SECOND-ORDER OPTIMIZATION FOR NEURAL NETWORKS

by

James Martens

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

Abstract

Second-order Optimization for Neural Networks

James Martens
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2016

Neural networks are an important class of highly flexible and powerful models inspired by the structure of the brain. They consist of a sequence of interconnected layers, each comprised of basic computational units similar to the gates of a classical circuit. And like circuits, they have the capacity to perform simple computational procedures such as those which might underlie the generating process of the dataset they are trained on. The most popular and successful approach for learning neural networks is to optimize their parameters with respect to some objective function using standard methods for nonlinear optimization. Because basic methods like stochastic gradient descent (SGD) can often be very slow for deeply layered neural networks, or ones with recurrent connections, it is worthwhile to consider more advanced methods. In this thesis we review and analyze various such methods that have been proposed over the past few decades, with a particular focus on approximate-Newton/2nd-order ones, and develop two of our own which we call Hessian-free optimization (HF) and Kronecker-factored Approximate Curvature (K-FAC) respectively. Our experiments show that K-FAC can be much faster in practice at optimizing deep neural networks than well-tuned SGD with momentum.

Acknowledgements

I am grateful to my advisers Rich Zemel and Geoff Hinton for their constant support and guidance over the years, and for giving me the freedom to pursue my own interests. Their brilliance and creativity has been an ongoing source of inspiration.

I've been fortunate to be part of an excellent department filled with great people. I would like to thank my supervisory committee members Ken Jackson and Radford Neal for their insightful commentary and thoroughness. I've also been fortunate to interact with other great faculty such as Ruslan Salakhutdinov, Toni Pitassi and Raquel Urtasun. And of course I must thank Relu and Luna for all of their hard work keeping the ML lab running.

This thesis would have been much weaker without the help of my close collaborators Ilya Sutskever and Roger Grosse, to whom I'm truly indebted. And I would like to thank my thesis external examiner Jorge Nocedal for his wisdom and enthusiastic engagement with the ML community.

I would also like to acknowledge the great DCS students, postdocs and visitors who have made my time at Toronto a much more enjoyable and interesting one. These include Jake, Ryan K., Yujia, Charlie, George, Abdelrahman, Renjie, Eleni, Maks, Joanna, Kevin, Russ, Iain, Hugo, Emily, Amit, James B., Ben, Arkadev, Nitish, Alex G, Elman, Yura, Roland, Tony, Martin, Alex S., Vinod, Abe, Ilya, Tim, Tyler, Danny, Josh, Azin, Nikola, Marc'Aurelio, Ivan, Laurent, David, Chris, Michael, Erin, Jimmy, Roger, Jasper, Andriy, Tanya, Vlad, Tijmen, Alex K., Graham, Laurens, Venkatesh, Tapani, Ryan A., Navdeep, Neill, Tom, Marina, and many other people that I'm surely forgetting.

I would like to thank my friends and colleagues at Massey for enriching my grad school experience, and all my other friends and family in Toronto for their encouragement.

Finally, I would like to thank my mother for raising me and providing a great role-model, and for her ongoing love and support.

Contents

Relationship to published work

This thesis was in large part directly derived from previously published or presented works. The table below lists these works for each corresponding chapter.

Chapter 3
- Depth oral report previously presented to committee

Chapter 4
- Deep learning via Hessian-free optimization.
  J. Martens. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, 2010.
- Learning recurrent neural networks with Hessian-free optimization.
  J. Martens and I. Sutskever. In *Proceedings of the 28th International Conference on Machine Learning (ICML)*, pages 1033–1040, 2011.
- Training deep and recurrent networks with Hessian-free optimization.
  J. Martens and I. Sutskever. In *Neural Networks: Tricks of the Trade*, pages 479–535. Springer, 2012.

Chapter 5
- Depth oral report previously presented to committee
- New insights and perspectives on the natural gradient method.
  J. Martens. 2014, `arXiv:1412.1193`.

Chapter 6
- Optimizing neural networks with Kronecker-factored approximate curvature.
  J. Martens and R. Grosse. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 2015.

Table of notation

Notation	Description
$[v]_i$	i-th entry of a vector v
$[A]_{i,j}$	(i,j)-th entry a matrix A
$\mathrm{diag}(A)$	vector consisting of the diagonal of the matrix A
$\mathrm{diag}(v)$	diagonal matrix A satisfying $[A]_{i,i} = [v]_i$
$\mathbf{1}_m$	vector of length m whose entries are 1
$\mathrm{sq}(\cdot)$	element-wise square of a vector or a matrix
γ'	element-wise first derivative of a function γ
γ''	element-wise second derivative of a function γ
$\nabla\gamma$	gradient of a scalar function γ
J_γ	Jacobian of a vector-valued function γ
H_γ	Hessian of a scalar function γ (typically taken with respect to θ unless otherwise specified)
θ	vector of parameters
W_i	weight matrix at layer i
s_i	unit inputs at layer i
a_i	unit activities at layer i
$\bar{a}_i$	unit activities at layer i with a homogeneous coordinate of value 1 appended
ℓ	number of layers
m	dimension of the network's output $f(x,\theta)$
$W_{i,j}^{(\Delta t)}$	matrix of weights from layer j to i across a time difference Δt (for RNNs)
T	number of timesteps (for RNNS)
$f(x,\theta)$	function mapping the neural network's inputs to its output
$L(y,z)$	loss function
h	objective function
S	training set
k	current iteration
n	dimension of θ
δ_k	candidate update produced at iteration k
M_k	local quadratic approximation of f at θ_k
B_k	The curvature matrix associated with M_k
D	damping matrix
P	preconditioning matrix
$K_i(A, r_0)$	subspace span$\{r_0, Ar_0, \ldots, A^{i-1}r_0\}$
λ	strength constant for penalty-based damping
$\lambda_j(A)$	j-th largest eigenvalue a symmetric matrix A
$x \odot y$	element-wise (Hadamard) product between x and y
$\mathcal{D}v$	$\dfrac{\mathrm{d}L(y, f(x,\theta))}{\mathrm{d}v}$
G	generalized Gauss-Newton matrix (GGN)
$P_{x,y}(\theta)$	model's distribution
$Q_{x,y}(\theta)$	data distribution
$\hat{Q}_{x,y}(\theta)$	training/empirical distribution
$R_{y\|z}$	predictive distribution used at network's output (so $P_{y\|x}(\theta) = R_{y\|f(x,\theta)}$)
p, q, r	density functions associated with above P, Q, and R (resp.)
F	Fisher information matrix (associated with $P_{x,y}$ typically)

Table 1: A table summarizing some of the notation used throughout this thesis. Note we will occasionally use some of these symbols to describe certain concepts that are local to a given section, in which case the notation is effectively "overridden". The subscript "k" will occasionally be dropped for compactness where it is implied by the context.

Chapter 1

Introduction

Neural networks are powerful and highly flexible conditional probabilistic models (or functions approximators) inspired by the structure and low-level functionality of the brain. Neural networks typically consist of a series of layers of units/neurons, where each unit acts as a primitive information processor, similar to a logic gate in an electronic circuit. Units in adjacent layers are connected via trainable weighted connections, and as in a circuit, information flows forward across the layers (or in time), allowing the network to implement serial computations.

One of the main technical challenges faced when using neural networks is that of optimizing these weighted connections according to some measure of quality of the network's output. While finding the precise optimum is formally intractable in the worst case, in practice neural networks can be trained quite effectively on real problems using standard local optimization methods. Beyond some basic intuitions, the reasons for this are not well understood.

As neural networks become deeper, and recurrent connections are added, optimization using stochastic gradient descent (SGD) proceeds much more slowly, which motivates the consideration of more advanced local optimization methods such as approximate-Newton/2nd-order methods. In this thesis we review and analyze some of these methods, and develop several of our own.

Chapter 2 gives some basic background on neural networks and establishes the notation we will use through the rest of this thesis.

Chapter 3 provides a detailed discussion of some of the difficulties faced when optimizing deep/recurrent networks, and provides some reasons why 2nd-order methods may help. A representative selection of such methods proposed over the past two decades are then reviewed in detail.

Chapter 4 provides a detailed description of the "Hessian-free" (HF) method, which is a type of truncated Newton method that has been specifically designed with neural networks in mind. Various concepts that are important to HF, but are also highly relevant to 2nd-order optimization in general, such as damping, preconditioning, the Gauss-Newton matrix, and the use of mini-batches, are thoroughly examined. Experimental results are presented demonstrating the effectiveness of HF on difficult neural network optimization problems.

Chapter 5 reviews and discusses some theoretical aspects of Amari's natural gradient method, provides a unifying picture of the many different versions of it which have appeared over the years, and offers some new insights and perspectives regarding the method and its relationship to other optimization methods. A general condition is identified under which the Fisher information matrix and Schraudolph's generalized Gauss-Newton matrix are equivalent, a critical analysis of the parameterization invariance properties of natural gradient descent is presented, along with a novel analysis of the asymptotic convergence speed of general stochastic 2nd-order

methods.

Finally, Chapter 6 develops an efficient method for approximating natural gradient descent in neural networks called Kronecker-factored Approximate Curvature (K-FAC), which is derived by approximating various large blocks of the Fisher information matrix (corresponding to entire layers) as being the Kronecker product of two much smaller matrices. An efficient and robust optimization method is developed from this approximation, by exploiting the links explored in Chapter 5 between natural gradient descent and more classical 2nd-order optimization methods, thus enabling the application of the techniques discussed in Chapter 4. Experiments are presented that show K-FAC to be much more efficient in practice at optimizing deep neural networks than even carefully tuned SGD with momentum.

Chapter 2

Notation and basic background

2.1 Feed-forward neural networks

In this section we will define the basic notation for feed-forward neural networks (FNNs) which we will use throughout this thesis.

A feed-forward neural network transforms its input $a_0 = x$ to an output $f(x, \theta) = a_\ell$ through a series of ℓ layers, each of which consists of a bank of units/neurons. The units each receive as input a weighted sum of the outputs of units from the previous layer and compute their output via a nonlinear "activation" function. We denote by s_i the vector of these weighted sums for the i-th layer, and by a_i the vector of unit outputs (aka "activities"). The precise computation performed at each layer $i \in \{1, \ldots, \ell\}$ is given as follows:

$$s_i = W_i \bar{a}_{i-1}$$
$$a_i = \phi_i(s_i)$$

where ϕ_i is an element-wise nonlinear function, W_i is a weight matrix, and $\bar{a}_i$ is defined as the vector formed by appending to a_i an additional homogeneous coordinate with value 1. Note that we do not include explicit bias parameters here as these are captured implicitly through our use of homogeneous coordinates. In particular, the last column of each weight matrix W_i corresponds to what is usually thought of as the "bias vector". Figure 2.1 illustrates our definition for $\ell = 2$.

We will define $\theta = [\mathrm{vec}(W_1)^\top \mathrm{vec}(W_2)^\top \ldots \mathrm{vec}(W_\ell)^\top]^\top$, which is the vector consisting of all of the network's parameters concatenated together, where vec is the operator which vectorizes matrices by stacking their columns together.

2.2 Recurrent neural networks

Recurrent neural networks (RNNs) are a temporal generalization of classical feed-forward neural networks, which map input *sequences* to output *sequences*, and have recurrent connections on the hidden layers which allow information to be transferred and processed over time. Formally, they map an input sequence $x = a_0 =$

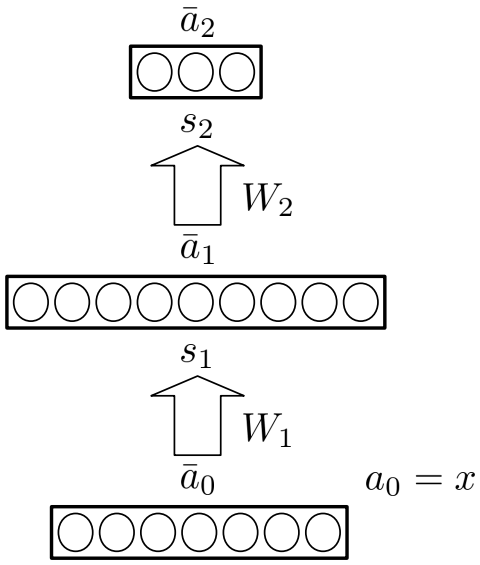

Figure 2.1: A depiction of a standard feed-forward neural network for $\ell = 2$.

$(a_0^{(1)}, a_0^{(2)}, \ldots, a_0^{(T)})$ to an output sequence $f(x, \theta) = a_\ell = (a_\ell^{(1)}, a_\ell^{(2)}, \ldots, a_\ell^{(T)})$ according to the recursions

$$s_i^{(t)} = W_{i,i}^{(1)} \bar{a}_i^{(t-1)} + W_{i,i-1}^{(0)} \bar{a}_{i-1}^{(t)}$$
$$a_i^{(t)} = \phi_i(s_i^{(t)})$$

where $W_{i,i}^{(1)}$ is a matrix of the recurrent weights at layer i, and $W_{i,i-1}^{(0)}$ is a matrix of weights connecting layer $i-1$ to i (the superscript indicates the difference in time index between the corresponding source and destination units), and the remaining quantities are equivalent to their analogs in the feed-forward case, except that they are replicated across T timesteps, where the superscript indicates the current time-step. The parameters θ of an RNN are defined as the concatenation of these weights matrices into a single vector, i.e.

$$\theta \equiv [\mathrm{vec}(W_{1,0}^{(0)})^\top, \mathrm{vec}(W_{1,1}^{(1)})^\top, \mathrm{vec}(W_{2,1}^{(0)})^\top, \mathrm{vec}(W_{2,2}^{(1)})^\top, \ldots, \mathrm{vec}(W_{\ell,\ell-1}^{(0)})^\top, \mathrm{vec}(W_{\ell,\ell}^{(1)})^\top]^\top$$

Note that we are using a generalized multi-layer definition of RNNs here. The classical "shallow" definition can be obtained by choosing $\ell = 2$, and $W_{\ell,\ell} = 0$ (i.e. removing the recurrent connections at the output layer).

Typically, the initial values of hidden units, i.e. $a_0^{(i)}$ for $1 \leq i \leq \ell$, are considered parameters and are learned along with the weight matrices. However, we will assume they are 0 in this work for the purpose of simplicity.

2.3 Training objective

We let $L(y, z)$ denote the loss function which measures the disagreement between a prediction z made by the network, and a target y.

For RNNs, the loss function $L(y, z)$ is typically defined as the sum of time-step specific losses $L_t(y^{(t)}, z^{(t)})$

which measure the degree to which each $f^{(t)}(x,\theta) = a_\ell^{(t)}$ fails to predict $y^{(t)}$. It is given by

$$L(y,z) \equiv \sum_{t=1}^{T} L_t(y^{(t)}, z^{(t)})$$

In the context of RNNs, we define $y = (y^{(1)}, y^{(2)}, \ldots, y^{(T)})$, so that the single case loss can be written more simply as $L(y, f(x,\theta))$.

In general, the training objective function $h(\theta)$ is the average (or expectation) of losses $L(y, f(x,\theta))$ with respect to a training distribution $\hat{Q}_{x,y}$ over input-target pairs (x, y). $h(\theta)$ is a proxy for the objective which we actually care about but don't have access to, which is the expectation of the loss taken with respect to the true data distribution $Q_{x,y}$.

Typically, $\hat{Q}_{x,y}$ is given as a finite set S of samples from $Q_{x,y}$, so that the objective function can be written as

$$h(\theta) = \mathrm{E}_{\hat{Q}_{x,y}}[L(y, f(x,\theta))] = \frac{1}{|S|} \sum_{(x,y)\in S} L(y, f(x,\theta))$$

In the sections which follow, we will use the notation h to denote either the loss as averaged over the entire training set S, or over a subset of S called a "mini-batch", or sometimes over just a single training case (x, y). Quantities related to h such as the gradient ∇h and the Hessian, will share a similar abuse of notation, with the correct interpretation hopefully being clear from the context. In some cases, any of these interpretations will be valid.

For convenience we will define the following additional notation:

$$\mathcal{D}v = \frac{\mathrm{d}L(y, f(x,\theta))}{\mathrm{d}v}$$

2.4 Neural network learning basics

The main technical tool underlying most of the methods discussed in this thesis is the gradient of h, denoted ∇h and the algorithm used to compute it, which is backwards-mode automatic differentiation. Backwards-mode automatic differentiation is an algorithm for efficiently computing the gradient of a scalar function in time comparable to evaluating that function, and it operates by working backwards through the computation graph and computing derivatives of h w.r.t. each node. The value of the derivative is, by the chain rule, a relatively simple function of the derivatives at the child nodes (i.e. the nodes that take the current one as input).

Neural networks are very naturally expressed as computational graphs, since neurons can map onto nodes, and the weight multiplications can either be assigned nodes as well, or treated as a part of the computation occurring at each node. Backwards-mode automatic differentiation, like the "forward" evaluation of h (via f), involves transmitting information only between connected nodes/neurons, and thus has an elegant interpretation as a purely "local" mechanism for computing a learning signal by propagation of derivative information "backwards" through the network. In the work of Rumelhart et al. (1986), this insight was used to derive a neurologically plausible learning rule that amounted to (stochastic) gradient descent with derivatives computed using backwards-mode automatic differentiation. The suggestive name Back Propagation (BP) was given to the resulting optimization algorithm, although more recently this term is often used to describe only the part of the algorithm which computes the gradient via backwards-mode automatic differentiation, as there are now many alternative ways to use the gradient within a learning algorithm.

Algorithm 1 An algorithm for computing the gradient of the loss $L(y, f(x, \theta))$ for a given (x, y). Note that we are assuming here for simplicity that the ϕ_i are defined as coordinate-wise functions.

input: $a_0 = x$; θ mapped to $(W_1, W_2, \ldots, W_\ell)$.

