidden states. If it is smaller than 1, then the hidden state dynamics will have a tendency to move towards one of only a few attractor states, quickly "forgetting" whatever input signal they may have been exposed to. In contrast, when the largest eigenvalue is slightly greater than 1, then the RNN's hidden state produces responses that are sufficiently varied for different input signals, thus allowing information to be retained over many time steps. Jaeger and Haas (2004) also prescribe the weight matrix to be sparsely connected, which causes the hidden state to consist of multiple "loosely-coupled oscillators". The ESN initialization was designed to work specifically with tanh hidden units, and we indeed found that our ESN-inspired initialization worked best with these units as well, and so we used these units in all of our experiments.

We considered four of the pathological long-term dependency problems from Hochreiter and Schmidhuber (1997) and on which we trained RNNs with 100 tanh hidden units (as was used by Martens and Sutskever (2011) and Chapter 4). These were the 5-bit memorization task, the 20-bit memorization task, the addition problem, and the multiplication problem. These artificial problems were each designed to be difficult or impossible to learn with regular RNNs using standard optimization methods, owing to the presence of extremely long-range temporal dependencies of the target outputs on the early inputs. And with the possible exception of the 5-bit memorization problem, they cannot be learned with an ESN that has only 100 hidden units that do not make use of explicit temporal integration (Jaeger, 2012a).

7.4.2 The problems

The addition, multiplication, the 5-bit and the 20-bit memorization problems are presented in section 4.4. To achieve low error on each of these tasks the RNN must learn to memorize and transform some information contained at the beginning of the sequence within its hidden state, retaining it all of the way to the end. This is made especially difficult because there are no easier-to-learn short or medium-term dependencies that can act as hints to help the learning see the long-term ones.

We found the RNNs to be more sensitive than feedforward neural networks to the choice schedules

for ε and μ. The addition and the multiplication problems were particularly sensitive to the schedule for ε. For these problems, ε was set to 3e-5 for the first 1500 iterations, 3e-4 for another 1500, 3e-3 for another 3000, 1e-3 for another 24000 iterations, and 1e-4 for the remainder. μ was set to 0.9 for the first 4000 iterations, and then 0.98 for the remainder. Every RNN was given 50,000 iterations. These particular schedules resulted in reliable performance on both the addition and the multiplication problem, while some other choices we tried would sometimes cause the optimization to fail to establish the necessary long-term "communication" between the input units and targets. In contrast, the memorization problems were considerably more robust to the learning rate and the momentum schedule. The only requirement on the schedule for ε was that it needed to be reduced to around to 0.00001 fairly early on (after 1500 iterations) in order to prevent optimization from diverging due to exploding gradients.

The results of our RNN experiments are presented in table 7.3. They show that after 50,000 parameter updates of either momentum or Nesterov's accelerated gradient on minibatches of size 100 (or 32 for the 5-bit memorization problem), the RNN is able to achieve very low errors on these problems, which are at a level only achievable by modeling the long-term dependencies. We did not observe a single dramatic failure over 10 different random initializations, implying that our approach is reasonably robust.

The numbers in the table are the average loss over 10 different random seeds. Instead of reporting the loss being minimized (which is the squared error or cross entropy), we report an easier to interpret zero-one loss: for the bit memorization, we report the fraction of timesteps that are predicted incorrectly. And for the addition and the multiplication problems, we report the fraction of cases where the RNN is wrong by more than 0.04. Our results also suggest that Nesterov's accelerated gradient is better suited than momentum for these problems.

7.5 Discussion

This chapter demonstrates that momentum methods, especially Nesterov's accelerated gradient, can be configured to reach levels of performance that were previously believed to be attainable only with powerful second-order methods such as HF. In particular, these results make it seem that the work in Chapter 4 is no longer relevant, but it is not so because HF can train RNNs to solve harder instances of these problems starting from initializations of lower quality. It is therefore likely that the RNN results of this chapter could be considerably improved if RNNs are used.

The simplicity of the initializations and of momentum methods makes it easier for practitioners to use deep and recurrent neural networks in their applications. But while the initializations are straightforward to implement correctly, momentum methods can be cumbersome to use because of the need to tune the μ- and the ε-schedules.