/* Forward pass */
for all i **from** 1 **to** ℓ **do**
  $s_i \leftarrow W_i \bar{a}_{i-1}$
  $a_i \leftarrow \phi_i(s_i)$
end for

/* Loss derivative computation */
$$\mathcal{D}a_\ell \leftarrow \left.\frac{\partial L(y, z)}{\partial z}\right|_{z=a_\ell}$$

/* Backwards pass */
for all i **from** ℓ **downto** 1 **do**
  $g_i \leftarrow \mathcal{D}a_i \odot \phi_i'(s_i)$
  $\mathcal{D}W_i \leftarrow g_i \bar{a}_{i-1}^\top$
  $\mathcal{D}a_{i-1} \leftarrow W_i^\top g_i$
end for

output: $\mathcal{D}\theta = [\mathrm{vec}(\mathcal{D}W_1)^\top \mathrm{vec}(\mathcal{D}W_2)^\top \ldots \mathrm{vec}(\mathcal{D}W_\ell)^\top]^\top$

Algorithm 1 shows how to compute the gradient $\mathcal{D}\theta$ of the loss function of a feed-forward neural network using standard backpropagation. ∇h is obtained by averaging $\mathcal{D}\theta$ over each $(x, y) \in S$.

Chapter 3

Optimization Difficulties and the role of 2nd-order Optimization

3.1 Sources of difficulty

3.1.1 Local optimization issues

The activation functions in neural networks are usually chosen to be smooth functions, which facilitates optimization since this implies that the network function $f(x, \theta)$ will be smooth, and hence h will be as well (assuming the loss L is), thus allowing general purpose gradient-based optimization methods to be applied, and their associated convergence theorems to be valid. However, despite being "smooth" in a strictly mathematical sense, the optimization surface for h may contain many sharp turns, deep and narrow valleys, huge variations in curvature along different directions which will be reflected in badly conditioned Hessian matrices, and other features for which mathematical non-smoothness can be seen as a limiting case.

RNNs may be particularly problematic in this regard, since the non-linear dynamical system represented by the hidden state sequence, and controlled by the hidden weight matrix, may be nearly insensitive to changes in some parameters, and hypersensitive to others, due to the way that the weight matrix repeatedly multiplies the output of each temporal "layer" to determine the input of the next, and thus creates an extremely high-order dependency. In fact, if we take the limit as T, the length of a training time-series, goes to infinity, the objective function for RNN training tends to a non-smooth function, although it will still be strictly speaking smooth for any finite T. As a simple analogy, consider the case where the RNN has 1 dimensional hidden state with a linear transition function given by multiplication by some scalar a. In the input-free case, the hidden state after T steps is just a^T, a quantity which is hyper sensitive to the value of a around $a = 1$ and becomes a non-smooth function of a (discontinuous in fact) at that point, as T goes to infinity.

Jaeger and Haas (2004) have studied this kind of behavior of RNNs and observed that in the hypothetical case where the units compute the identity function, so that the RNN become a purely linear system, its stability is determined the magnitudes of the eigenvalues of the weight matrix, and in particular as they exceed 1 the state will blow up or "explode", while if they drop below 1 it will decay to zero (at least in the absence of new input signals or if the input is 0).

But even if f itself doesn't explode or decay for any setting of the parameters due to the "squashing" sigmoidal non-linearities keeping the total magnitude of the activations in each hidden layer bounded, h may still be very

badly behaved for certain θ's, as evidenced by the value of the gradient of f (on which the gradient of the objective h depends). In particular, local derivatives of the components of f w.r.t. intermediate unit activations, which are computed by taking a local "linearization" of the network, may explode or decay as the states themselves do in a purely linear network, due to how the gradient is computed by repeated multiplication of a vector of errors by the transposed hidden dynamics matrix $W_{1,1}^{(1)\top}$. Several papers (e.g. Hochreiter, 1991; Bengio et al., 1994; Hochreiter et al., 2000) described this phenomenon, and cited it as one of the main reasons for the failure of standard optimization approaches to learn good solutions to sequence prediction datasets/tasks that contain long-range dependencies (i.e. where inputs relevant for predicting a given output may be separated by many time-steps).

Gradients that have exploded or decayed indicate an extreme local sensitivity or insensitivity of the hidden state a_t of the network at some time-step t to the hidden state at earlier points in time, and are problematic from the standpoint of optimization. In particular, when the gradient explodes, this indicates that following it (over a short distance) will result in changes to the output of the network (given by f) that will be almost entirely a result of changes in the activations of earlier units propagating forward over many time-steps to much later ones. Conversely, when it decays, this indicates that following the gradient will lead to changes in the output resulting almost entirely from changes in hidden unit activations propagating forward over only a few time-steps. Neither of these situations is desirable, as efficient and effective optimization of the output at some given time-step will likely require that both "short-term behavior" (defined as the *recently occurring* beneficial transformations in the hidden state and processing of new input data) and the "long-term behavior" (defined analogously) are optimized together.

To demonstrate this, we will consider the case of decay, where the error signals back-propagated over short temporal distances dominate the gradient, and suppose that we have an optimizer which makes updates based on the unmodified gradient, so that the only significant changes that get made to θ are those predicted to have a beneficial effect on the short-term behavior. Long-term behavior can and will also change, but not in a way predicted to decrease the loss. Suppose also that our dataset is such that the correct prediction of various outputs depends strongly on inputs which occur much earlier in the input sequence (such as in the RNN learning tasks proposed in Hochreiter and Schmidhuber (1997)), so that the RNN must store information about inputs and propagate it forward through the hidden state over many time-steps. Clearly, optimization of long-term behavior is critical to a successful solution of this learning problem, but will unfortunately not occur, given the nature of our hypothetical optimizer and the decay of the long-term error signals.

One might hope that eventually the optimization would reach a point where continued refinement of short-range behaviors would provide no additional (significant) reduction in h, and so the short-range error signals would become small, thus allowing the long-range ones to achieve relative significance. But if we are optimizing with a stochastic gradient, as is often done, the optimal short-term behavior will almost always be a compromise made over the whole dataset, meaning that the contribution to the gradient from short-range errors will only become small when averaged over the whole dataset (through cancellation), and will still be large for any particular training case, or a sufficiently small subset of them.

And even if we assume a deterministic gradient, there are several things that could still go wrong:

- Firstly, the decay may be so extreme and the resulting gradient contribution from the long-term errors so small, that the level of convergence in the short-term behavior required may be too strict to be practical (or it might even exceed machine precision).

- Secondly, as the network learns to use long-term information and transform it appropriately, the shorter-term behavior of the network may constantly require readjustment. For example, because they are determined by

the same set of parameters [1], or because they require coordinated interaction to properly perform the type of underlying computation needed to achieve low error on the dataset[2].

- Thirdly, if many iterations are spent only optimizing short-term behavior of the network without any regard for what it does to the long-term behavior, this may eventually place the parameters in a position from which no locally downhill updates can be made which could allow for the exploitation of long-range information. For example, the decay problem may become so extreme that the long-term errors become effectively zero. This issue is directly related to the non-convexity of the neural network training objective and global optimization, which will be discussed in the next sub-section.

- Fourthly, due to the sharing of resources, optimizing the long-range behavior of the network after the short-term behavior has been optimized, even if the previously non-convexity issue doesn't make this strictly impossible, may nonetheless turn out to be a very "rigid" optimization problem, wherein it becomes extremely difficult to make large local updates to the parameters without causing disruptive changes in the already heavily optimized (and perhaps nearly optimal) short-term behavior of the network.

While the above discussion has focused on RNNs, similar local optimization issues exist for feed-forward networks, and much of the above discussion caries over, if we replace the notion of 'time-step' with the analogous one of 'layer'. Fortunately, they tend to be less extreme, likely because the number of layers in a feed-forward network tends to be much smaller than the number of time-steps that separate relevant inputs and outputs in an RNN. But for very deep neural networks, such as deep auto-encoders (Hinton and Salakhutdinov, 2006), optimization may still be very difficult (e.g. Hinton and Salakhutdinov, 2006; Bengio et al., 1994) for basic methods like generic stochastic gradient descent.

The exploding/vanishing gradient phenomenon will have slightly different implications in the case of feed-forward networks, due to the fact that distinct parameters determine input weights to each layer, and the way that inputs/outputs only enter/leave the network at the first/last layer (versus the RNN where inputs and outputs exist at every time-step). In particular, the exploding/vanishing of the error signal may lead to differences in the scale of the gradient and curvature localized to particular layers, and this can in practice lead to situations where entire layers of the network are hardly optimized at all. The resulting networks may in some cases completely ignore their inputs and simply predict outputs based on the mean of the target output y's. This problem is exacerbated by the use of a L2 regularization term of the form $\lambda||\theta||^2$ being added to h, as is common in practice, since this may cause the optimization to send all of the parameters associated with the early layers to zero in the case of gradient decay, leaving the network in an extremely poor local minimum of h. On the other hand, methods which are based on the idea of transforming the gradient by multiplication with a diagonal matrix (e.g. an approximate Newton method with a diagonal approximation to the inverse curvature), re-scaling the optimization in a strictly per-parameter way, will have a better chance of correcting these problems, as we will discuss further in Section 3.3.2.

Some methods which may be able to overcome the local optimization issues discussed in this section include:

- approximate Newton methods (e.g. various diagonal approximations found in the machine-learning literature, and methods based on non-diagonal approximations from the optimization literature such as L-BFGS (Liu and Nocedal, 1989), Krylov-subspace methods),

[1] Which is true of RNNs but not feed-forward networks.

[2] This will always be true to some extent since in RNNs, any long-term information must pass through all of the recent hidden states

- per-parameter learning rate adjustment methods which can also be interpreted as approximate Newton methods, alternative optimization procedures based on dynamic reparameterization (Raiko et al., 2012) and heuristic gradient modification (Schraudolph, 1998; Bengio et al., 1994),
- alternative architectures and models modifications such as Echo-state networks (Jaeger and Haas, 2004) and Long Short-term Memory (Hochreiter and Schmidhuber, 1997) which are explicitly designed to be easier to optimize,
- methods based around dynamic reparameterization of the network such as the "centering" methods of Schraudolph (1998) and Raiko et al. (2012),
- and finally there are initialization methods (e.g. Nguyen-Widrow, layer-wise pre-training (Hinton and Salakhutdinov, 2006)) which can help local optimization by placing the initial parameters somewhere that various local optimization issues (such as vanishing gradients) are not an immediate problem.

3.1.2 Global optimization issues

Owing to the complex way in which θ parameterizes f, the objective h is highly non-convex, and even if we can efficiently find locally optimal solutions, they will not in general be globally optimal ones. This issue of non-convexity is a problem that relatively few methods even attempt to deal with, and among the ones that do, none provide solutions that are completely satisfying. But this shouldn't be surprising, since we know from classical neural network theory that finding the *global* optimum, even for relatively simple feed-forward networks with a single hidden layer, is an NP-hard problem in general[3] (Blum and Rivest, 1988).

In addition to making it impossible to guarantee that a given locally optimal solution will yield comparable performance to the globally optimal one, the negative consequences of non-convexity can be observed even before convergence takes place. In particular, standard optimization approaches which don't use randomization can easily get permanently stuck in large areas of the parameter space that are clearly seen to be sub-optimal. For example, if a symmetry develops in the parameters where the incoming and outgoing weights associated with two units in the same layer are identical, it is easy to see that no deterministic gradient-based method (which includes all of the methods considered in this thesis) can possibly break the symmetry. Similarly, if the incoming and outgoing weights to a unit are all zero, such methods will also fail to ever rectify this, and the unit will remain effectively "dead".

One way to potentially avoid issues of global optimization is to adopt MCMC method such as Hybrid/Hamiltonian Monte Carlo (Duane et al., 1987; Neal, 1996) which, roughly speaking, can be thought of as optimization methods that add noise to the iterates in a principled way. Another way may to adopt intelligent initialization schemes such as Nguyen-Widrow or layer-wise pre-training (Hinton and Salakhutdinov, 2006).

3.1.3 Interaction of global and local optimization

It should be noted that issues of global and local optimization cannot be cleanly separated, as the problems associated with local optimization depend very much on the local geometry of the error surface, which, like the quality of the local minima, will vary across parameter space. Indeed, when parameters of an RNN are in a state associated with vanishing gradients, for reasons discussed in Section 3.1.1, the network may never learn to take advantage of long-range dependencies (which represents a very poor local minima of h).

[3] This holds true even for neural networks with only 2 units with linear threshold activations in its (single) hidden layer.

Obviously, the manner and extent to which this happens depends on the local optimizer and the type of trajectories it tends to take. By correcting for differences in the scale of contributions made by short and long-term signals due to vanishing gradients, it is conceivable that global optimization will be improved insofar as the network's long-range behavior will be optimized to actually exploit long-range dependencies in the data before continued optimization of short-term behaviors makes this impossible. This was the hypothesis put forth in (Martens and Sutskever, 2011), where fairly generic (but nonetheless carefully calibrated) initialization methods were used in conjunction with a powerful approximate Newton method to train RNNs to make use of very long-range temporal dependencies in various specially designed toy datasets from Hochreiter and Schmidhuber (1997). Such problems were well-documented to be extremely difficult or impossible using standard gradient descent. More recently, Sutskever, Martens, Dahl, and Hinton (2013) has produced similar results by using a carefully tuned version of momentum inspired by ideas from research on accelerated gradient techniques (Nesterov, 1983a). Both of these results also rely on the use of well-designed initializations designed to ensure that the networks do not start with extreme vanishing or exploding gradients.

Note that better local optimization can in principle only do so much to resolve the problems with RNNs, since in the extreme case of vanishing gradients, where the back-propagated error signals is zero up to machine precision, the gradient will simply not contain the required information to learn the needed long-term behaviors. And as mentioned previously in Section 3.1.2, there are a variety of situations, such as where units are symmetric or dead, where recovery is provably impossible using deterministic optimization.

In the domain of feed-forward neural networks there is also some evidence to suggest a connection between ease of local optimization and quality of nearby local minima. The method of Hinton and Salakhutdinov (2006), which was developed primarily to help combat the underfitting problem, has been shown to yield converged solutions that exhibit lower levels of overfitting (Erhan et al., 2010). Glorot and Bengio (2010) also found that initialization methods which resulted in faster optimization also tended to yield lower values of h close to convergence, which agrees with our experience. Whether there is a substantial difference in quality between the best achievable training set performance among different initialization schemes that are all "reasonably good" (in the sense of avoiding extreme vanishing/exploding gradients and symmetric/dead unit degeneracies, and generally setting the initial scales of each parameter to sensible values) is debatable. Indeed, Martens (2010), and later Sutskever, Martens, Dahl, and Hinton (2013), reported only minor differences in final training performance across different *reasonable* initialization schemes. Again, it is likely that the quality of the initialization may play a diminished role for some optimizers (approximate Newton and momentum methods) than for methods such as standard gradient descent, although there will certainly always be a large difference in performance between reasonably good initializations and very poor ones (such as those compared against in Hinton and Salakhutdinov (2006), for example).

3.2 Depth and underfitting

Model fitting comes down to the selection of a family of models, such as a particular neural network architecture, followed by the selection of a particular member of this family (usually according to some loss function), which in the case of neural networks, is done via optimization of objective function $h(\theta)$ w.r.t. θ.

Underfitting is simply the failure to adequately minimize some penalty function (which may often, but not always, be the loss function used in the training objective). Possible causes of underfitting include the use of an inappropriate or overly restrictive family of models, and the failure of the optimization process to find a good member of a given family.

Since deeper networks are generally considered much harder to optimize, one might wonder if they are worth the trouble. After all, Hornik et al. (1989) showed that neural networks with a single hidden layer of sigmoidal units can approximate any reasonably smooth functional mapping arbitrarily well, if the hidden layer is allowed to be large enough. Thus, the minimized training loss should converge to zero as the size of the hidden layer is allowed to grow. However, if we believe that the types of functional mappings used to generate the training set are best described as sequential computations involving many stages of relatively simple processing operations, deeper networks are more likely to provide much more compact models of these functions than shallow ones can (in the sense of requiring a smaller total number of units and hence free parameters). Indeed, the universality result of Hornik et al. (1989) requires a number of hidden units that will be completely impractical in general (exponential in the dimension n of the input, or worse). Compact models are desirable as they are both cheaper to work with, and have a lower dimensional parameterization which makes them much less prone to overfitting.

In addition to these intuitions, circuit theory tells us that when it comes to layers of generic processing units, the types of computations you can efficiently[4] perform with deeper circuits form a much "richer" family, computationally. To be precise, there are many functions that can be described compactly in mathematical and/or algorithmic terms, and can therefore be compactly computed by relatively small circuits of moderate depth. Such functions are widely believed by computational theorists to be uncomputable by small circuits of much lower depth. In some cases there are even formal results which prove exponential lower bounds on the resources required by shallow circuits to compute certain simple functions, although work in this (very) difficult area of mathematics has been slow. Results of this nature, such as they exist, are available for various kinds of circuits which include "linear threshold circuits", which are basically neural networks with hard-threshold activation functions (for which the standard sigmoids are a smooth analogue).

In the following subsection we briefly review some of these results.

3.2.1 Concrete results separating shallow models from deeper ones

It is known (Hajnal et al., 1993; Forster, 2002) that under various realistic constraints on the precision[5] of their weights, neural networks with a single hidden layer of threshold units (not counting the output layer) cannot efficiently compute the inner-product of two halves of their input (assumed to be in $\{0, 1\}^n$), modulo two, or indeed any function which strongly correlates with this one. Later, it was shown by Martens et al. (2013)[6] that such networks can efficiently simulate neural networks with a single hidden layer of rectified linear neurons (using similarly sized weights), from which a similar hardness result holds for the latter as well. Meanwhile, there are linearly sized neural networks with two or more such hidden layers, of either threshold or linear rectified neurons, that compute this function.

Martens et al. (2013) also showed that under analogous restrictions on their weights, Restricted Boltzmann Machines (RBMs) are incapable of efficiently capturing a certain simple distribution (whose density is essentially the inner product function described above). And by the results of Martens (2016), such a distribution can be efficiently captured by Deep Boltzmann Machines, which are thus more expressively efficient than RBMs.

Pascanu and Bengio (2014) showed that there is a real-valued function which can be efficiently computed by neural networks with two hidden layers of rectified linear units, but not by such networks with only one hidden layer. Notably, this result doesn't require any hypotheses about the precision of the weights. However, unlike in

[4] Where "efficient" usually means requiring linear resources instead of exponential resources

[5] Here "precision" refers to the size of the range of values that the weights can take when constrained to be integer multiples of some constant ϵ.

[6] The proof of Theorem 6 from Martens et al. (2013) can be easily generalized to a version of "RBM networks" where the connections in the second layer are weighted, which are exactly neural networks with a single hidden layer of rectified linear neurons.

the above results, the constructed function's domain is the entire $\mathbb{R}^n$ (as opposed $\{0,1\}^n$), and the proof of its hardness for shallow networks breaks down if the output of the function is "thresholded" to give discrete values.

More recently, Martens and Medabalimi (2014) showed that the set of distributions which can be efficiently captured by a certain kind of deep model called Sum Product Networks (SPNs), grows with each layer of depth permitted. This kind of "depth hierarchy" property has never before been shown to hold for any other well-known deep model, neural networks included, despite the widespread belief that it does hold for most of them.

3.3 Approximate Newton/2nd-order approaches

To optimize an objective function $h(\theta)$, the Newton-Raphson method, or simply Newton's method, from which all "2nd-order methods" derive, works by generating update proposals δ (to be added to the current θ_k to produce θ_{k+1}) as the optimal solutions to sub-problems given local quadratic models of h centered around the current iterate θ_k. Typically, these quadratics are of the form

$$h(\theta_k + \delta) \approx M_k(\delta) = \frac{1}{2}\delta^\top B_k \delta + \nabla h(\theta_k)^\top \delta + h(\theta_k)$$

where B is the curvature matrix (usually the Hessian H of h at θ_k), and $\nabla h(\theta_k)$ is the gradient at θ_k. Note that going forward we will drop the k subscripts when they are clear from the context. Optimization of $M(\delta)$ can be done exactly by solving an $n \times n$ dimensional linear system of the form $B\delta = -\nabla h$ (whose solution is $\delta^* = -B^{-1}\nabla h$ when B is invertible)

Standard Newton's method, where B is given by H and $M(\delta)$ is thus a 2nd-order Taylor series approximation of h, runs into numerous problems when applied to neural network training, owing to the non-convex objective leading to indefinite quadratic optimization sub-problems whose optima are not well defined, and related issues of model trust, where unless certain controls are applied, the method will trust its own local quadratic approximations too much and generate huge/non-nonsensical proposals. Using the Gauss-Newton approximation to the Hessian matrix (which will always be positive definite, even for non-convex functions), and applying various update damping/trust-region techniques, such issues can be addressed and 2nd-order methods can be made practical, at least for low dimensional optimization problems. The remaining issue which prevents their application to neural networks is the generally very high dimensionality of the parameter space which prohibits the calculation, or even the storage, of the n^2-entry Hessian (or Gauss-Newton) matrix (which won't be significantly sparse), never mind the prohibitive cost of fully optimizing the local quadratic model (usually by inversion of B).

To address this issue, various approximate Newton methods have been developed within the optimization and machine learning communities. These methods work by approximating the Hessian matrix with something easier to compute/store such as a low-rank or diagonal matrix and/or approximate the optimization of the resulting quadratic model. Since a comprehensive survey of these methods is impossible (and of limited relevance), this thesis will focus primarily on those which were originally developed within the machine learning community, and with neural network optimization as the target application. But note that the only properties of h which are assumed by these methods are ones of basic smoothness and existence of derivatives, and in some cases that the objective function is of the general form $\frac{1}{|S|}\sum_{(x,y)\in S} L(y, f(x,\theta)))$ for some smooth f. That f is a neural network will not be critical to the correctness or applicability of most of these methods (with the notable exception of the method developed in Chapter 6).

3.3.1 Why should 2nd-order optimization approaches help?

While neural networks haven't been closely studied by optimization researchers, many of the local optimization issues related to neural network learning can be seen as extreme special cases of problems which arise more generally in non-linear optimization. For example, tightly coupled parameters with strong local dependencies, and large variations in scale along different directions in parameter space, are precisely the sorts of issues for which 2nd-order optimization is well suited. Gradient descent on the other hand is well-known to be very sensitive to such issues, and in order to avoid large oscillations and instability must use a learning rate which is inversely proportional to the size of the curvature along the highest curvature direction.

2nd-order optimization methods provide a much more powerful and elegant solution to the problem of variations in scale/curvature along different directions, by selectively re-scaling the gradient along different eigen-directions of the curvature matrix B (which we define as the unit eigenvectors s.t. the sign of the inner product with the gradient ∇h is zero or negative, making each of them descent directions) according to their associated curvature (eigenvalue) instead of employing a one-size-fits-all learning rate. And for reasons discussed in Section 3.1.3 this re-scaling which is designed to accelerate local optimization, may also result in learning trajectories that are more favorable in terms of global optimization.

Note however that it is not entirely clear that re-scaling eigen-directions according to their curvature will necessarily provide a satisfactory solution to the specific local optimization issues which exist with deep networks and RNNs, such as the vanishing/exploding gradients problem, or even help at all. Martens (2010) provided some hypotheses as to why it should help, which were later extended to RNNs, but were rather speculative and lacking in detail. We will now attempt to fill in some of this detail.

Returning to the example discussed in Section 3.1.1 of vanishing gradients in an RNN, we note that as long as this vanishing isn't too severe, the gradient will still contain non-zero contributions from long-term backpropagated errors. In such a situation, it may be the case that there is some unit eigen-direction d (a direction in parameter space that is an eigenvector of B, with $\|d\| = 1$) such that:

1. d has a relatively large negative inner-product with the contribution ∇h_{long} made to ∇h from long-range errors, as compared to its inner product with the contribution ∇h_{short} from shorter range errors, or in other words, $|d^\top \nabla h_{\text{long}}|$ is of larger or comparable size to $|d^\top \nabla h_{\text{short}}|$, despite $\|\nabla h_{\text{short}}\|$ being potentially much larger than $\|\nabla h_{\text{long}}\|$. Intuitively, this says that d is a direction which locally reduces h via reductions in error that result from changes to much earlier hidden states (i.e. changes to the long-term behavior of the RNN)

2. the eigenvalue of d, or in other words, the curvature associated with the direction d according to the quadratic model (which is given by $d^\top B d$), is small.

It can be shown that for certain choices of B, such as the Gauss-Newton, and possibly the Hessian too, the second condition will be implied by vanishing gradients and the first condition, provided that it is *not* the case that $d^\top \nabla h_{\text{long}}$ is small only due to cancellation of many different contributions to ∇h_{long} (at different time-steps and/or different training cases and/or different components) that each have a comparatively much larger inner product with d. In particular, $d^\top B d$ will be small because, given the "no cancellation" assumption, we will have $d^\top B d \approx d^\top B_{\text{long}} d$, where B_{long} is the contribution made to B due purely to long-range errors (a concept which can be made rigorous by analyzing certain expressions for B), and B_{long} will decay just as the gradient does during back-propagation.

Since the Newton update can be understood as decomposing ∇h according to the orthonormal basis of eigenvectors of B and multiplying each by the inverse of the associated eigenvalue/curvature, and because the eigen-

value/curvature associated with d is small (compared to the other eigenvalues), d will be effectively "boosted" in the Newton update versus compared to the standard gradient descent update.

For feed-forward networks the situation is similar and perhaps more favorable, due to the way each layer (the analogous concept to time-step) has its own set of weight/bias parameters. In particular, condition 2 will follow automatically from vanishing gradients, condition 1, and a weakened version of the "no cancellation" condition which only requires lack of significantly large cancellation across components. This is because the large contributions to the curvature matrix B made by short-term errors will occupy long rectangular blocks given by the rows and columns of B that correspond to parameters of later layers, while $d^\top Bd$ will be a weighted sum over entries in a square block on the diagonal of B given by the intersection of the rows and columns associated with the parameters at earlier layers. This block will in fact only contain contributions from long-term errors, and these vanish in both the Gauss-Newton and Hessian curvature matrices just as the gradient does during backpropagation.

While 2nd-order optimization can *potentially* increase the effective learning rate along directions which optimize long-term behaviors, this won't necessarily improve optimization performance (locally or globally). For example, due to the highly nonlinear dependence of the state at a given time-step on a much earlier one, the proposed updates may be very unreliable along such directions, and so following them over the prescribed distance may do nothing to improve the long-term behavior of the network. Another possibility is that condition 1 fails and the eigen-directions of B do not break down nicely in terms of those mostly responsible for modulating short-term behavior versus those responsible for long-term behavior. When this is the case, it could easily be true that following each eigen-direction along a positive distance[7] will result in an improvement to short-term prediction error at the cost of raising long-term errors. In such a scenario it may still be possible, in principle, to make local updates which result in beneficial changes to long-term behavior (e.g. by constructing δ^* so that a certain few d's are pursued in their ascent directions, while the overall δ^* still remains a descent direction), despite the Newton update's failure to do this.

Despite these possible objections, Martens and Sutskever (2011) found that with certain natural and highly non-diagonal choices of B (the generalized Gauss-Newton matrix with certain update-damping terms), approximate Newton updates seemed to have superior local and global convergence properties over plain gradient updates when applied to various RNN learning tasks (from Hochreiter and Schmidhuber (1997)) involving very long-term input-output dependencies in the data (see Section 4.13).

3.3.2 Diagonal B's

Many of the approximate Newton approaches which will be discussed in this section involve the use of a diagonal curvature matrix B. Such choices are understandably popular since diagonal matrices can be stored and inverted in $\mathcal{O}(n)$ space and time.

However, there is a significant disadvantage to using such diagonal methods when the true local curvature (as given by the Hessian) is highly non-diagonal. This situation arises when, for example, there are two parameters (components of θ) which are locally highly coupled, so that making large changes in either one of the them independently will produce poor updates that increase h, but changing them together in a coordinated way will result in a significant reduction in h. Graphically this corresponds to the situation of a narrow ravine in the error surface that is not aligned with any of the coordinate axes. A diagonal method, because it does not model these interactions, must be conservative and restrict the change in each parameter according to the curvature associated

[7] Recall that we defined eigen-directions as satisfying $\nabla h^\top d < 0$ so that each of them is a descent direction and so the fully-optimized Newton update $\delta^* = -B^{-1}\nabla h$ will always do this, as long as B is PSD.

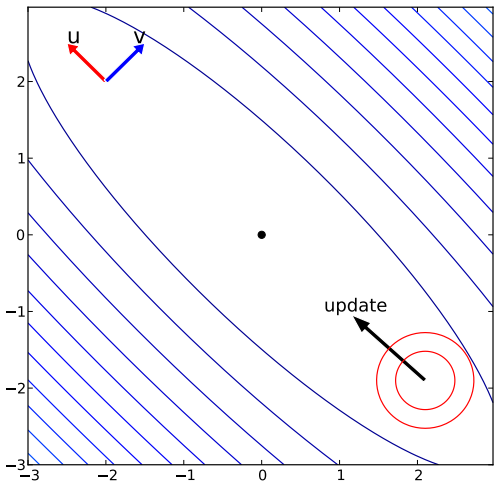

Figure 3.1: Non-diagonal curvature example with $B = uu^\top + avv^\top$ with $u = [-1, 1]^\top$ and $v = [1, 1]^\top$, where a is large (e.g. 10^4, although we take $a = 15$ for display purposes). This matrix is full rank, and its diagonal entries are given by $[a + 1, a + 1]^\top$, representing the fact that the quadratic is highly sensitive to independent changes to the 2 parameters. The small circular region is where the update will be effectively restricted when using a diagonal approximation of the curvature.

with changing only that parameter. This situation is illustrated in Figure 3.1.

It has been observed by Schraudolph et al. (2007) that even sophisticated diagonal methods like Stochastic Meta-Descent (see Subsection 3.4.3) do not perform significantly better than plain SGD when applied to certain ill-conditioned quadratic objectives (where the variations in curvature are very far from being axis-aligned).

It is natural to ask whether the local scaling issues which complicate the optimization of neural networks are "axis-aligned" or not, and whether they can be mostly resolved by modeling the curvature as diagonal, or if non-diagonal methods will be much more effective.

As mentioned in Subsection 3.3.1, deep feed-forward neural networks are likely to exhibit significant axis-aligned scaling issues due to vanishing/exploding gradients and thus diagonal methods are likely to be very beneficial. And indeed, Martens (2010) found that the careful use of diagonal preconditioning (using the diagonal of the empirical Fisher information matrix) improved optimization efficiency for very deep neural networks significantly (see Section 4.9 for a detailed discussion of preconditioning). This finding was reinforced by the work of Chapelle and Erhan (2011), and was consistent with previous findings that diagonal methods helped (although to a lesser degree) when optimizing shallower feed-forward networks (e.g. Becker and LeCun, 1989).

Nonetheless, Martens (2010) found that a fully non-diagonal 2nd-order approach required several orders of magnitude fewer total updates than what is required by a purely diagonal approach (even with the inclusion of momentum-based acceleration). While these very powerful updates were significantly more expensive to compute, their very existence proves that a large part of the difficulty inherent in the optimization of deep networks is a result of purely local variations in curvature along different directions that *cannot* be well modeled as axis-aligned/diagonal.

3.3.3 Adaptive Learning Rates

Silva and Almeida (1990) proposed an approach for dynamically adjusting individual "learning rates" for each parameter according to how the sign of the corresponding entry of the gradient changes across iterations. In particular, the learning rate is increased when the sign remains the same, and is decreased whenever it changes.

The intuition behind this method is that when the sign of the derivative for some component $[\theta]_j$ doesn't change across several iterations, it suggests that $-\operatorname{sgn}([\nabla h]_j)e_j$ remains a direction of descent over longer distances (where e_j is the vector which is 1 in the j-th spot and 0 elsewhere), and so we can afford to take larger steps in that direction with reduced risk of the objective suddenly curving upwards. Another way of putting this is that the positive curvature in h associated with the direction is determined "empirically" to be low.

Like all methods that adjust per-parameter learning rates, this approach can be viewed as an approximate Newton algorithm that maintains a heuristically adjusted diagonal inverse curvature matrix $P \equiv B^{-1}$.

The particular rule used by Silva and Almeida to adjust P is:

$$[P_{k+1}]_{jj} = \begin{cases} u[P_k]_{jj} & \text{if } [\nabla h(\theta_k)]_j [\nabla h(\theta_{k-1})]_j \geq 0 \\ d[P_k]_{jj} & \text{otherwise} \end{cases} \tag{3.1}$$

where P_k is the value of P to be used at iteration k, and $u > 1$ and $0 < d < 1$ are constants that determine the percentage increase and decrease, respectively.

One potential problem with this approach is that using a stochastic gradient estimate could easily cause rapid switching of the direction of the gradient along each coordinate of θ, even in situations where the standard deterministic gradient would have a relatively stable sign. We could hope that this effect would "average out" across multiple iterations, resulting in a noisy adaptation of P that would nonetheless move in the right direction over the long-term, but there is no guarantee of this. And in fact Almeida et al. (1998) found using a stochastic gradient resulted in the entries of P either growing or shrinking without bound, *except* with the choice $u = 1/d$ so that the multiplicative increases and decreases exactly cancel each other.

In order to explain this observation, and develop a possibly more principled learning rate approach, Almeida et al. (1998) proposed to adjust P by performing a single preconditioned gradient-descent update to P of a particular meta-objective given by:

$$h(\theta_{k+1}) = h\left(\theta_i - P\nabla h(\theta_k)\right)$$

resulting in the update:

$$[P_k]_{jj} = [P_{k-1}]_{jj} + \kappa_k [\nabla h(\theta_{k+1})]_j [\nabla h(\theta_k)]_j \tag{3.2}$$

where the κ_k are learning rates/preconditioning constants, and θ_{k+1} is treated as independent of P (c.f. the SMD method described in Section 3.4.3). If we independently draw the training points used to estimate $\nabla h(\theta_{k+1})$ and $\nabla h(\theta_k)$ then the product of our estimates of these will be an unbiased estimate of their product, and hence the naive stochastic version of 3.2 will be a valid stochastic update rule.

Note that since θ_{k+1} is not technically available at step k, since it depends on P_k, we can simply evaluate θ_{k+1} using the previous value of P given by P_{k-1}, and then compute the corrected θ_{k+1} based on the new update retroactively. However this will involve the re-computation of the gradient at θ_{k+1}, which will double the overall cost of the algorithm and thus should be avoided. In order to do this, Almeida et al. (1998) estimated the derivative as $[\nabla h(\theta_k)]_j [\nabla h(\theta_{k-1})]_j$ instead, arguing that since P usually shouldn't need to change very quickly,

this "lagged" estimate is sufficient.

The particular choice of κ_k advocated is $\alpha[P_k]_{jj}/[v_k]_j$, where $\alpha > 0$ is some learning rate constant, and v_k is a running average of the entry-wise squared gradient estimates, given by $[v_k]_j = \gamma[v_{k-1}]_j + (1-\gamma)[\nabla h(\theta)]_j^2$ for some $0 \leq \gamma < 1$ (typically $\gamma = 0.95$ or thereabouts). The inclusion of the $[P_k]_{jj}$ term is done to ensure that the update to P_k is multiplicative, and the normalization by v_k is done to aid in the convergence of the iteration for P, and provide robustness to basic scaling problems. In fact, this choice of κ_k can be seen as an approximate Newton method applied to the *meta*-optimization problem of optimizing the P_k's, using as the curvature matrix an approximation to the square root of the diagonal of the empirical Fisher.