An interesting approach for automatically choosing μ is described by O'Donoghue and Candes (2012), who show that the momentum schedule of Nesterov's accelerated gradient is suboptimal for strongly convex functions. More specifically, if f is σ-strongly convex and ∇f is L-Lipshitz, then the optimal momentum is the constant $\mu^* = \frac{1-\sqrt{\sigma/L}}{1+\sqrt{\sigma/L}}$, which makes Nesterov's accelerated gradient converge at a faster rate:

$$f(\theta_t) - f(\theta^*) \leq L \left(1 - \sqrt{\frac{\sigma}{L}}\right)^t \|\theta_0 - \theta^*\|^2 \tag{7.15}$$

To compute the optimal momentum, we need to estimate both L and σ, and while L is relatively easy to estimate (in fact, Nesterov (1983) does precisely this), it is apparently difficult to estimate σ.

O'Donoghue and Candes (2012) observed that the sequence $f(\theta_t)$ is non-monotonic and that its largest frequency component has a period proportional to $\sqrt{L/\sigma}$. This makes it possible to estimate

$\sqrt{L/\sigma}$ by starting Nesterov's accelerated gradient (with the usual momentum of eq. 7.8), and reporting the iteration at which $f(\theta_t)$ increases. They also show that if Nesterov's accelerated gradient is restarted (i.e., v_k is set to zero) every $\sqrt{L/\sigma}$ iterations (approximately), then it converges at a rate similar to that in eq. 7.15 (this is proven with a technique similar to the one in Chap. 2, footnote 3). Therefore, a method that restarts the momentum whenever the objective $f(\theta_t)$ increases will attain the rate of eq. 7.15 without prior knowledge of L and σ.

This technique completely solves the problem of determining μ for continuously-differentiable convex functions, so it seems plausible that it could also be used for deep and recurrent neural networks. Unfortunately, it was difficult to determine the period of $f(\theta_t)$ because of the stochasticity of the mini-batches. It also did not perform well when we eliminated the stochasticity, possibly due to the non-convexity of the neural network objective, which strongly violates the assumptions of the method.

It is possible to determine the momentum parameters using automatic hyper-parameter optimization packages such as Bayesian optimization (Snoek et al., 2012) or random search (Bergstra and Bengio, 2012). These methods can achieve very high performance, but they require multiple learning trials to explore the space of hyper-parameter.

Our results may appear to contradict the existence of the vanishing and the exploding gradients in RNNs, but on closer inspection it is not the case. The carefully tuned scale of our random initialization is able to prevent the gradients from vanishing and exploding too severely at the beginning, and our careful choices of ε and μ help the optimization avoid causing exploding and vanishing gradients to develop to some extent. Our method is only partially successful at not causing exploding gradients: they occur in the bit-memorization problems (which is addressed by using a very small learning rate), and we are unable to train RNNs to solve these problems for values of T much larger than those reported in table 7.3.

Our results also illustrate the importance of momentum. An aggressive momentum schedule was required in order to attain HF-level performance on the deep autoencoders; we were unable to reach non-trivial performance without momentum on the addition and the multiplication problems, and were only able to obtain the lowest error on the deep autoencoding tasks by using a very large momentum constant (of 0.999) combined with Nesterov-type updates. This can be explained by the difficult curvature of the deep neural networks (as suggested by Martens (2010)), which benefits from larger momentum values.

Chapter 8

Conclusions

8.1 Summary of Contributions

The goal of this thesis was to understand the nature of the difficulty of the RNN learning problem, to develop methods that mitigate this difficulty, and to show that the RNN is a viable sequence model that can achieve excellent performance in challenging domains: character-level language modelling and motor control. Our training methods make it possible to train RNNs to solve tasks that have pathological long-term dependencies, which was considered completely unsolvable by RNNs prior to our work (Bengio et al., 1994). Our final result shows that properly initialized RNNs can be trained by momentum methods, whose simplicity makes it easier to use RNNs in applications.

Chapter 3 investigated RNN-based probabilistic sequence models. We introduced the TRBM, a powerful RBM-inspired sequence model, and presented an approximate parameter update that could train the TRBM to model complex and high-dimensional sequences. However, the update relied on crude approximations that prevented it from explicitly modelling the longer-term structure of the data. We showed that a minor modification of the TRBM yields the first RNN-RBM hybrid, which has similar expressive power but whose parameter update is easy to compute with BPTT and CD. Although the RTRBM was still difficult to train (because it is an RNN), its hidden units learned to store relevant information over time, and thus allowed it to represent temporal structure considerably more faithfully than the TRBM.