In order to relate this approach back to the original update scheme of Silva and Almeida (1990), Almeida et al. (1998) considered optimizing $\log[P]_{jj}$ instead of $[P]_{jj}$ via a gradient update rule, with learning rates given by $\alpha/|[\nabla h(\theta_k)]_j[\nabla h(\theta_{k-1})]_j|$. It can be shown that this results in an update identical to eqn. 3.1, but with $u = \exp(\alpha)$ and $d = \exp(-\alpha)$, thus explaining the observation that $u = 1/d$ was required for the original method to behave well with stochastic gradient estimates. Finally, Almeida et al. (1998) observed that since the learning rate is also stochastically estimated and from the same quantities used to compute the gradient (and unlike the choice of κ_k is not based on a running average which is much more stable), the stochastic update will only be an unbiased estimate of the true update when the gradient noise is in a sense symmetric about 0.

Acknowledging the fundamental limitations of their diagonal approach, and in particular its inability to speed the progress of the optimization of h along narrow ravines which are not axis-aligned, Almeida et al. (1998) combined their method with a standard momentum acceleration (see Section 3.4.1), resulting in an update of the form:

$$\theta_{k+1} = \theta_k - P_k \nabla h(\theta_k) + \mu(\theta_k - \theta_{k-1})$$

for a friction constant $0 \leq \mu < 1$.

3.3.4 R-PROP

Following the work of Silva and Almeida (1990), Riedmiller and Braun (1993) proposed the well known method Resilient Backprop (R-PROP), which uses essentially the same adjustment rule for P, but also enforces a maximum and minimum value for each diagonal entry. Additionally, R-PROP uses an unconventional update rule for θ, where the gradient ∇h is replaced by $\operatorname{sgn}(\nabla h)$ (taken entry-wise), giving:

$$\theta_k = \theta_{k-1} - P_k \operatorname{sgn}(\nabla h(\theta_{k-1}))$$

The interpretation here is that the entries of P_k completely control the length of the step, and the gradient is only ever used to determine its direction along each entry of θ.

3.4 The generalized Gauss-Newton matrix

The classical Gauss-Newton matrix (or more simply the Gauss-Newton matrix) is the curvature matrix G which arises in the Gauss-Newton method for non-linear least squares problems. It is applicable to our standard neural network training objective h in the case where $L(y,z) = \frac{1}{2}\|y-z\|^2$, and is given by $G = (\sum_{(x,y)\in S} J_f^\top J_f)/|S|$, where J_f is the Jabobian of $f(x,\theta)$ w.r.t. the parameters θ. It is usually defined as the approximation to the

Hessian H of h (w.r.t. θ) obtained by dropping the second term inside the sum of the following expression for H:

$$H = \frac{1}{|S|} \sum_{(x,y) \in S} \left(J_f^\top J_f - \sum_{j=1}^{m} [y - f(x,\theta)]_j H_{[f]_j} \right)$$

where $H_{[f]_j}$ is the Hessian (w.r.t. θ) of the j-th component of $f(x,\theta)$.

An alternative way to derive the classical Gauss-Newton is to simply replace the non-linear function $f(x,\theta)$ by its own local linear approximation, centered at the current value θ_k of θ. In particular, we replace f by $\tilde{f}(x,\theta) = J_f(\theta - \theta_k) + f(x,\theta_k)$ so that h becomes a quadratic function of θ, with derivative $\nabla h(\theta_k)$ and Hessian given by G.

Schraudolph (2002) showed how the idea of the Gauss-Newton matrix can be generalized to the situation where $L(y,z)$ is any loss function which is convex in z. The generalized formula for G is

$$G = \frac{1}{|S|} \sum_{(x,y) \in S} J_f^\top H_L J_f \tag{3.3}$$

where H_L is the Hessian of $L(y,z)$ w.r.t. z, evaluated at $z = f(x,\theta)$. Because L is convex, H_L will be PSD for each (x,y), and thus so will G. We will call this G the Generalized Gauss-Newton matrix (GGN). Similarly to with the classical Gauss-Newton matrix, the GGN can be obtained by dropping the second term inside the sum of the following expression for the Hessian H:

$$H = \frac{1}{|S|} \sum_{(x,y) \in S} \left(J_f^\top H_L J_f + \sum_{j=1}^{m} \left[\nabla_z L(y,z)|_{z=f(x,\theta)} \right]_j H_{[f]_j} \right)$$

where $\nabla_z L(y,z)|_{z=f(x,\theta)}$ is the gradient of $L(y,z)$ w.r.t. z, evaluated at $z = f(x,\theta)$.

Like the Hessian, the GGN can be used to define a local quadratic model of h given by:

$$M(\delta) = \frac{1}{2} \delta^\top G \delta + \nabla h^\top \delta + h(\theta)$$

In approximate Newton/2nd-order methods based on the GGN, parameter updates are computed by minimizing $M(\delta)$ w.r.t. δ. Since the exact minimizer $\delta^* = -G^{-1} \nabla h$ is often too difficult to compute, practical methods like the Hessian-free optimization of Martens (2010), or Krylov Subspace Descent (Vinyals and Povey, 2012), will only approximately minimize $M(\delta)$.

A key property of G which is not shared by the Hessian H is that it is positive semi-definite (PSD), and can thus be used to define a local quadratic model to the objective h which is bounded. While the unboundedness of local quadratic models defined by the Hessian can be worked around by imposing a trust region, it has nevertheless been observed by various researchers (Schraudolph, 2002; Martens, 2010; Vinyals and Povey, 2012) that G works much better in practice for neural network optimization.

Since computing the whole matrix explicitly is usually too expensive, the GGN is typically accessed via matrix-vector products. To compute such products efficiently one can use the method of Schraudolph (2002), which is a generalization of the well-known method for computing such products with the classical Gauss-Newton. The method is similar in cost and structure to standard backpropagation, although it can sometimes be tricky to implement (see Section 4.4 and 4.5.1).

Martens and Sutskever (2011) showed that the GGN matrix can also be viewed as the Hessian of a partic-

ular approximation of h constructed by replacing $f(x, \theta)$ with its 1st-order approximation w.r.t. θ. Consider the following local convex approximation $\hat{h}$ to h at θ_k that is obtained by taking the first-order approximation $f(x, \theta) \approx f(x, \theta_k) + J\delta$ (where $\delta = \theta - \theta_k$):

$$\hat{h}(\delta) = \frac{1}{|S|} \sum_{(x,y) \in S} L(y, f(x, \theta_k) + J\delta) \tag{3.4}$$

The approximation $\hat{h}$ is convex because it is a composition of a convex function and an affine function. It is easy to see that $\hat{h}$ and h have the same derivative at $\theta = \theta_k$, because

$$\nabla \hat{h} = \frac{1}{|S|} \sum_{(x,y) \in S} J^\top \nabla_z L(y, z)$$

which is precisely the gradient of h at θ_k. And the Hessian of $\hat{h}$ at $\theta = \theta_k$ is precisely the GGN matrix:

$$H_{\hat{h}} = \frac{1}{|S|} \sum_{(x,y) \in S} J^\top H_L J = G$$

Schraudolph (2002) advocated that when computing the GGN, L and f be redefined so that as much as possible of the network's computation is formally performed by L instead of f, while maintaining the convexity of L. This is because, unlike f, L is not linearly approximated in the GGN, and so its associated second-order derivative terms are faithfully captured. What this almost always means in practice is that what is usually thought of as the final non-linearity of the network (i.e. ϕ_ℓ) is folded into L, and the network itself just computes the identity function at its top layer. Interestingly, in many natural situations which occur in practice, doing this gives a much simpler and more elegant expression for H_L. Exactly when and why this happens will be made clear in Section 5.4.

Because contributions made to the GGN for each training case and each individual component of $f(x, \theta)$ are PSD, there can be no cancellation between positive and negative/indefinite contributions. This means that the GGN can be more robustly estimated from subsets of the training data than the Hessian. By analogy, consider how it is harder to estimate the scale of the mean value of a variable when that variable can take on both positive and negative values and has a mean close to 0.

The GGN is arguably a much more "conservative" curvature matrix for similar reasons. For example, if the curvature associated with some direction d is large according to a version of the G obtained by averaging over just a few training cases, this will be reflected to some degree in the full version of G (obtained by averaging over the whole training set). By contrast, if we instead use the Hessian, cancellation with negative curvature from other training cases in the direction d becomes a distinct possibility, and can lead to very small (or negative) overall curvature in that direction. Intuitively, the Hessian-based quadratic model is saying "there is a large quadratic penalty on certain cases for going in this direction but it will be offset by a large quadratic reward on certain other cases". Since negative curvature is arguably less trustworthy over long distances than positive curvature, it is probably better not to allow it to override positive curvature in this manner, and thus the GGN seems superior in this regard.

3.4.1 Momentum

Originating with the "heavy ball" method of Polyak (1964), and in even earlier work on multi-term recurrences for solving linear systems, the momentum method is a simple and inexpensive modification of standard gradient descent, which is known to accelerate convergence in a variety of practical and theoretical settings. The momentum method, as its name suggests, works by storing a "momentum" or "velocity" vector δ and updating θ in the direction of δ instead of the usual gradient. We may write the momentum update as

$$\theta_{k+1} = \theta_k + \alpha\delta_k$$

where δ is updated according to

$$\delta_{k+1} = \mu\delta_k - \nabla h(\theta_k) \tag{3.5}$$

and where $0 \leq \mu < 1$ is a "friction constant". If we expand the recurrence we see that the update δ_k is actually an exponentially decaying averaging of past gradients:

$$\theta_{k+1} = \theta_0 - \alpha \sum_{j=1}^{k} \mu^{k-j} \nabla h(\theta_j)$$

This might give the naive impression that this method is merely blending previous gradients together to produce a "smoothed" average, and that it might behave like standard gradient descent with some possible added stability. But this ignores the dependency of each θ_j on the history of gradients, and the non-trivial back-and-forth interaction that takes place between δ and θ, which results in dynamics which are in many cases quite distinct from gradient descent.

To better understand momentum, we can observe that at each iteration, $\nabla h(\theta_k)$ is added to δ, effectively accelerating the optimization in the direction of steepest descent. Over many iterations, δ tends to accumulate contributions in directions of persistent descent, experiencing the highest velocity in those, while directions which oscillate between locally increasing and decreasing h tend to get canceled and remain small. The intuitive picture is of a heavy ball rolling through a narrow valley (ignoring rotational momentum), accelerating as it rolls down the gentle but persistent slope at the base of the valley, and experiencing short oscillations as it rolls side to side and up and down the steeply rising sides of the valley. And just like in the physical analogy, velocity in all directions is lost slowly over time due to the force of friction of the ball on the surface, as determined by the constant μ.

Of all the approaches considered in this section, momentum is arguably the least well suited to the category of "approximate Newton method". There is no curvature matrix, or anything that could be interpreted as such, and the updates are essentially linear combinations of unmodified gradients, making momentum conform to the formal definition of a "1st-order method". Despite this, it seems to work well in optimization for reasons which are similar to those justifying the use of 2nd-order methods. In particular, directions of low curvature and low gradient, which in the previous analogy would be the base of the valley, are effectively "boosted" through the use of momentum. Meanwhile, directions of high curvature, which would be the sides of the valley, may oscillate but are otherwise kept relatively small (due to how the sign of the derivative of h along these directions will oscillate over time, thus leading to cancellations in the contribution to δ of such directions). It also forms the basis of the Matrix Momentum method which we will discuss in the next subsection.

Polyak (1964) showed that the local convergence properties of momentum are superior to that of gradient descent in the context of deterministic optimization. In particular, while both methods experience linear conver-

gence[8] (assuming appropriate choices of various constants), the rate r of convergence is approximately $1 - 1/\kappa$ for gradient descent and $1-1/\sqrt{\kappa}$ for momentum, where κ is the condition number of the Hessian at the optimum θ^*.

To properly interpret this result we note that $1 - 1/\sqrt{\kappa} < (1 - 1/\kappa)^{\sqrt{\kappa}}$, which implies that gradient descent will require $\sqrt{\kappa}$ times more iterations than momentum to achieve the same ratio of reduction in h, at least as far as these upper bounds are concerned. This difference is significant, since κ is essentially measuring the degree of spread in the local curvature of h, which for problems like deep neural network optimizations can be very large. That being said, it should be noted that optimizing along directions of the very lowest possible curvature might not always be important to obtaining good solutions in practice, and so these bounds are very pessimistic.

A valid question to ask is whether these kinds of bounds are tight, or in other words, are there examples of objective functions where either method exhibits the local convergence properties predicted by these bounds? To answer this question, and gain some insight into the relationship between linear conjugate gradient (abbrv. CG – see Section 4.2) and momentum, we note that the local convergence result for momentum is similar to global convergence upper bounds which are known for CG (as applied to quadratics), which is strongly optimal[9] among 1st-order methods (which are defined as methods whose updates can be expressed as a linear combination of gradients at previous iterates). Fortunately, it is known that these bounds are tight in the sense that there are some particularly hard examples of quadratic optimizations which result in a linear convergence rate for CG with rate $1-1/\sqrt{\kappa}$ (e.g. Li, 2005). Due to the strong optimality of CG, this result implies that these examples also apply to gradient descent with momentum and so the worst case upper bounds for that method (in the context of strongly convex functions) are indeed tight.

Another interesting point to note is that these hard examples are instances of problems where CG will have no distinct advantage in asymptotic[10] convergence rate over (properly calibrated) gradient descent with momentum.

Early work on neural network learning with gradient descent (Plaut et al., 1986) investigated momentum and found it to be beneficial in practice, even for shallower networks. However, when combined with a stochastically sampled gradient, the theory breaks down and many researchers reported that the benefits of using momentum seemed to diminish or vanish completely.

In an attempt to explain these negative results, Wiegerinck et al. (1994) and Orr (1995) analyzed the limiting behavior of stochastic versions of momentum as $\alpha \to 0$ and $\mu \to 1$ s.t. the ratio $\alpha/(1-\mu)^2$ remains constant, and where θ starts very close to a local optimum. In this situation the discrete-time (i.e. iteration based) optimization process can be viewed as a continuous time one, provided that the training cases are somehow modeled as a continuous time stochastic process.

Wiegerinck et al. (1994) found that in the continuous time model described above, the behavior of stochastic gradient descent with momentum coincides with standard stochastic gradient descent, with an effective learning rate[11] of $\alpha/(1-\mu)$. A similar statement can likely be made in the discrete-time case as well.

Orr (1995) examined the situation where the learning rate is annealed according to $\alpha = \alpha_0/k$, as is common practice in stochastic learning. For (the continuous time model of) stochastic gradient with momentum they showed that the expectation of the estimation error $\|\theta_k - \theta_*\|$ decreases as $1/\sqrt{k}$, provided that $\alpha_0/(1-\beta)$ is large enough. For standard gradient descent they provided an analogous result, provided that α_0 is large enough.

[8] This is also sometimes also called "exponential convergence", and means that $\|\theta_{k+1} - \theta^*\| \leq r\|\theta_k - \theta^*\|$ where θ^* is the optimum and $0 < r < 1$ is a constant known as the "rate of convergence"

[9] In the sense that k iterations of CG will always outperform k iterations of another 1st-order method on *any* quadratic optimization problem

[10] It should be noted that CG will always converge to the exact solution after n iterations, at least in hypothetical exact arithmetic implementations. Thus, in the precise meaning of the word "asymptotic", it still does better. However, for any number of iterations *before* n, the results of Li (2005) imply the existence of problems where the reduction in error per iteration is comparable to momentum.

[11] Here, "learning rate" refers to the speed of movement along the idealized continuous time trajectory, and will vary across time (going to zero in particular).

Notably, these two results imply each other in light of the equivalence result described above.

These results should *not* be interpreted as implying that stochastic gradient descent with momentum has no particular advantage over standard SGD. Indeed, the hypothesis that the optimization process starts close to the local minimum is significant, as once this happens, learning in some sense reduces to mere estimation (where *any* stochastic sampling algorithm for θ^* will have, at best, a convergence rate of $1/\sqrt{k}$ in $\|\theta_k - \theta^*\|$). And fortunately, it is our experience that in deep/temporal neural network learning, this final phase of learning is relatively insignificant or unimportant.

Meanwhile, before this final phase of learning occurs, the gradient will arguably follow a distinct trajectory for which a noisy estimate over several training cases will be sufficient to make significant progress in the "correct" overall direction, and acceleration via momentum seems not only conceivable but likely.

3.4.2 Matrix Momentum

In the Matrix Momentum approach of Orr and Leen (1996); Rattray and Saad (1998), the update vector δ is optimized simultaneously with θ to minimize the quadratic model $M(\delta)$ according to a gradient descent iteration.

In particular, θ is updated according to

$$\theta_{k+1} = \theta_k + \delta_k$$

where δ optimized separately and in parallel so that it tends towards $-\alpha B^{-1}\nabla h$, according to:

$$\delta_{k+1} = \delta_k - \eta(\alpha \nabla h(\theta_k) + B_k \delta_k) \tag{3.6}$$

where η is a learning rate for δ, α is the implicit learning rate for θ (at least, if the iteration for δ were allowed to converge to $-\alpha B^{-1}\nabla h$), and the curvature matrix B is computed at θ_k. As with classical momentum, the method can be adapted, perhaps naively, as a stochastic method by simply computing B and ∇h on a single random case, or on subsets of the training data (i.e. mini-batches).

Intuitively, if θ doesn't change too quickly, iteration 3.6 should allow δ to converge to the 2nd-order update $-\alpha B^{-1}\nabla h(\theta_k)$, and hence the overall optimization to resemble Newton's method and achieve the asymptotic convergence enjoyed by that method (at least in the non-stochastic case). However, the various publications on Matrix Momentum don't seem to provide theoretical guarantees as to when this will happen. Instead, evidence is usually given in the form of the empirical observation that Matrix Momentum achieves an asymptotic convergence rate comparable to Newton's method on certain problems.

Orr (1995) found it to be beneficial to separate learning into two distinct phases, as several researchers before had done with other stochastic learning algorithms (e.g. Darken et al. (1992)). The first, an "exploration" or "transient" phase, is characterized by long distances from any local minimum, and somewhat steady movement of θ along some rough path in parameter space in the direction of decreased h. The second, an "asymptotic" or "convergent" phase, is characterized by the iterate θ_k being essentially a noisy estimate of a nearby local optimum, and the main barrier to convergence being the stochasticity in the estimate of the gradient. In this second phase, the difficulty of optimization mostly comes down to statistical estimation of h (or more precisely, its gradient), and convergences rate of stochastic gradient descent is known to be $\mathcal{O}(1/\sqrt{k})$ for general smooth convex functions. In order to automatically detect the phase transition, the heuristic approach of Darken et al. (1992), known as ASTC was used, although this was found to be somewhat unreliable in practice.