In Chapter 4, we attempted to directly address the difficulty of the RNN learning problem using powerful second-order optimization. We showed that Hessian-free optimization with a novel "structural damping" technique can train RNNs on problems that have long-term dependencies spanning 200 timesteps. And while HF does not completely solve the RNN learning problem (since it fails as the span of the long-term dependencies increases), it performs well if the long-term dependencies span no more than 200 timesteps. Prior to this work, the learning problem of standard RNNs on these tasks was believed to be completely unsolvable, although we empirically found it difficult to train RNNs on long range dependencies that span much more than 200 timesteps.

We then sought to use the Hessian-free optimizer to train RNNs on challenging problems. Chapter 5 used HF to train RNNs on character-level language modelling. The RNN language model achieved high performance, and learned to balance parentheses and quotes over hundreds of characters. This is an example of a naturally-occurring long-term dependency that is impossible to model with all other existing language model due to their inability to utilize contexts of such length.

In Chapter 6 we used HF to train RNNs to control a two-dimensional arm to solve reaching tasks. Although controlling such an arm is easy because it is low-dimensional and fully-actuated, the problem we considered was made difficult by the presence of the delayed feedback and unpredictable distur-

bances that prevent simpler control methods from achieving high performance. This setting is of interest because the motor cortex is believed to operate under these conditions. Delayed feedback makes the problem difficult because a high-quality controller must use a model of the future in order to, for instance, stop the arm at the target without overshooting. The unpredictable disturbances are similarly difficult to handle, because a good controller must rapidly estimate and then counteract the disturbance by observing the manner in which the arm responds to the muscle commands.

Chapter 7 demonstrated that RNNs are considerably easier to train than was previously believed, provided they are properly initialized and are trained with a simple momentum method whose learning parameters are well-tuned. RNNs were believed to be impossible to train with plain momentum methods, even after the results of Chapter 4 were published. This is significant because momentum methods are relatively easy to implement, which makes it likely that RNNs (and deep neural networks) will be used in practice more frequently.

8.2 Future Directions

There are several potential problems with RNNs that would be worthwhile to investigate.

Faster learning. A fundamental problem is the high cost of computing gradients on long sequences. This makes it impossible to use the large number of parameter updates needed to achieve high performance. While the computation of the gradient of a single long sequence requires the same number of FLOPs as that of the gradients of many shorter sequences, the latter can be easily parallelized on a GPU and is therefore much more cost-effective. Hence, it is desirable to develop methods that approximate derivatives over long sequences quickly and efficiently.

Representational capacity. Representationally, our RNNs may have failed to represent the "semantic" longer-term structure of natural text because of the relatively small size of their hidden states. Any RNN-like method that can model the high-level long-range structure of text must have a much larger hidden state, because natural text describes many events and entities that must be remembered if the text is to be modelled correctly. A similar effect occurs in melodies, which tend to be repetitive in subtle ways. Such long-range structure can be captured only with models that can remember the repetitions and references made in a given sequence. It is therefore important to develop RNNs with much larger hidden states that perform more computation at each timestep, in order to obtain a model that is not *obviously incapable* of representing this structure.

Weights as states. We can vastly increase the hidden state of the RNN by introducing special rapidly-changing connections and treating them as part of the state (Mikolov et al., 2010; Tieleman and Hinton, 2009); this approach approximately squares the size of the hidden state (if the hidden-to-hidden matrix is fully connected) and allows the RNN to remember much more information about a given sequence. The challenge is to develop practical methods that can train such RNNs to make effective use of their rapidly-changing connections.

The ultimate goal is to develop practical RNN models that can represent and learn the extremely complex long-range structure of natural text, music, and other sequences. But doing so will require new ideas.

Bibliography

Agrawal, S. (1991). Inertia matrix singularity of planar series-chain manipulators. In *Proceedings of the 1991 IEEE Intemational Conference on Robotics and Automation*, pages 102–107, Sacramento, California.

Baur, W. and Strassen, V. (1983). The complexity of partial derivatives. *Theoretical computer science*, 22(3):317–330.

Bell, A. and Sejnowski, T. (1995). An Information-Maximization Approach to Blind Separation and Blind Deconvolution. *Neural Computation*, 7(6):1129–1159.

Bell, R., Koren, Y., and Volinsky, C. (2007). The BellKor solution to the Netflix prize. *KorBell Team's Report to Netflix*.

Bengio, Y. (1991). *Neural Networks and Markovian Models*. PhD thesis, McGill University.

Bengio, Y. (2009). Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127.

Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007). Greedy layer-wise training of deep networks. In *In NIPS*. MIT Press.

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

Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13:281–305.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*.

Bottou, L., Bengio, Y., and Le Cun, Y. (1997). Global training of document processing systems using graph transformer networks. In *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, pages 489–494. IEEE.

Boulanger-Lewandowski, N., Bengio, Y., and Vincent, P. (2012). Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. *arXiv preprint arXiv:1206.6392*.

Boulanger-Lewandowski, N., Vincent, P., and Bengio, Y. (2011). An energy-based recurrent neural network for multiple fundamental frequency estimation.

Boyen, X. and Koller, D. (1998). Tractable inference for complex stochastic processes. In Cooper, G. and Moral, S., editors, *Proc. of the 14th Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 33–42, San Francisco. Morgan Kaufmann.

Brown, A. and Hinton, G. (2001). Products of hidden markov models. *Proceedings of Artificial Intelligence and Statistics*, pages 3–11.

Chapelle, O. and Erhan, D. (2011). Improved Preconditioner for Hessian Free Optimization. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*.

Cheng, E. and Scott, S. (2000). Morphometry of Macaca Mulatta Forelimb. i. shoulder and elbow muscles and segment inertial parameters. *J. Morphol.*, 245:206–224.

Cook, J., Sutskever, I., Mnih, A., and Hinton, G. (2007). Visualizing similarity data with a mixture of maps. In *Proceedings of the 11th International Conference on Artificial Intelligence and Statistics*, volume 2, pages 67–74. Citeseer.

Dahl, G., Yu, D., Deng, L., and Acero, A. (2012). Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *Audio, Speech, and Language Processing, IEEE Transactions on*, 20(1):30 –42.

Datar, M., Immorlica, N., Indyk, P., and Mirrokni, V. (2004). Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM.

Dung, L. T., Komeda, T., and Takagi, M. (2008). Reinforcement learning for POMDP using state classification. *Applied Artificial Intelligence*, 22(7-8):761–779.

Elman, J. (1990). Finding structure in time. *Cognitive science*, 14(2):179–211.

Gasthaus, J., Wood, F., and Teh, Y. (2010). Lossless compression based on the Sequence Memoizer. In *Data Compression Conference (DCC), 2010*, pages 337–345. IEEE.

Ghahramani, Z. and Hinton, G. (2000). Variational learning for switching state-space models. *Neural Computation*, 12(4):831–864.

Ghahramani, Z. and Jordan, M. (1997). Factorial hidden markov models. *Machine Learning*, 29(4):245–273.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of AISTATS 2010*, volume 9, pages 249–256.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Domain Adaptation for Large-Scale Sentiment Classification: A Deep Learning Approach. In *ICML-2011*.

Graves, A., Fernández, S., Gomez, F., and Schmidhuber, J. (2006). Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376. ACM.

Graves, A., Fernández, S., Liwicki, M., Bunke, H., and Schmidhuber, J. (2008). Unconstrained online handwriting recognition with recurrent neural networks. *Advances in Neural Information Processing Systems*, 20:1–8.

Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, 18(5-6):602–610.

Graves, A. and Schmidhuber, J. (2009). Offline handwriting recognition with multidimensional recurrent neural networks. In Koller, D., Schuurmans, D., Bengio, Y., and Bottou, L., editors, *Advances in Neural Information Processing Systems 21*, pages 545–552.

Hinton, G. (1978). *Relaxation and its role in vision*. PhD thesis, University of Edinburgh.

Hinton, G. (2002). Training Products of Experts by Minimizing Contrastive Divergence. *Neural Computation*, 14(8):1771–1800.

Hinton, G., Osindero, S., and Teh, Y. (2006). A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18(7):1527–1554.

Hinton, G. and Salakhutdinov, R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313:504–507.

Hinton, G., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *Arxiv preprint arXiv:1207.0580*.