In practical implementations of Matrix Momentum, the method is usually only enabled at the point of phase

transition (determined using the ASTC method or manually). One possible reason for doing this is that the asymptotic convergence theory for Newton's method, the algorithm that Matrix Momentum is in a sense approximating, only applies in such a situation. However, practical implementations seem to require doing this in order to achieve stability and in some cases prevent divergence. Scarpetta et al. (1999) found that with careful tuning of η, and the use of the Fisher information matrix instead of the Hessian, Matrix Momentum could be applied during the transient phase, although it wasn't clear if this provided any performance benefits over standard gradient/momentum methods, at least for single-layer neural network learning.

Expanding upon the discussion of Matrix Momentum appearing in Schraudolph (2002), alternative causes for this instability include:

- the optimization of δ by gradient descent being too slow to keep up with the changing definition of B (as θ changes)
- the fact that even 2nd-order updates computed exactly on the whole training set can result in a highly unstable optimization process (assuming various damping or trust-region methods are not used)
- the stochasticity in the estimates of ∇h and B, or in other words, the gradient w.r.t. the local quadratic model

Of these, the third was most often cited as the most likely cause of the instability.

One possible objection to the idea that Matrix Momentum should converge like Newton's method, even in idealized settings[12], is that the iteration 3.6 is itself a gradient descent iteration applied to a local quadratic model of h, and won't necessarily converge any faster than gradient descent applied to h itself. Moreover, the convergence of this iteration will critically depend on the value of η (which must be less than the inverse of the largest eigenvalue of G to ensure convergence). This being said, just as with standard momentum, there may be an interesting interaction between the iteration for δ and the iteration for θ which could result in some acceleration.

Supporting this idea, and providing further insight into the Matrix Momentum approach and its relationship to standard momentum, is the following observation: if we apply Matrix Momentum with $\alpha = 1$ to a quadratic objective, then we recover standard momentum with $\mu = 1$. To see this, note that when h is quadratic, it is equal to its own local quadratic model (assuming B is the Hessian or the Gauss-Newton matrix), and thus the term $\alpha \nabla h(\theta_k) + B\delta_k$ in eqn. 3.6, which is the gradient of the local quadratic model to h, is also the gradient of h, so that eqn. 3.6 and eqn. 3.5 are identical (assuming $\mu = 1$).

Without the assumption that $\alpha = \mu = 1$, the two schemes are still similar, and differ in the sense that $\alpha\mu I$ is implicitly added to B in the case of standard momentum, while $\eta\alpha B$ is added to B in the case of Matrix Momentum. From the perspective of 2nd-order optimization, these represent different choices of update-damping penalty terms applied to the local quadratic model of h.

Despite the potential drawbacks of optimizing δ with a simple gradient-descent iteration, its stationarity property supports its use within an alternating (and possibly stochastic) optimization for θ and δ, where the fact that B and ∇h change at each iteration (due to the changing value of θ) may be somewhat tolerable.

Assuming for simplicity that B and ∇h remain fixed, the iteration for δ is a special case of the iteration

$$\delta_{k+1} = \delta_k - D(\nabla h + B\delta_k) \tag{3.7}$$

where $D = \eta I$. After j steps, such an iteration produces solutions which lie in the k-dimensional preconditioned

[12] Where we start close enough to the local minimum, B is the Hessian, and both ∇h and B are computed on the whole training set.

Krylov subspace

$$K_j = \text{span}\{D\nabla h, DBD\nabla h, (DB)^2 D\nabla h, ..., (DB)^{j-1} D\nabla h\}$$

(assuming for simplicity that $\delta_0 = 0$) and so it would be a strict improvement to use j steps of the preconditioned Conjugate Gradient algorithm (CG), which is known to find the optimal $\delta \in K_j$ in the sense that $M(\delta)$ is minimized.

Truncated Newton methods such as the Hessian-free approach of Martens (2010) can be viewed as a type of Matrix Momentum where ∇h and B are periodically fixed, allowing preconditioned CG to be applied. However, if ∇h and B change at each iteration, preconditioned CG, which is a non-stationary method that doesn't work well as a stochastic optimization algorithm (Schraudolph and Graepel, 2002), won't be appropriate.

3.4.3 Stochastic Meta-Descent

Stochastic Meta-Descent (SMD) (Schraudolph, 1999; Schraudolph, 2002) is a variant of stochastic gradient descent which, like the method of Silva and Almeida (1990), maintains a vector p of learning rates, for each component of the parameter vector given by p (applied coordinate-wise) giving the following update rule:

$$\theta_k = \theta_{k-1} - \text{diag}(p)\nabla h(\theta_{k-1}) \tag{3.8}$$

p is adapted according to a "meta-optimization", which attempts to increase the effectiveness of the gradient updates as measured by h. Since it's more natural to optimize learning rates in log-scale (since they are positive and get applied multiplicatively instead of additively), we instead perform gradient descent on $q = \log(p)$, giving the update

$$q_k = q_{k-1} - \mu \frac{\partial h(\theta_k)}{\partial q} = q_{k-1} - \mu\, \text{diag}(v_k)\nabla h(\theta_k) \tag{3.9}$$

where v_k is the vector of diagonal entries of the diagonal matrix given by $\frac{\partial \theta_k}{\partial q}$. Here, $h(\theta_k)$ is considered to depend on p in some sense (or equivalently q) through θ_k, so that $\frac{\partial h(\theta_k)}{\partial q} = -\frac{\partial h(\theta_k)}{\partial \theta_k}\frac{\partial \theta_k}{\partial q}$.

The main innovation in SMD is to pretend that changes in p retroactively affect previous updates when computing the update to p, so that

$$\begin{aligned} v_k \equiv \frac{\partial \theta_k}{\partial q} &= \frac{\partial \theta_{k-1}}{\partial q} - \frac{\partial\, \text{diag}(p_{k-1})\nabla h(\theta_{k-1})}{\partial q} \\ &= v_{k-1} - \text{diag}(p_{k-1})\nabla h(\theta_{k-1}) - \text{diag}(p_{k-1})H(\theta_{k-1})\frac{\partial \theta_{k-1}}{\partial q} \\ &= v_{k-1} - \text{diag}(p_{k-1})(\nabla h(\theta_{k-1}) + H(\theta_{k-1})v_{k-1}) \end{aligned}$$

However, to be purely consistent with this interpretation of SMD as performing optimization in p, one would have to go back and recompute the entire sequence of updates to θ, as well as recompute the iteration for v_k implied by the above equations. Even if this were somehow a practical thing to do, it seems unlikely that it would be beneficial, as we don't actually think a constant p would be appropriate throughout the entire course of optimization.

While similar meta-adjustment schemes were proposed before, they made no attempt to account for how changes in p affect the entire course of optimization, and instead computed updates to p based on how they would

affect only the most recent update step, giving $v_k = -\operatorname{diag}(p_{k-1})\nabla h(\theta_{k-1})$ (which follows from treating θ_{k-1} as independent of p). Note that with this choice, the change in $[q]_j$ is proportional to $[\nabla h(\theta_{k-1})]_j[\nabla h(\theta_k)]_j$, which is very similar in spirit to the method of Almeida et al. (1998).

Like eqn. 3.7, if all the relevant quantities are fixed the above iteration for v can be viewed as a preconditioned gradient iteration for minimizing the local quadratic approximation of h at θ given by $M(v)$, with $D = \operatorname{diag}(p)$ which will converge to $-H^{-1}\nabla h$. In practice this converge may be achieve if the relevant quantities are sufficiently stable, assuming that H is PSD.

But because h is non-convex and H will not be PSD, and ∇h and H will not be stationary (due to the constantly changing θ), Schraudolph (2002) advocated replacing H with the generalized Gauss-Newton matrix G, and introducing the "forgetting constant" $0 \leq \lambda \leq 1$, giving the iteration:

$$v_k = \lambda v_{k-1} - \operatorname{diag}(p_{k-1})(\nabla h(\theta_k) + \lambda G v_{k-1}) \tag{3.10}$$

While the above discussion was given assuming a stable definition of h, in practice SMD and related methods use stochastic estimates of h (and the related quantities ∇h, H and G) which are computed from a single training case (or a mini-batch). It is not unreasonable to suppose that with the right tuning, the gradient descent iterations for θ, p and v could produce a reasonable algorithm and converge as they do in the batch-case as θ does. That being said, the stability concerns that exist for Matrix Momentum may also apply to SMD to a less extent, and a rigorous theory describing how SMD behaves and when it converges does not exist.

3.4.4 Diagonal Curvature Approximations

Becker and LeCun (1989) proposed a practical approximate Newton method for neural networks which approximates H by the matrix $\hat{H} = \operatorname{diag}(\operatorname{diag}(H))$ constructed from its diagonal. This kind of approximation has several appealing characteristics from the standpoint of practicality. First, it allows the local quadratic model to be minimized trivially since inverting a diagonal matrix amounts to taking the reciprocal of the diagonal entries. This gives the simple component-wise formula for the update δ:

$$[\delta]_j = -[H(\theta)]_{jj}^{-1}[\nabla h(\theta)]_j \tag{3.11}$$

for each component $1 \leq j \leq n$. Second, it can be represented just as efficiently in memory as the gradient, which makes it easy to maintain a running average over past training cases in a stochastic-style implementation, for example.

In practice, $\hat{H}$ may not be positive definite since H won't be, and thus the update $\delta = -\hat{H}^{-1}\nabla h$ may not even be a descent direction, just as can sometimes happen when using standard Newton's method on a non-convex objective. Becker and LeCun (1989) advocates using the entry-wise absolute value of the diagonal to correct this problem. While this will ensure that δ is a descent direction, it is a problematic idea. In particular, it will cause accelerated descent in directions of small negative curvature, and decelerated descent in directions of large negative curvature, which is not obviously the correct thing to do (and is demonstrably bad in some cases).

LeCun et al. (1998) proposed the arguably much better alternative of using a curvature matrix given by the diagonal matrix $\hat{G} = \operatorname{diag}(\operatorname{diag}(G))$, where G is the Gauss-Newton matrix G. Because G is always positive definite, so is $\hat{G}$. LeCun et al. (1998) doesn't provide much justification for using G beyond the guarantee that the diagonal entries will be positive. But following along similar lines as the discussion in sec. 3.4, we can see that the diagonal entries of G will be more robustly estimated than the diagonal entries of H due to cancellation that

can occur between the different training cases when using the Hessian. And when entries of $\mathrm{diag}(G)$ are small (leading to a very large effective learning rate for the corresponding parameter entry), it will be because they are small on most training cases, and not because of some large cancellation involving a few highly negative cases.

As is often done with full Newton type methods, Becker and LeCun (1989) add a constant λ to each diagonal entry (or equivalently, adds λI to $\hat{G}$), before dividing by them as in eqn. 3.11. While such modifications are sometimes said to repair curvature matrices that would otherwise be too badly conditioned, or "prevent the update from blowing up" such explanations are at best over-simplifcations and at worst fallacious. For example, it is not clear why ill-conditioning is necessarily a bad thing, unless it is so extreme that it causes numerical problems when solving for δ. When various eigen-directions of B really do operate at vastly different scales which are accurately modeled over long distances by the corresponding eigenvalues, it is not immediately obvious why one wouldn't want to correct this by multiplying by the inverse Hessian. For example, if the objective function was actually a quadratic doing anything other than this would yield a sub-optimal update in general.

Fortunately, optimization theory provides a comprehensive (and useful) explanation for why adding λI to B is a good idea. In particular, one observes that the local quadratic model $M(\delta)$ from which eqn. 3.11 is implicitly derived, is only a reasonable approximation within some region in parameter space (not necessarily spherical). Updates produced by optimizing a local quadratic model, even ones like eqn. 3.11 that uses a diagonal approximation for the curvature matrix B, can send the parameter very far away from their previous position and thus can often lie far outside this region, especially when some of the eigenvalues of B are very small. It can also be shown that adding λI to B, a technique sometimes called Tikhonov regularization/damping, is equivalent to minimizing the quadratic model δ subject to a constraint on the norm of the solution, thus forcing the solution to lie in a region where the quadratic model may be considered trustworthy. Note that this explanation implies that no single fixed value of λ will be appropriate throughout the entire course of optimization (since the local properties of the objective will change), and so an adaptive adjustment scheme, such as the one present in HF based on the Levenberg-Marquardt method (see Section 4.6.5) should be used.

To actually compute the diagonal of H or G, Becker and LeCun (1989) propose a recursive formula which is similar in both structure and computational cost to the back-propagation method for computing ∇h. Note however that this formula gives only a crude biased approximation to the diagonal.

More recently, Chapelle and Erhan (2011) proposed a randomized method for producing an unbiased estimate of the diagonal Gauss-Newton matrix of an arbitrary function described by a computational graph in the amount of time required to compute the gradient. This type of approach was adapted in the Curvature Propagation algorithm of Martens et al. (2012) to compute unbiased rank-1 estimates of the entire Gauss-Newton or Hessian matrix (from which estimates of the diagonals can be easily and cheaply extracted). These estimates were shown to be much better than the naive rank-1 estimate (given by $(Hv)v^\top = Hvv^\top$ for a random v satisfying $E[vv^\top] = I$) in both theory and practice, and also produced better estimates of the diagonal of a neural network Hessian than the (biased) formula of LeCun after just a single sample. It was also proven in Martens et al. (2012), by using arithmetic circuit theory, that there can be no efficient[13] algorithm for exactly computing the diagonal of either the Hessian of the Gauss-Newton matrix for neural-network type functions.

3.4.5 SGD-QN

Bordes et al. (2009) proposed the SGD-QN approach, which as its name implies, is a hybrid of quasi-Newton algorithms (like BFGS) and SGD. Like many of the approaches discussed in this section it is based on the idea

[13] By efficient we mean of comparable cost as computing the gradient with back-propagation.

of forming a diagonal curvature matrix from gradients. In the case of SGD-QN, this is done using an approach inspired by BFGS-type algorithms, and relies on secant equations.

In particular, after θ is updated at each iteration k (producing θ_{k+1}), a diagonal inverse curvature matrix $P \equiv B^{-1}$ is updated in order to agree with the k-th secant equation (in some weak sense):

$$\theta_{k+1} - \theta_k = P(\nabla h(\theta_{k+1}) - \nabla h(\theta_k))$$

These equations can be viewed as approximations arising from 2nd-order Taylor series expansions, where P would be the inverse Hessian H^{-1} at θ_k. They form the basis of the classical approximate Newton method known as the BFGS method.

In order to compute the quantities in the secant equations, so that they can be used to update P, Bordes et al. (2009) adopted the (obvious) idea from the online L-BFGS algorithm of Schraudolph et al. (2007) and used stochastic approximations to $\nabla h(\theta_{k+1})$ and $\nabla h(\theta_k)$ based on the current mini-batch (i.e. the mini-batch used to estimate the gradient used in the update for θ). However, in a follow-up paper (Bordes et al., 2010) it was determined that using the same mini-batch to compute both the update for P and θ at step k in this way was problematic.

In particular, the dependence of θ_{k+1} on the (random) choice of mini-batch used to compute the k-th update of θ means that an estimate of $\nabla h(\theta_{k+1})$ based on the same mini-batch is not a conditionally unbiased estimator and thus the entire mini-batch-estimated secant equation is not an unbiased estimator of its exact version (as computed on the whole training set). While the authors did not fully explain why this property is particularly important for the derivation of SGD-QN (and the original derivation did not obviously depend on any rigorous statistical estimation properties), they nonetheless found that using the *next* mini-batch to compute the quantities required in the secant equation gave better results, but only if the learning rate for θ was adjusted using a particular heuristic designed to compensate for additional instability introduced by making this choice.

The standard BFGS update is derived by minimizing the change in P, as measured by $\|P_{k+1} - P_k\|_F^2$, subject to the (new) secant equation holding. Applying this strategy to SGD-QN would result in a trivial update given by

$$[P_{k+1}]_{jj} = [\theta_{k+1} - \theta_k]_j / [\nabla h(\theta_{k+1}) - \nabla h(\theta_k)]_j$$

which discards all curvature information collected from prior secant equations. To incorporate information from the entire sequence of secant equations into P, one of several heuristic updates schemes is used by SGD-QN. The first, proposed in the original paper, sets P to be the exponentially decayed average of the trivial exact updates:

$$[P_k]_{jj} = (1 - \gamma_k)[P_{k-1}]_{jj} + \gamma_k \frac{[\theta_{k+1} - \theta_k]_j}{[\nabla h(\theta_{k+1}) - \nabla h(\theta_k)]_j}$$

where γ_k should shrink towards 0 roughly at the rate that $1/k$ does. The second, proposed in the follow-up paper, forgoes the use of an exponential decay and uses a sum instead, and does this not over P but over $B = P^{-1}$. This results in the update:

$$\frac{1}{[P_k]_{jj}} = \frac{1}{[P_k]_{jj}} + \frac{[\nabla h(\theta_{k+1}) - \nabla h(\theta_k)]_j}{[\theta_{k+1} - \theta_k]_j}$$

No explanation or experimental evidence is given as to why either of these schemes is sensible, or which one is preferable. Moreover, these updates are used with a few additional tricks such as thresholding the min and max values of the additive contributions on the RHS's, which may possibly act as a primitive form of update damping

(see Section 4.6).