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen netzen. diploma thesis, institut für informatik, lehrstuhl prof. brauer, technische universität münchen.

Hochreiter, S., Bengio, Y., Frasconi, P., and Schmidhuber, J. (2001). *A Field Guide to Dynamical Recurrent Neural Networks*, chapter Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. IEEE press.

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

Huh, D. and Todorov, E. (2009). Real-time motor control using recurrent neural networks. In *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 42–49.

Hutter, M. (2006). The Human Knowledge Compression Prize.

Ihler, A., Fisher, III, J., Moses, R. L., and Willsky, A. S. (2004). Nonparametric belief propagation for self-calibration in sensor networks. In *Proceedings of the third international symposium on Information processing in sensor networks (IPSN-04)*, pages 225–233, New York. ACM Press.

Isard, M. and Blake, A. (1996). Contour tracking by stochastic propagation of conditional density. In *Proceedings of the 4th European Conference on Computer Vision*. Springer-Verlag.

Jaeger, H. (2000). Observable Operator Models for Discrete Stochastic Time Series. *Neural Computation*, 12(6):1371–1398.

Jaeger, H. (2012a). Personal Communication.

Jaeger, H. (2012b). Long Short-Term Memory in Echo State Networks: Details of a simulation study. Technical Report 27, Jacobs University Bremen - School of Engineering and Science.

Jaeger, H. and Haas, H. (2004). Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667):78.

Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2009). What is the best multi-stage architecture for object recognition? In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 2146–2153. IEEE.

Johnson, W. and Lindenstrauss, J. (1984). Extensions of Lipschitz mappings into a Hilbert space. *Contemporary mathematics*, 26(189-206):1–1.

Jordan, M., Ghahramani, Z., Jaakkola, T., and Saul, L. (1999). An Introduction to Variational Methods for Graphical Models. *Machine Learning*, 37(2):183–233.

Kearns, M. and Vazirani, U. (1994). *An introduction to computational learning theory*. MIT Press.

Krizhevsky, A. (2010). Convolutional deep belief networks on cifar-10. *Unpublished manuscript*.

Krizhevsky, A. and Hinton, G. (2011). Using very deep autoencoders for content-based image retrieval. In *European Symposium on Artificial Neural Networks ESANN-2011, Bruges, Belgium.*

Krizhevsky, A., Sutskever, I., and Hinton, G. (2012). ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems (NIPS)*.

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

Li, W. and Todorov, E. (2004). Iterative linear-quadratic regulator design for nonlinear biological movement systems. In *ICRA-1*, volume 1, pages 222–229, Setubal, Portugal.

Lin, T., Horne, B., Tino, P., and Giles, C. (1996). Learning long-term dependencies in NARX recurrent neural networks. *Neural Networks, IEEE Transactions on*, 7(6):1329–1338.

Long, P. and Servedio, R. (2010). Restricted boltzmann machines are hard to approximately evaluate or simulate. In *Proceedings of the 27th International Conference on Machine Learning*, pages 703–710.

Lugosi, G. (2004). Concentration-of-measure inequalities.

Mahoney, M. (2005). Adaptive weighing of context models for lossless data compression. *Florida Inst. Technol., Melbourne, FL, Tech. Rep. CS-2005-16*.

Martens, J. (2010). Deep learning via hessian-free optimization. In *ICML-27*, pages 735–742, Haifa, Israel.

Martens, J. and Sutskever, I. (2010). Parallelizable sampling of markov random fields. In *Artificial Intelligence and Statistics*.

Martens, J. and Sutskever, I. (2011). Learning recurrent neural networks with hessian-free optimization. In *ICML-28*, pages 1033–1040.

Martens, J., Sutskever, I., and Swersky, K. (2012). Estimating the hessian by back-propagating curvature.

Mayer, H., Gomez, F., Wierstra, D., Nagy, I., Knoll, A., and Schmidhuber, J. (2006). A system for robotic heart surgery that learns to tie knots using recurrent neural networks. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 543–548. IEEE.

Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., and Khudanpur, S. (2010). Recurrent Neural Network Based Language Model. In *Proceedings of the Eleventh Annual Conference of the International Speech Communication Association*.

Mikolov, T., Kombrink, S., Burget, L., Cernocky, J., and Khudanpur, S. (2011). Extensions of recurrent neural network language model. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 5528–5531. IEEE.

Mikolov, T., Kopecky, J., Burget, L., Glembek, O., and Cernocky, J. (2009). Neural network based language models for highly inflective languages. In *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, pages 4725–4728. IEEE.

Mikolov, T., Sutskever, I., Deoras, A., Le, H., Kombrink, S., and Cernocky, J. (2012). Subword language modeling with neural networks. *preprint (http://www. fit. vutbr. cz/imikolov/rnnlm/char. pdf)*.

Mnih, A. and Hinton, G. (2009). A scalable hierarchical distributed language model. *Advances in Neural Information Processing Systems*, 21:1081–1088.

Mohamed, A., Dahl, G., and Hinton, G. (2012). Acoustic modeling using deep belief networks. *Audio, Speech, and Language Processing, IEEE Transactions on*, 20(1):14 –22.

More, J. (1978). The levenberg-marquardt algorithm: implementation and theory. *Numerical analysis*, pages 105–116.

Neal, R. (2001). Annealed importance sampling. *Statistics and Computing*, 11(2):125–139.

Neal, R. and Hinton, G. (1998). A view of the EM algorithm that justifies incremental, sparse, and other variants. *NATO ASI Series D Behavioural and Social Sciences*, 89:355–370.

Nesterov, Y. (1983). A method of solving a convex programming problem with convergence rate O(1/sqr(k)). *Soviet Mathematics Doklady*, 27:372–376.

Nocedal, J. and Wright, S. (1999). *Numerical optimization*. Springer verlag.

O'Donoghue, B. and Candes, E. (2012). Adaptive restart for accelerated gradient schemes. *Arxiv preprint arXiv:1204.3982*.

Papineni, K., Roukos, S., Ward, T., and Zhu, W. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics.

Pearlmutter, B. (1994). Fast exact multiplication by the Hessian. *Neural Computation*, 6(1):147–160.

Peterson, C. and Anderson, J. (1987). A mean field theory learning algorithm for neural networks. *Complex Systems*, 1(5):995–1019.

Plaut, D., Nowlan, S., and Hinton, G. E. (1986). Experiments on learning by back propagation. Technical Report CMU-CS-86-126, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Rissanen, J. and Langdon, G. (1979). Arithmetic coding. *IBM Journal of Research and Development*, 23(2):149–162.

Rumelhart, D., Hinton, G., and Williams, R. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533–536.

Salakhutdinov, R. and Hinton, G. (2009). Semantic hashing. *International Journal of Approximate Reasoning*, 50(7):969–978.

Salakhutdinov, R. and Murray, I. (2008). On the quantitative analysis of deep belief networks. In *Proceedings of the 25th international conference on Machine learning*, pages 872–879. ACM.

Sandhaus, E. (2008). The new york times annotated corpus. *Linguistic Data Consortium, Philadelphia*.

Saxe, A., Koh, P., Chen, Z., Bhand, M., Suresh, B., and Ng, A. (2010). On random weights and unsupervised feature learning. In *Workshop: Deep Learning and Unsupervised Feature Learning (NIPS)*.

Schmidt, M. (2005-2009). minfunc. http://www.di.ens.fr/ mschmidt/Software/minFunc.html.

Schraudolph, N. (2002). Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, 14(7):1723–1738.

Scott, S. (2004). Optimal feedback control and the neural basis of volitional motor control. *Nature Reviews Neuroscience*, 5:534–546.

Scott, S., Gribble, P., Graham, K., and Cabel, D. (2001). Dissociation between hand motion and population vectors from neural activity in motor cortex. *Nature*, 413:161–165.

Sergio, L. and Kalaska, J. (2003). Systematic changes in motor cortex cell activity with arm posture during directional isometric force generation. *J. Neurophysiol.*, 89:212–228.

Shewchuk, J. (1994). An introduction to the conjugate gradient method without the agonizing pain.

Shimansky, Y., Kang, T., and He, J. (2004). A novel model of motor learning capable of developing an optimal movement control law online from scratch. *Biological Cybernetics*, 90:133–145.

Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing: Volume 1: Foundations*, pages 194–281. MIT Press, Cambridge.

Snoek, J., Larochelle, H., and Adams, R. (2012). Practical bayesian optimization of machine learning algorithms. *arXiv preprint arXiv:1206.2944*.