It is interesting to consider what the contribution to P looks like in the limit as the step size used to update θ approaches zero. We can gain some insight into this by exploiting the well-known identity $\lim_{\epsilon \to 0} \frac{\nabla g(x + \epsilon d) - \nabla g(x)}{\epsilon} = H_g d$ for a general twice-differentiable g and direction vector d, where H_g is the Hessian of g at x. This implies in particular that as the size of the SGD-QN step $\delta_k = \theta_{k+1} - \theta_k$ approaches zero (e.g. because the learning rate is decreasing), we will have:

$$\frac{[\nabla h(\theta_{k+1}) - \nabla h(\theta_k)]_j}{[\theta_{k+1} - \theta_k]_j} \longrightarrow \frac{[H(\theta_k)\delta_k]_j}{[\delta_k]_j} \tag{3.12}$$

Note that the expression on the RHS is completely invariant to the magnitude of δ_k and depends only on its direction.

Interestingly, if we were to pretend δ_k is a random variable which satisfies $\mathrm{E}[[\delta_k]_j / [\delta_k]_m] = 0$ for each $m \neq j$ then the expectation of the RHS of eqn. 3.12 is in fact equal to $[H]_{jj}$, the j-th diagonal entry of H. Additionally, it can be shown that if we replace $[\delta_k]_j$ in the denominator with $[\delta_k]_m$, then expectation turns into $[H]_{jm}$ and so the off-diagonal elements can be recovered in this way.

Many natural choices of distributions for δ_k satisfy this property, including any which draw each component independently and with a distribution satisfying $\mathrm{E}[1/[\delta_k]_j] = \mathrm{E}[[\delta_k]_j] = 0$ for each j (which is true for any zero-mean symmetric distribution).

Note that compared to similar but more advanced estimators, such as Curvature Propagation (Martens et al., 2012), the estimation variance will likely be much larger (this can even be proved when each $[\delta_k]_j$ is drawn independently from a Bernoulli distribution).

In practice $\delta_k = \theta_{k+1} - \theta_k$ will not satisfy this property, even approximately, except possibly in some special circumstances. Instead, δ_k will be a reasonably deterministic function of θ_k at the beginning of optimization, especially if large mini-batches are used. In such a situation, it may however be possible to interpret this bias in the estimate in a positive light, in the sense that the curvature is being estimated along the directions which matter most for optimization.

While the SGD-QN approach can be applied to non-convex objectives such as neural networks, the experiments performed by Bordes et al. (2010) were focused on simpler convex objectives. And no attempt was made to compare the performance of the method to similar approximate Newton approaches that use diagonal curvature matrices.

Chapter 4

Hessian-free Optimization

4.1 Introduction

Hessian-Free optimization (HF) is an approach for unconstrained minimization of real-valued smooth objective functions. Like standard Newton's method, it uses local quadratic approximations to generate update proposals. However, unlike Newton's method, which optimizes its quadratic approximations using matrix-inverses, HF performs a sub-optimization using the linear conjugate gradient algorithm (CG), which doesn't require ever even forming the curvature matrix explicitly, let alone inverting it. It thus belongs to the broad class of approximate Newton methods that are practical for high-dimensional optimization problems (like neural network training).

Different algorithms that use many of the same key principles have appeared in the literature of various communities under different names such as Newton-CG, CG-Steihaug, Newton-Lanczos, and Truncated Newton (Nash, 1984; Nash, 2000; Nocedal and Wright, 2006), but applications to machine learning and especially neural networks, have been limited or non-existent until the work of Martens (2010). In Martens (2010) and later Martens and Sutskever (2011) it was demonstrated that such an approach, if carefully designed and implemented, works very well for training deep neural networks and recurrent neural networks, given sensible random initializations. This was significant because gradient descent methods had previously been observed to be very slow on these problems, making very small progress with each iteration, and sometimes seeming to stop completely (Hinton and Salakhutdinov, 2006; Hochreiter et al., 2000; Hochreiter, 1991).

Recent progress in the area of neural network optimization has revealed that stochastic gradient descent (SGD), used with properly calibrated meta-parameters and a strong form of momentum (Sutskever, Martens, Dahl, and Hinton, 2013), also works well on very deep neural network optimization problems, provided sensible initialization is used.

SGD has a distinct advantage for very large data sets in that it works well with highly stochastic gradients that are estimated on relatively small fractions of the training data. HF on the other hand, by computing 2nd-order updates using high-quality curvature matrices that are not approximated in any way (such as being diagonal (LeCun et al., 1998)), has the advantage that it requires orders of magnitude fewer iterations that SGD ($\sim 10^2$ vs $\sim 10^4 - 10^5$), and can make rapid progress towards the minimum when the gradient (and curvature) are estimated more precisely.

HF continues to be useful in certain deep neural network applications (Sutskever et al., 2011; Graff et al., 2012; Graff et al., 2014; Boulanger-Lewandowski et al., 2012), and in particular those where high-quality estimation of the gradient and curvature is practical (Kingsbury et al., 2012; Sainath et al., 2013b; Sainath et al., 2013c;

Chung et al., 2014), or where the local curvature properties of the objective function are particularly extreme (e.g. when training RNNs on the long-term dependency problems of Hochreiter and Schmidhuber (1997)). HF has also served as a basis or inspiration for several new optimization approaches (Chapelle and Erhan, 2011; Vinyals and Povey, 2012; Kiros, 2013; Sainath et al., 2013a; Bengio et al., 2013; Desjardins et al., 2013), and in particular the recent method of Martens and Grosse (2015) known as K-FAC.

In this chapter we will first describe the basic HF approach, and then examine some performance-improving techniques that are applicable to general gradient-based optimization methods (including stochastic gradient descent [SGD]) as well as others that are more specific to HF and/or neural networks.

4.2 Outline of the approach

We consider the setting of unconstrained minimization of a twice continuously differentiable objective function $h : \mathbb{R}^n \to \mathbb{R}$ w.r.t. to a vector of real-valued parameters $\theta \in \mathbb{R}^n$. 2nd-order optimizers such as HF are derived from the classical Newton's method (a.k.a. the Newton-Raphson method), an approach based on the idea of iteratively optimizing a sequence of local quadratic models/approximations of the objective function in order to produce updates to θ. In the simplest situation, given the current setting of the parameters θ_k, iteration k produces a new iterate θ_{k+1} by minimizing a local quadratic model $M_k(\delta)$ of the objective $h(\theta_k + \delta)$, which is formed using gradient and curvature information local to θ_k. More precisely, we define

$$M_k(\delta) = h(\theta_k) + \nabla h(\theta_k)^\top \delta + \frac{1}{2}\delta^\top B_k \delta \tag{4.1}$$

where B_k is the "curvature matrix", and is chosen to be the Hessian $H(\theta_k)$ of h at θ_k in the case of standard Newton's method. The new iterate θ_{k+1} is computed as $\theta_k + \alpha_k \delta_k^*$ where δ_k^* is the minimizer of eqn. 4.1, and the step-length $\alpha_k \in [0, \infty)$ is typically chosen via a line-search, with a preference for $\alpha_k = 1$. A standard efficient method for performing this kind of line search will be briefly discussed in Section 4.6.8. The multiplication of δ_k by α_k can be viewed as a crude instance of a general technique called "update damping", which we will introduce next, and later discuss in depth in Section 4.6.

When B_k is positive definite, $M(\delta_k)$ will be bounded below and so its minimizer will exist, and will be given by $\delta_k^* = -B_k^{-1}\nabla h(\theta_k)$, which is the standard Newton step. Unfortunately, for many good choices of B_k, such as the Hessian at θ_k, even computing the entire $n \times n$ curvature matrix B_k, let alone inverting it or solving the system $B_k\delta_k = -\nabla h(\theta_k)$, will be impractical for all but very small neural networks.

The main idea in Truncated-Newton methods such as HF is to avoid this costly inversion by partially optimizing the quadratic function M using the linear conjugate gradient algorithm (CG) (Hestenes and Stiefel, 1952), and using the resulting approximate minimizer δ_k to update θ. CG is a specialized optimizer created specifically for quadratic objectives of the form $q(x) = 1/2x^\top Ax - b^\top x$ where $A \in \mathbb{R}^{n\times n}$ is positive definite, and $b \in \mathbb{R}^n$. It works by constructing the update from a sequence of vectors which have the property that they are mutually "A-conjugate" and can thus be optimized independently in sequence. To apply CG to eqn. 4.1 we take $x = \delta$, $A = B_k$ and $b = \nabla h(\theta_k)$, noting that the constant term $h(\theta_k)$ can be ignored.

> **Note:** From this point forward, we will abbreviate M_k with M and B_k with B when the subscript is implied by the context.

CG has the nice property that it only requires access to matrix-vectors products with the curvature matrix B (which can be computed *much* more efficiently than the entire matrix in many cases, as we will discuss in Section

4.3), and it has a fixed-size storage overhead of a few n-dimensional vectors. Moreover, CG is a very powerful algorithm, which after i iterations, will find the provably optimal solution of any convex quadratic function $q(x)$ over the Krylov subspace $K_i(A, r_0) \equiv \text{span}\{r_0, Ar_0, A^2r_0, ..., A^{i-1}r_0\}$, where $r_0 = Ax_0 - b$ and x_0 is the initial solution (Shewchuk, 1994). Any other gradient based method applied directly to a quadratic function like M, even a very powerful one like Nesterov's accelerated gradient descent (Nesterov, 1983a), can also be shown to produce solutions which lie in the Krylov subspace, and thus, assuming exact arithmetic, will always be strictly outperformed by CG given the same number of iterations[1].

Fortunately, in addition to these strong optimality properties, CG works extremely well in practice and may often converge in a number of iterations $i \ll n$, depending on the structure of B. But even when it does not converge it tends to make very good partial progress.

Algorithm 2 Preconditioned conjugate gradient algorithm (PCG)

inputs: b, A, x_0, P
$r_0 \leftarrow Ax_0 - b$
$y_0 \leftarrow$ solution of $Py = r_0$
$p_0 \leftarrow -y_0$
$i \leftarrow 0$
while termination conditions do not apply **do**
  $\alpha_i \leftarrow \dfrac{r_i^\top y_i}{p_i^\top A p_i}$
  $x_{i+1} \leftarrow x_i + \alpha_i p_i$
  $r_{i+1} \leftarrow r_i + \alpha_i A p_i$
  $y_{i+1} \leftarrow$ solution of $Py = r_{i+1}$
  $\beta_{i+1} \leftarrow \dfrac{r_{i+1}^\top y_{i+1}}{r_i^\top y_i}$
  $p_{i+1} \leftarrow -y_{i+1} + \beta_{i+1} p_i$
  $i \leftarrow i + 1$
end while
output: x_i

The preconditioned CG algorithm is given in Algorithm 2. Note that Ap_i only needs to be computed once in each iteration of the main loop, and the quadratic objective $q(x_i)$ can be cheaply computed as $q(x_i) = \frac{1}{2}(r_i - b)^\top x_i$. Also note that any notation such as α_i or y_i should not be confused with the other uses of these symbols that occur elsewhere in this report. Preconditioning allows CG to operate within a transformed coordinate system and a good choice of the preconditioning matrix P can substantially accelerate the method. This is possible despite the previously claimed optimality of CG because preconditioning transforms the Krylov subspace itself. Preconditioning, methods for implementing it, its role within HF, and its subtle interaction with other parts of the HF approach, will be discussed in Section 4.9.

With practicality in mind, one can terminate CG according to various criteria, balancing the quality of the solution with the number of iterations (and hence the number of matrix-vector products) required to obtain it. The approach taken by Martens (2010) was to terminate CG based on a measure of relative progress optimizing M, computed as

$$s_j = \frac{M(x_j) - M(x_{j-k})}{M(x_j) - M(0)}$$

where x_j is the j-th iterate of CG and k is the size of the window over which the average is computed, which

[1]This being said, it is possible to construct quadratic optimization problems where CG will perform essentially no better than accelerated gradient descent. Although it is also possible to construct ones where CG converges in only a few iterations while accelerated gradient descent will take much longer.

should be increased with j. A reasonable choice that works well in practice is $k = \max(10, j/10)$. CG can be terminated at iteration j when

$$s_j < 0.0001 \tag{4.2}$$

or some other such constant. Depending on the situation it may make more sense to truncate earlier to find a more economical trade-off between relative progress and computation.

However, deciding when to terminate CG turns out to be a much more complex and subtle issue than implied by the above discussion, and in Section 4.6.7 we will discuss additional reasons to terminate CG that have nothing directly to do with the value of M. In particular, earlier truncations may sometimes have a beneficial damping effect, producing updates that give a better improvement in h than would be obtained by a fully converged solution (or equivalently, one produced by exact inversion of the curvature matrix).

When h is non-convex (such as it is for neural networks), B will sometimes be indefinite, and so the minimizer of M may not exist. In particular, progressively larger δ's may produce arbitrarily low values of M, leading to nonsensical or undefined updates. This issue can be viewed as an extreme example of the general problem that the quadratic model M is only a crude local approximation to h, and so its minimizer (assuming it even exists), might lie in a region of $\mathbb{R}^n$ where the approximation breaks down, sometimes catastrophically. While the aforementioned line-search can remedy this problem to some degree, this is a general problem with 2nd-order optimization that must be carefully addressed. Ways to do this are sometimes called "damping methods", a term which we shall use here, and are typically designed to keep the solution for δ within a region of $\mathbb{R}^n$ where M remains a good approximation to h, either by explicitly restricting the solution, or by augmenting M with various penalty terms which encourage the solution to be "small" in various senses. Such approaches must be used with care, since restricting/penalizing the optimization of M too much will result in very safe and reliable updates which are nonetheless useless due to being too "small". In Section 4.6 we will discuss various general damping methods in 2nd-order optimization, and some which are more specific to HF.

While the damping methods such as those mentioned above allow one to optimize M even when B is indefinite, there is another way to deal with the indefiniteness problem directly. The classical Gauss-Newton algorithm for non-linear least squares uses a positive semi-definite curvature matrix which is viewed as an approximation to the Hessian, and Schraudolph (2002) was able to generalize this idea to cover a much larger class of objective functions that include most neural network training objectives. This "generalized Gauss-Newton matrix" (GGN), is also guaranteed to be positive semi-definite, and tends to work much better than the Hessian in practice as a curvature matrix when optimizing non-convex objectives. While using the GGN matrix does not eliminate the need for damping Martens (2010) nonetheless found that it allows for much weaker damping and gives rise to much better updates as a result. The GGN is discussed in general in Section 3.4, and the computational aspects of using the GGN matrix and its role within HF are discussed in detail in Section 4.4.

Objective functions $h(\theta)$ that appear in machine learning are almost always defined as average losses over a training set S. This means that the gradient and curvature-matrix vector products are also averages over S. In other words, we have:

$$h(\theta) = \frac{1}{|S|} \sum_{(x,y) \in S} L(y, f(x, \theta))$$

$$\nabla h(\theta) = \frac{1}{|S|} \sum_{(x,y) \in S} \nabla L(y, f(x, \theta))$$

$$B(\theta)v = \frac{1}{|S|} \sum_{(x,y) \in S} B(\theta; (x, y))v$$

where $B(\theta; (x, y))$ the curvature matrix associated the loss on training case (x, y).

In order to make HF practical for large datasets it is necessary to estimate the gradient and curvature matrix-vector products using subsets of the training data, called "mini-batches." And while it may seem natural to compute the matrix-vector products required by CG using a newly sampled mini-batch at each iteration of Algorithm 2, CG is unfortunately not designed to handle this kind of "stochasticity" and its theory depends very much on a stable definition of B for concepts like B-conjugacy to even make sense. And in practice, we have found that such an approach does not seem to work very well, and results in CG itself diverging in some cases. The solution advocated by Martens (2010) and independently by Byrd et al. (2011) is to fix the mini-batch used to define B for the entire run of CG. Mini-batches and the practical issues which arise when using them will be discussed in more depth in Section 4.10.

Algorithm 3 High-level outline for the basic Hessian-free approach. Various details have been purposefully left unstated, and some aspects will be subject to change throughout this report.

inputs: θ_1, λ
$\delta_1 \leftarrow \vec{0}$
$k \leftarrow 1$
while solution is not satisfactory **do**
  Select a mini-batch $S' \subset S$ of training cases for the gradient — Section 4.10
  $b \leftarrow -\nabla h(\theta_k)$ on S' — Algorithm 1
  Select a mini-batch $S'' \subset S$ for the curvature — Section 4.10
  Compute a preconditioner P at θ_k — Section 4.9
  Compute a damping matrix D_k — Section 4.6
  Define $A(v) \equiv G(\theta_k)v + \lambda D_k v$ on S'' — Section 3.4
  Choose a decay constant $\zeta \in [0, 1]$ — Section 4.8
  $\delta_k \leftarrow \text{PCG}(b, A, \zeta\delta_{k-1}, P)$ — Algorithm 2
  Update λ with the Levenberg-Marquardt method — Section 4.6.5
  Choose/compute a step-size α — Section 4.6.8
  $\theta_{k+1} \leftarrow \theta_k + \alpha\delta_k$
  $k \leftarrow k + 1$
end while

4.3 Exact multiplication by the Hessian

To use the Hessian H of h as the curvature matrix B within HF we need an algorithm to efficiently compute matrix-vector products with arbitrary vectors $v \in \mathbb{R}^n$. Noting that the Hessian is the Jacobian of the gradient, we have that the Hessian-vector product $H(\theta)v$ is the directional derivative of the gradient $\nabla h(\theta)$ in the direction v, and so by the definition of directions derivatives,

$$H(\theta)v = \lim_{\varepsilon \to 0} \frac{\nabla h(\theta + \varepsilon v) - \nabla h(\theta)}{\varepsilon}$$

This equation implies a finite-differences algorithm for computing Hv at the cost of a single extra gradient evaluation. But in practice, and in particular when dealing with highly nonlinear functions like neural network training objectives, methods that use finite differences suffer from significant numerical issues, which can make them generally undesirable and perhaps even unusable in some situations.