Stroeve, S. (1998). An analysis of learning control by backpropagation through time. *Neural Networks*, 11:709–721.

Sutskever, I. (2007). Nonlinear multilayered sequence models. Master's thesis, University of Toronto.

Sutskever, I. (2009). A simpler unified analysis of budget perceptrons. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 985–992. ACM.

Sutskever, I. and Hinton, G. (2007). Learning multilevel distributed representations for high-dimensional sequences. *Proceeding of the Eleventh International Conference on Artificial Intelligence and Statistics*, pages 544–551.

Sutskever, I. and Hinton, G. (2008). Deep, narrow sigmoid belief networks are universal approximators. *Neural Computation*, 20(11):2629–2636.

Sutskever, I. and Hinton, G. (2009a). Temporal-Kernel Recurrent Neural Networks. *Neural Networks*.

Sutskever, I. and Hinton, G. (2009b). Using matrices to model symbolic relationships. *Advances in Neural Information Processing Systems*, 21:1593–1600.

Sutskever, I., Hinton, G., and Taylor, G. (2008). The recurrent temporal restricted boltzmann machine. In *NIPS*, volume 21, page 2008. Citeseer.

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

Sutskever, I. and Nair, V. (2008). Mimicking go experts with convolutional neural networks. *Artificial Neural Networks-ICANN 2008*, pages 101–110.

Sutskever, I., Salakhutdinov, R., and Tenenbaum, J. (2009). Modelling relational data using bayesian clustered tensor factorization. *Advances in Neural Information Processing Systems (NIPS)*.

Sutskever, I. and Tieleman, T. (2010). On the convergence properties of contrastive divergence. In *Proc. Conference on AI and Statistics (AI-Stats)*.

Swersky, K., Tarlow, D., Sutskever, I., Salakhutdinov, R., Zemel, R., and Adams, P. (2012). Cardinality restricted boltzmann machines. *Advances in Neural Information Processing Systems (NIPS)*.

Tang, Y. and Sutskever, I. (2011). Data normalization in the learning of restricted boltzmann machines.

Taylor, G. and Hinton, G. (2009). Factored conditional restricted boltzmann machines for modeling motion style. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1025–1032. ACM.

Taylor, G., Hinton, G., and Roweis, S. (2007). Modeling human motion using binary latent variables. *Advances in Neural Information Processing Systems*, 19:1345–1352.

Tieleman, T. and Hinton, G. (2009). Using fast weights to improve persistent contrastive divergence. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1033–1040. ACM.

Todorov, E. (2004). Optimality principles in sensorimotor control. *Nature Neuroscience*, 7(9):907–915.

Triefenbach, F., Jalalvand, A., Schrauwen, B., and Martens, J. (2010). Phoneme recognition with large hierarchical reservoirs. *Advances in neural information processing systems*, 23:2307–2315.

Valiant, L. (1984). A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142.

Vapnik, V. (2000). *The nature of statistical learning theory*. Springer-Verlag New York Inc.

Vinyals, O. and Povey, D. (2011). Krylov subspace descent for deep learning. *Arxiv preprint arXiv:1111.4259*.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., and Lang, K. (1989). Phoneme recognition using time-delay neural networks. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 37(3):328–339.

Wainwright, M. and Jordan, M. (2003). Graphical models, exponential families, and variational inference. *UC Berkeley, Dept. of Statistics, Technical Report*, 649.

Ward, D., Blackwell, A., and MacKay, D. (2000). Dasher–a data entry interface using continuous gestures and language models. In *Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 129–137. ACM.

Wasserman, L. (2004). *All of statistics: a concise course in statistical inference*. Springer Verlag.

Welling, M., Rosen-Zvi, M., and Hinton, G. (2005). Exponential family harmoniums with an application to information retrieval. *Advances in Neural Information Processing Systems*, 17:1481–1488.

Werbos, P. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

Wierstra, D. and Schmidhuber, J. (2007). Policy gradient critics. *Machine Learning: ECML 2007*, pages 466–477.

Williams, R. and Peng, J. (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 2(4):490–501.

Williams, R. and Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280.

Wood, F., Archambeau, C., Gasthaus, J., James, L., and Teh, Y. (2009). A stochastic memoizer for sequence data. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1129–1136. ACM.