Fortunately, there is a method for computing the sought-after directional derivative in a numerically stable way

that does not resort to finite differences. In the optimization theory literature, the method is known as "forward-differentiation" (Wengert, 1964; Nocedal and Wright, 2006), although we follow the exposition of Pearlmutter (1994), who rediscovered it for neural networks and other related models. The idea is to make repeated use of the chain rule, much like in the backpropagation algorithm, to compute the required derivatives with respect to the value of every node in the computational graph of the gradient. We formalize this notion by introducing the R_v-notation. Let $\mathrm{R}_v x$ denote the directional derivative of some θ-dependent variable x in direction v. We have

$$\mathrm{R}_v x = \lim_{\varepsilon \to 0} \frac{x(\theta + \varepsilon v) - x(\theta)}{\varepsilon} = \frac{\partial x}{\partial \theta} v \tag{4.3}$$

Being a derivative, the $\mathrm{R}_v(\cdot)$ operator obeys the usual rules of differentiation

$$\mathrm{R}_v(x + y) = \mathrm{R}_v x + \mathrm{R}_v y \qquad \text{linearity} \tag{4.4}$$

$$\mathrm{R}_v(xy) = (\mathrm{R}_v x)y + x\mathrm{R}_v y \qquad \text{product rule} \tag{4.5}$$

$$\mathrm{R}_v(\gamma(x)) = (\mathrm{R}_v x) J_\gamma(x) \qquad \text{chain rule} \tag{4.6}$$

where $J_\gamma(x)$ denotes the Jacobian of γ. From this point on we will abbreviate R_v as simply "R" to keep the notation compact.

Noting that $Hv = \mathrm{R}(\nabla h(\theta))$, computing the Hessian-vector product amounts to computing $\mathrm{R}(\nabla h(\theta))$ by applying these rules recursively to the computational graph for $\nabla h(\theta)$, in a way analogous to back-propagation (but operating forward instead of backwards).

To make this precise, we will formalize the notion of a computational graph for an arbitrary vector-valued function $g(\theta)$, which can be thought of as a special kind of graph which implements the computation of a given function by breaking it down as a collection of simpler operations, represented by L nodes, with various input-output dependencies between the nodes indicated by directed edges. The nodes of the computational graph are vector valued, and each node i computes an arbitrary differentiable functions $u_i = \gamma_i(z_i)$ of their input z_i. Each input vector z_i is formally the concatenation of the output of each of node i's parent nodes (given by the set P_i). The input θ is distributed over a set of input nodes $\mathcal{I} \subset \{1, ..., L\}$ and the outputs are computed at output nodes $\mathcal{O} \subset \{1, ..., L\}$.

In summary, the function $g(\theta)$ is computed according to the following procedure:

1. For each $i \in \mathcal{I}$ set u_i according to entries of θ

2. For i from 1 to L such that $i \notin \mathcal{I}$:

$$z_i = \text{concat}_{j \in P_i} u_j$$
$$u_i = \gamma_i(z_i)$$

3. Output $g(\theta)$ according to the values in $\{u_i\}_{i \in \mathcal{O}}$

The advantage of the computational graph formalism is that it allows the application of the R-operator to be performed in a fool-proof and mechanical way that can be automated. In particular, our function $\mathrm{R}(g(\theta))$ can be computed as follows:

1. For each $i \in \mathcal{I}$ set $\mathrm{R}u_i$ according to entries of v (which correspond to entries of θ)

2. For i from 1 to M such that $i \notin \mathcal{I}$:

$$\mathrm{R}z_i = \mathrm{concat}_{j \in P_i} \mathrm{R}u_j \tag{4.7}$$

$$\mathrm{R}u_i = J_{\gamma_i}(z_i)\mathrm{R}z_i \tag{4.8}$$

3. Output $\mathrm{R}(g(\theta))$ according to the values in $\{\mathrm{R}z_i\}_{i \in \mathcal{O}}$

where $J_{\gamma_i}(z_i)$ is the Jacobian of γ_i.

In general, computing $J_{\gamma_i}(z_i)$ (or more simply multiplying it by a vector) is simple[2] and is of comparable cost to computing $\gamma_i(z_i)$, which makes computing the Hessian-vector product using this method comparable to the cost of the gradient. Notice however that we need to have each z_i available in order to evaluate $J_{\gamma_i}(z_i)$ in general, so all of the z_i's (or equivalently all of the u_i's) must either be computed in tandem with the $\mathrm{R}u_i$'s and $\mathrm{R}z_i$'s (making the cost of the Hessian-vector product roughly comparable to the cost of two evaluations of the gradient), or be precomputed and cached (e.g. during the initial computation of the gradient ∇h).

When using an iterative algorithm like CG that requires multiple Hessian-vector products for the same θ, caching can save considerable computation, but as discussed in Section 4.5, may require considerable extra storage when computing matrix-vector products over large mini-batches.

Algorithm 4.3 gives the pseudo-code for computing the Hessian-vector product associated with the feedforward neural network defined in Section 2.1. The parameter vector θ defines the weight matrices $(W_1, \ldots, W_\ell)$ and v maps analogously to $(\mathrm{R}W_1, \ldots, \mathrm{R}W_\ell)$.

This algorithm was derived by applying the rules 4.4–4.6 to each line of Algorithm 1, where various required quantities such as a_i are assumed to be available either because they are cached, or by running the corresponding lines of Algorithm 1 in tandem.

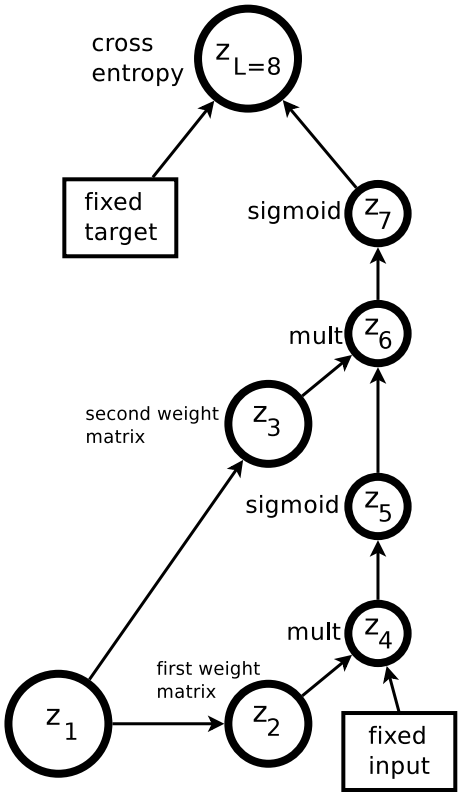

Figure 4.1: An example of a computational graph of the loss of a neural network objective. The weights are considered the inputs here.

4.4 The role of the generalized Gauss-Newton matrix

The indefiniteness of the Hessian is problematic for 2nd-order optimization of non-convex functions because an indefinite curvature matrix B may result in a quadratic M which is not bounded below and thus does not have a minimizer to use as a candidate update. This problem can be addressed in a multitude of ways. For example, imposing a trust-region (Section 4.6.6) will constrain the optimization, or a penalty-based damping method (Section 4.6.1) will effectively add a positive semi-definite (PSD) contribution to B which may render it positive definite. Another solution specific to truncated Newton methods like HF is to truncate CG as soon as it generates a conjugate direction with negative curvature (i.e., when $p_i^\top A p_i < 0$ in Algorithm 2), a solution which

[2] If this is not the case then node i should be split into several simpler operations.

Algorithm 4 An algorithm for computing $H(\theta;(x,y))v$ in feedforward neural networks.

input: v mapped to $(\mathrm{R}W_1, \ldots, \mathrm{R}W_\ell)$
$\mathrm{R}a_0 \leftarrow 0$ (since a_0 is not a function of the parameters)

/* Forward pass */
for all i **from** 1 **to** ℓ **do**
  $\mathrm{R}s_i \leftarrow \mathrm{R}W_i \bar{a}_{i-1} + W_i \mathrm{R}\bar{a}_{i-1}$ (product rule)
  $\mathrm{R}a_i \leftarrow \mathrm{R}s_i \phi_i'(s_i)$ (chain rule)
  /* Note that $R\bar{a}_i$ may be obtained from Ra_i by appending a column of zeros. */
end for

$$\mathrm{R}\mathcal{D}a_\ell \leftarrow \mathrm{R}\left(\left.\frac{\partial L(y,z)}{\partial z}\right|_{z=a_\ell} \right) = \frac{\partial \left[\left.\frac{\partial L(y,z)}{\partial z}\right|_{z=a_\ell} \right]}{\partial a_\ell} \mathrm{R}a_\ell = \left.\frac{\partial^2 L(y,z)}{\partial z^2}\right|_{z=a_\ell} \mathrm{R}a_\ell$$

/* Backwards pass */
for all i **from** ℓ **downto** 1 **do**
  $\mathrm{R}\mathcal{D}s_i \leftarrow \mathrm{R}\mathcal{D}a_i \odot \phi_i'(s_i) + \mathcal{D}a_i \odot \mathrm{R}\phi_i'(s_i)$ (product rule)
  $= \mathrm{R}\mathcal{D}a_i \odot \phi_i'(s_i) + \mathcal{D}a_i \odot \phi_i''(s_i) \odot \mathrm{R}s_i$ (chain rule)
  $\mathrm{R}\mathcal{D}W_i \leftarrow \mathrm{R}\mathcal{D}s_i \bar{a}_{i-1}^\top + \mathcal{D}s_i \mathrm{R}\bar{a}_{i-1}^\top$ (product rule)
  $\mathrm{R}\mathcal{D}\bar{a}_{i-1} \leftarrow \mathrm{R}W_i^\top \mathcal{D}s_i + W_i^\top \mathrm{R}\mathcal{D}s_i$ (product rule)
end for

output: $H(\theta;(x,y))v$ as mapped from $(\mathrm{R}\mathcal{D}W_1, \ldots, \mathrm{R}\mathcal{D}W_\ell)$.

may be useful in some applications but which we have not found to be particularly effective for neural network training.

Based on our experience, the best solution to the indefiniteness problem is to instead use the generalized Gauss-Newton (GGN) matrix proposed by Schraudolph (2002), which is a provably positive semidefinite curvature matrix that can be viewed as an approximation to the Hessian. The GGN, which we denote by G, is discussed in detail in Section 3.4.

4.4.1 Multiplying by the Generalized Gauss-Newton matrix

For the GGN matrix to be useful in the context of HF, we need an efficient algorithm for computing the Gv products. Methods for multiplying by the classical Gauss-Newton matrix are well-known in the optimization literature (Nocedal and Wright, 2006), and these methods were generalized by Schraudolph (2002) for the GGN matrix, using an approach which we will now describe.

As we saw in Section 3.4, the GGN matrix can be expressed as the average of a product of three matrices as $G = \frac{1}{|S|}\sum_{(x,y)\in S} J^\top H_L J$. Thus multiplication of a vector v by the GGN matrix for a single case (x,y) amounts to the sequential multiplication of that vector by these 3 matrices. First, the product Jv is a Jacobian times vector and is therefore precisely equal to the directional derivative $\mathrm{R}_v(f(x,\theta))$, and thus can be efficiently computed with the R-method as in Section 4.3. Next, given that the loss function H_L is usually simple, multiplication of Jv by H_L is also simple (see Section 4.4.2). Finally, we multiply the vector $H_L Jv$ by the matrix $J^\top$ using the backpropagation algorithm [3].

[3] Note that the backpropagation algorithm can be interpreted as taking the derivative of the loss w.r.t. z ($\nabla_z L(y,z)$) as input, and returning the derivative w.r.t. θ, namely $J^\top \nabla_z L(y,z)$. But we can replace $\nabla_z L(y,z)$ in this process with any vector v that we want, thus computing $J^\top v$ instead.

Algorithm 5 An algorithm for computing $G(\theta;(x,y))v$ in feedforward neural networks.

input: v mapped to $(\mathrm{R}W_1,\ldots,\mathrm{R}W_\ell)$
$\mathrm{R}a_0 \leftarrow 0$

/* Forward pass */
for all i **from** 1 **to** ℓ **do**
  $\mathrm{R}s_i \leftarrow \mathrm{R}W_i\bar{a}_{i-1} + W_i\mathrm{R}\bar{a}_{i-1}$
  $\mathrm{R}a_i \leftarrow \mathrm{R}s_i\phi_i'(s_i)$
  /* Note that $R\bar{a}_i$ may be obtained from Ra_i by appending a column of zeros. */
end for

$$\mathrm{R}\mathcal{D}a_\ell \leftarrow \left.\frac{\partial^2 L(y,z)}{\partial z^2}\right|_{z=a_\ell} \mathrm{R}a_\ell$$

/* Backwards pass */
for all i **from** ℓ **downto** 1 **do**
  $\mathrm{R}\mathcal{D}s_i \leftarrow \mathrm{R}\mathcal{D}a_i \odot \phi_i'(s_i)$
  $\mathrm{R}\mathcal{D}W_i \leftarrow \mathrm{R}\mathcal{D}s_i\bar{a}_{i-1}^\top$
  $\mathrm{R}\mathcal{D}\bar{a}_{i-1} \leftarrow W_i^\top\mathrm{R}\mathcal{D}s_i$
end for

output: $G(\theta;(x,y))v$ as mapped from $(\mathrm{R}\mathcal{D}W_1,\ldots,\mathrm{R}\mathcal{D}W_\ell)$.

As observed by Martens and Sutskever (2011), the second interpretation of the GGN matrix given in Section 3.4 immediately implies an alternative method for computing Gv products. In particular, we can use the R method from Section 4.3 to efficiently multiply by the Hessian of $\hat{h}$, given a computational graph for $\nabla\hat{h}$. While doing this would require one to replace the forward pass (corresponding to f) with a multiplication by the Jacobian of f evaluated at θ_k, a simpler approach is just to modify the algorithm for computing ∇h so that in the backwards pass, all non-differential quantities computed during the forward pass (like the a_i, but not $\mathcal{D}a_i$) are treated as "constants", which while they are computed from θ_k, are considered formally independent of θ.

Algorithm 5 multiplies by the GGN matrix for the special case of a feedforward neural network and is derived using this second technique. Note that $\left.\frac{\partial^2 L(y,z)}{\partial z^2}\right|_{z=a_\ell}$ is just H_L.

4.4.2 Typical losses

In this section we present a number of typical loss functions and their Hessians (Table 4.1). These losses are

Name	$\hat{y}$	$L(y,z)$	$\nabla_z L(y,z)$	H_L
Squared error	$\hat{y} = z$	$\frac{1}{2}\lVert\hat{y}-y\rVert^2$	$\hat{y}-y$	I
Cross-entropy	$\hat{y} = \mathrm{Sigmoid}(z)$	$-y\log\hat{y} - (1-y)\log(1-\hat{y})$	$\hat{y}-y$	$\mathrm{diag}(\hat{y}\odot(1-\hat{y}))$
Cross-entropy (multi-dim)	$\hat{y} = \mathrm{Softmax}(z)$	$-\sum_i [y]_i \log[\hat{y}]_i$	$\hat{y}-y$	$\mathrm{diag}(\hat{y}) - \hat{y}\hat{y}^\top$

Table 4.1: Typical losses with their derivatives and Hessians. The loss $L(y,z)$ and the nonlinearity $\hat{y}(z)$ are "matching", which means that the Hessian is independent of the target y and is PSD.

convex and it is easy to multiply by their Hessians without explicitly forming the matrix, since they are each either diagonal or the sum of a diagonal and a rank-1 term.

When applying this formulation to FNNs, note that because it formally includes the computation of the predictions $\hat{y}$ from z in the loss function itself instead of in the activation function ϕ_ℓ at the output layer, and ϕ_ℓ

should be set to the identity function.

4.4.3 Dealing with non-convex losses

We may sometimes want to have a non-convex loss function. The generalized Gauss-Newton matrix construction will not produce a positive definite matrix in this case because the GGN matrix $J^\top H_L J$ will usually be PSD only when H_L is, which is a problem that can be addressed in one of several ways. For example, if our loss is $L(y, z) = \frac{1}{2}\|y - \tanh(z)\|^2$, which is non-convex, we could formally treat the element-wise $\tanh$ nonlinearity as being part of $f(x, \theta)$ (replacing f with $\tanh \circ f(x, \theta)$), and redefine the loss L as $\frac{1}{2}\|y - z\|^2$. Another trick which may work would be to approximate the Hessian H_L of the loss with a positive definite matrix, which could be done, say, by adding a scaled multiple of the diagonal to H_L, or by taking the eigen-decomposition of H_L and discarding the eigenvectors that have negative eigenvalues.

4.5 Implementation details

4.5.1 Verifying the correctness of G products

A well-known pitfall for neural networks practitioners is an incorrect implementation of the backpropagation algorithm for computing the gradient, which is hard to diagnose without having a correct implementation to compare against. The usual procedure is to re-implement the gradient computation using finite differences and verify that the two implementations agree, up to some reasonable precision.

To verify the correctness of an implementation of a Truncated-Newton optimizer like HF, we must also verify the correctness of the curvature-matrix vector products. When $B = H$, there are well-known black-box finite differentiation implementations available which can be used for this purpose. Thus we will concentrate on how to verify the correctness of the Gv products.

Given that $G = \frac{1}{|S|}\sum_{(x,y)\in S} J^\top H_L J$ so $Gv = \frac{1}{|S|}\sum_{(x,y)\in S} J^\top (H_L(Jv))$, computing the matrix-vector product with G via finite differences amounts to computing matrix-vector products of the forms Jw, $H_L w$ and $J^\top w$ for arbitrary vectors w of the appropriate dimensions.

1. For Jw we can compute $(f(x, \theta + \varepsilon w) - f(x, \theta - \varepsilon w))/(2\varepsilon)$ for a small ε.

2. For $H_L w$ we can simply approximate H_L using one of the aforementioned finite-differences implementations that are available for approximating Hessians.

3. For $J^\top w$ we can exploit the fact that $[J]^\top_{j,i} = [Je_j]_i$ where e_j is the j-th standard basis vector, and use the method in point 1 to approximate Je_j

To be especially thorough, one should probably test that Ge_j agrees with its finite differences version for each j, effectively constructing the whole matrix G.

For this kind of finite-differences numerical differentiation to be practical it is important to use small toy versions of the target networks, with much fewer units in each layer, and smaller values for the depth ℓ or sequence length T (such as 4). In most situations, a good value of ε is often around 10^{-4}, and it is possible to achieve a relative estimation error from the finite differences approximation of around 10^{-6}, assuming a high-precision floating point implementation (i.e. float64 rather than float32).

It is also important to use random θ's that are of a reasonable scale. Parameters that are too small will fail to engage the nonlinearities, leaving them in their "linear regions" and making them behave like linear functions,

while parameters that are too large may cause "saturation" of the units of the network, making them behave like step-functions (the opposite extreme). In either case, a proposed implementation of some exact derivative computation could match the finite differences versions to high precision despite being incorrect, as the local derivatives of the activation functions may be constant or even zero.

Another option to consider when implementing complex gradient/matrix computations is to use an automatic differentiation system package such as Theano (Bergstra et al., 2010). This approach has the advantage of being mostly fool proof, at the possible cost of customization and efficiency (e.g. it may be hard to cache the activities using previously discussed techniques).

4.6 Damping

While unmodified Newton's method may work well for certain objectives, it tends to do very poorly if applied directly to highly nonlinear objective functions, such as those which arise when training neural networks. The reason for this failure has to do with the fact that the minimizer δ^* of the quadratic approximation $M(\delta)$ may be very large and "aggressive" in the early and the intermediate stages of the optimization, in the sense that it is often located far beyond the region where the quadratic approximation is reasonably trust-worthy.

The convergence theory for non-convex smooth optimization problems (which include neural network training objectives) describes what happens only when the optimization process gets close enough to a local minimum so that the steps taken are small compared to the change in curvature (e.g. as measured by the Lipschitz constant of the Hessian). In such a situation, the quadratic model will always be highly accurate at δ^*, and so one can fully optimize M and generate a sequence of updates which will converge "quadratically" to the local minimum of h. And for some very simple optimization problems which can arise in practice it may even be possible to apply unmodified Newton's method without any trouble, ignoring the theoretical requirement of proximity to a local minimum. However, for neural network training objectives, and in particular deep feedforward networks and RNNs, the necessity of these proximity assumptions quickly becomes clear after basic experiments, where such naive 2nd-order optimization tends to diverge rapidly from most sensible random initializations of θ.

The solution that is sometimes advocated for this problem is to use a more stable and reliable method, like gradient-descent for the beginning of optimization, and then switch later to 2nd-order methods for "fine convergence". Optimization theory guarantees that as long as the learning rate constant is sufficiently small, gradient descent will converge from any starting point. But precise convergence is often not necessary (Bottou and Bousquet, 2011), or even undesirable (due to issues of overfitting). Instead, if we believe that making use of curvature information can be beneficial in constructing updates long before the "fine convergence" regime described by local convergence theory sets in, it may be worthwhile to consider how to make more careful and conservative use of curvature information in order to construct large but still sensible update proposals, instead of defaulting to 1st-order ones out of necessity.

"Damping", a term used mostly in the engineering literature, and one which we will adopt here, refers to methods which modify M or constrain the optimization over it in order to make it more likely that the resulting update δ will lie in a region where M remains a reasonable approximation to h and hence yield a substantial reduction. The key difficulty with damping methods is that if they are overused or improperly calibrated, the resulting updates will be "reliable" but also be too small and insignificant (as measured by the reduction in h).

An effective damping method is of critical importance to the performance of a 2nd-order method, and obtaining the best results will likely require the use of a variety of different techniques, whose usefulness depends both on the particular application and the underlying 2nd-order method. In this section we will discuss some generic

damping methods that can be used in 2nd-order optimizers and how to apply them in HF (either separately or in some combination) along with methods which are specific to neural networks and HF.

One thing to keep in mind when reading this section is that while the immediate goal of damping methods is to increase the quality of the parameter update produced by optimizing M (as measured by the immediate improvement in the objective h), damping methods can and will have an important influence on the global optimization performance of 2nd-order optimizers when applied to multimodal objectives functions, in ways that are sometimes difficult to predict or explain, and will be problem dependent. For example, we have observed empirically that on difficult neural network training objectives, damping schemes which tend to produce updates that give the best reductions in h in the short term, may not always yield the best global optimization performance in the long term.

4.6.1 Tikhonov Damping

Tikhonov regularization or Tikhonov damping (the latter of which we will use to avoid confusion with the more common meaning of "regularization" in the machine learning literature) is arguably the most well-known damping method, and works by penalizing the squared magnitude $\|\delta\|^2$ of the update δ by introducing an additional quadratic penalty term into the quadratic model M. Thus, instead of minimizing M, we minimize a "damped" quadratic

$$\hat{M}(\delta) \equiv M(\delta) + \frac{\lambda}{2}\delta^\top\delta = f(\theta) + \nabla h(\theta)^\top\delta + \frac{1}{2}\delta^\top\hat{B}\delta$$

where $\hat{B} = B + \lambda I$, where $\lambda \geq 0$ is a scalar parameter determining the "strength" of the damping. Computing the matrix-vector product with $\hat{B}$ is straightforward since $\hat{B}v = (B + \lambda I)v = Bv + \lambda v$.

As $\lambda \to \infty$, the damped curvature matrix $\hat{B}$ tends to a multiple of the identity and the minimizer δ^* has the property that $\delta^* \to \nabla h(\theta)/\lambda$, meaning the overall optimization process reduces to gradient descent with a particular learning rate.

To better understand the effect of the Tikhonov damping, note that the addition of a scalar multiple of the identity matrix to B has the effect of increasing each of the eigenvalues by precisely λ. This can be seen by noting that if $B = V\Sigma V^\top$ where $V = [v_1|v_2|\dots|v_n]$ are eigenvectors of B (which are orthonormal since B is symmetric), and $\Sigma \equiv \mathrm{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$ is the diagonal matrix of eigenvalues, then $\hat{B} = V\Sigma V^\top + \lambda I = V\Sigma V^\top + \lambda V V^\top = V(\Sigma + \lambda I)V^\top$. Thus the curvature associated with each eigenvector v_j in the damped matrix is given by $v_j^\top \hat{B} v_j = \lambda_j + \lambda$.

This modulation of the curvature has profound effect on the inverse of $\hat{B}$ since $\hat{B} = V^\top(\Sigma + \lambda I)^{-1}V$, where $(\Sigma + \lambda I)^{-1} = \mathrm{diag}\left((\lambda_1 + \lambda)^{-1}, (\lambda_2 + \lambda)^{-1}, \dots, (\lambda_n + \lambda)^{-1}\right)$ and this will be particularly significant for λ_j's that are small compared to λ, since $(\lambda_j + \lambda)^{-1}$ will generally be much smaller than λ_j^{-1} in such cases.

The effect on the minimizer δ^* of $\hat{M}$ can be seen by noting that

$$\delta^* = -\sum_j \frac{v_j^\top \nabla h(\theta_k)}{\lambda_j + \lambda} v_j$$

so the distance $v_j^\top\delta^*$ that δ^* moves θ in the direction v_j will be effectively multiplied by $\dfrac{\lambda_j}{\lambda_j + \lambda}$. Thus, Tikhonov damping should be appropriate when the quadratic model is most untrustworthy along directions of very low-curvature (along which δ^* will tend to travel very far in the absence of damping).

Picking a good value of λ is critical to the success of a Tikhonov damping approach. Too high, and the update

will resemble gradient descent with a very small learning rate and most of the power of 2nd-order optimization will be lost, with the low-curvature directions particularly affected. Conversely, if λ is too small, the resulting update may be very large (particular in directions of low curvature) which may cause an increase in h instead of a decrease. Unfortunately, determining a good value of λ is a nontrivial problem, which is sensitive to the overall scale of the objective function (i.e. using $\lambda = 1$ for h gives the same update as $\lambda = 2$ would for $2h$), and other more subtle properties of h, many of which will vary over the parameter space. It is in fact very rarely the case that a single value of λ will be appropriate at all θ's.

A method for dynamically adapting λ during optimization, which we have found works reasonably well in practice, will be discussed in Section 4.6.5.

Note that Tikhonov damping is the method used by LeCun et al. (1998), where the constant "μ" (which is *not* adapted) plays the role of λ.

It is worth noting that Vinyals and Povey (2012) have recently developed an alternative approach to Tikhonov damping, based on the idea of directly optimizing h over a K-dimensional Krylov basis generated by CG (or equivalently, a Lanczos iteration). Because the Krylov subspace generated by CG using $\hat{B} = B + \lambda I$ doesn't depend on λ (provided CG is initialized from 0), this method searches over a space of solutions that contain all those which would be found by optimizing a Tikhonov-damped $\hat{M}$ for some λ. Because of this, it can find solutions which will give more reduction in h than CG could obtain for *any* value of λ. The downsides of the approach are that the searching must be performed using a general-purpose 2nd-order optimizer like BFGS, which will require extra gradient and function evaluations, and that a basis for the entire Krylov subspace must be stored in memory (which may not always be practical when n is large), and finally that CG initializations cannot influence the construction of the Krylov subspace.

4.6.2 Problems with Tikhonov damping

For standard parameterizations of neural networks, where entries of the weight-matrices and bias vectors are precisely the entries of θ, and the regularization is the standard spherical ℓ_2 penalty $\beta\|\theta\|^2$, Tikhonov damping appears to be a natural choice, and works pretty well in practice. This is because for certain nicely behaved and also useful areas of the parameter space, the effective scale at which each parameter operates is (very) roughly equal. But imagine a simple reparameterization of an FNN so that at some particular layer j, θ parameterizes $10^4 W_j$ instead of W_j. Now the objective function is 10^4 times more sensitive than it was before to changes in the parameters associated with layer j, and imposing a Tikhonov damping penalty consisting of an equally weighted sum of squared changes over *all* entries of θ (given by $\frac{\lambda}{2}\|\delta\|^2 = \frac{\lambda}{2}\sum_{i=1}^{n}[\delta]_i^2$) no longer seems like a good idea.

For an even more extreme example, consider the case where we would like to constrain some of the weights of the network to be positive, and do this by a simple reparameterization via the $\exp$ function, so that for each component $[\theta]_i$ of θ corresponding to one of these weights w, we have $w = \exp([\theta]_i)$ instead of $w = [\theta]_i$. By applying the chain rule we see that in the new parameterization, the i-th component of the gradient, and the i-th row and column of the GGN matrix are both effectively multiplied by $\exp([\theta]_i)$, resulting in the update δ^* changing by a factor $\exp([\theta]_i)^{-1}$ in entry i.

More formally, if we define $C \in \mathbb{R}^{n \times n}$ to be Jacobian of the function ψ which maps the new parameters back to the default ones, then the gradient and GGN matrix in the new parameterization can be expressed in terms of

those from the original parameterization as $C^\top \nabla h$ and $C^\top G C$ respectively[4]. The optimal update thus becomes

$$\delta^* = (C^\top B C)^{-1} C^\top \nabla h = C^{-\top} B^{-1} \nabla h$$

For our particular example, C is a diagonal matrix satisfying $[C]_{i,i} = \exp([\theta]_i)$ for those entries of θ which are reparameterized, and $[C]_{i,i} = 1$ for those that are not.

Assuming that the original 2nd-order update was a reasonable one in the original parameterization, the 2nd-order update as computed in the new parameterization should also be reasonable (when taken in the new parameterization). In particular, a reparameterized weight w (and hence h) will become exponentially more sensitive to changes in $[\theta]_i$ as $[\theta]_i$ itself grows, and exponentially less sensitive as it shrinks, so an extra multiplicative factor of $\exp([\theta]_i)^{-1}$ compensates for this nicely. This should be contrasted with gradient descent, where the update will change in exactly the opposite way (being multiplied by $\exp([\theta]_i)$) thus further compounding the sensitivity problem.

Unfortunately, if we use standard Tikhonov damping directly in the reparameterized space, the assumption that all parameters operate at similar scales will be strongly violated, and we will lose the nice self-rescaling property of our update. For example, the curvature associated with $[\theta]_i$, which is equal to the curvature for w multiplied by $\exp([\theta]_i)^2$, may be completely overwhelmed by the addition of λ to the diagonal of G when $[\theta]_i$ is far below 0, resulting in an update which will fail to make a substantial change in $[\theta]_i$. Conversely, if $[\theta]_i$ is far above 0 then the Tikhonov damping contribution won't properly penalize large changes to $[\theta]_i$ which may lead to a very large and untrustworthy update.

We could hope that a sensible scheme for adapting λ would compensate by adjusting λ in proportion with $\exp([\theta]_i)$, but the issue is that there are many other components of θ, such as other $\exp$-reparameterized weights, and these may easily be much smaller or larger than $[\theta]_i$, and thus operate at vastly different scales. In practice, what will mostly likely happen is that any sensible scheme for dynamically adjusting λ will cause it to increase until it matches the scale of the largest of these reparameterized weights, resulting in updates which make virtually no changes to the other weights of the network.

In general, Tikhonov damping and any of the other quadratic penalty-based damping methods we will discuss in subsequent sections, can all be made arbitrarily restrictive through the choice of λ, thus constraining the optimization of $\hat{M}$ to a degree sufficient to ensure that the update will not leave the region where M is a sensible approximation. What differentiates good approaches from bad ones is how they weigh different directions relative to each other. Schemes that tend to assign more weight to directions associated with more serious violations of the approximation quality of M will get away with using smaller values of λ, thus allowing the optimization of $\hat{M}$ to be less constrained and thus produce larger and more useful updates to θ.

4.6.3 Scale-Sensitive damping

The scale sensitivity of the Tikhonov damping is similar to the scale sensitivity that plagues 1st-order methods, and is precisely the type of issue we would like to avoid by moving to 2nd-order methods. Tikhonov damping makes the same implicit assumptions about scale that are made by first-order methods: that the default norm $\|\cdot\|$ on $\mathbb{R}^n$ is a reasonable way to measure change in θ and a reasonable quantity to penalize when searching for a suitable update to θ. 1st-order methods can even be viewed as a special case of 2nd-order methods where the curvature term is given entirely by a Tikhonov-type damping penalty, so that $\hat{B} = \lambda I$ and $\delta^* = -\frac{1}{\lambda}\nabla h(\theta)$.

[4]Note that this result holds for smooth and invertible ψ, as long as we use the GGN matrix. If we use the Hessian, it holds only if ψ is affine.

One solution to this problem is to only use parameterizations which exhibit approximately uniform sensitivity properties, but this is limiting and it may be hard to tell at-a-glance if such a property holds for a particular network and associated parameterization.

A potential way to address this problem is to use a quadratic penalty function which depends on the current position in parameter space (θ_k) and is designed to better respect the local scale properties of h at θ_k. In particular, instead of adding the penalty term $\frac{\lambda}{2}\|\delta\|^2$ to M we may instead add $\frac{\lambda}{2}\|\delta\|^2_{D_k} = \frac{\lambda}{2}\delta^\top D_k \delta$, where D_k is some symmetric positive definite matrix that depends on θ_k. Such a term may provide a more meaningful measure of change in θ, by accounting for the sensitivity properties of h more precisely. We call the matrix D_k, the damping matrix, and will drop the subscript k for brevity. Scale sensitive damping is implemented in HF by working with a "damped" curvature matrix given $\hat{B} = B + \lambda D$, where the required matrix-vector products can be computed using as $\hat{B}v = Bv + \lambda Dv$, assuming an efficient algorithm for computing matrix-vector products with D.

A specific damping matrix which may work well in the case of the exp-reparameterized network discussed in the previous sub-section would be $D = C^\top C$ (for a definition of C, see the previous sub-section). With such a choice we find that the update δ^* produced by fully optimizing $\hat{M}$ is equal to C^{-1} times the update which would have been obtained with the original parameterization and standard Tikhonov damping with strength λ. Similarly to the undamped case, this is true because

$$(C^\top B C + \lambda C^\top C)^{-1} C^\top g = C^{-1}(B + \lambda I)^{-1} C^{-\top} C^\top g = C^{-1}(B + \lambda I)^{-1} g$$

It should also be noted that this choice of damping matrix corresponds to a penalty function $\frac{1}{2}\|\delta\|^2_{C^\top C}$ which is precisely the Gauss-Newton approximation of $\frac{\lambda}{2}\|\theta^\dagger\|^2 = \frac{\lambda}{2}\|\psi(\theta)\|^2$ w.r.t. to the new parameters θ, where $\theta^\dagger = \psi(\theta)$ are the default/original parameters. The interpretation is that we are penalizing change in the original parameters (which are assumed to have a very roughly uniform scale), despite performing optimization w.r.t. the new ones.

While we were able to design a sensible custom scheme in this example, exploiting the fact that the default parameterization of a neural network gives parameters which tend to operate at approximately similar scales (in most areas of the parameter space anyway), it would be nice to have a more generic and self-adaptive approach in the cases where we do not have such a property. One possible approach is to set D to be the diagonal matrix formed by taking the diagonal of B, a choice made in the classical Levenberg-Marquardt algorithm. With this choice, the update produced by fully optimizing the damped quadratic $\hat{M}$ will be invariant to diagonal linear reparameterizations of θ.

Another nice property of this choice of D is that it produces an update which is invariant to rescaling of h (i.e. optimizing βh instead of h for some $\beta > 0$). By contrast, a pure Tikhonov damping scheme would rely on the careful adjustment of λ to achieve such an invariance.

One obvious way to overcome the deficiencies of a damping approach based on a diagonal matrix would be to use a non-diagonal one, such as the original curvature matrix B itself. Such a choice for D produces updates that share all of the desirable invariance properties associated with a pure undamped Newton approach (assuming full optimization of $\hat{M}$), such as invariance to *arbitrary* linear reparameterizations, and rescalings of h. This is because with this choice, the damping-modified curvature matrix $\hat{B}$ is simply $(1 + \lambda)B$, and if we assume either full optimization of $\hat{M}$, or partial optimization via a run of CG initialized from 0, this type of damping has the effect of simply rescaling the update δ by a factor of $\frac{1}{1+\delta}$. In Section 4.6.8 we will discuss line-search methods which effectively accomplish the same type of rescaling.

Despite the nice scale invariance properties associated with these choices, there are good reasons not to use either of them in practice, or at least to use them only with certain modifications, and in conjunction with other

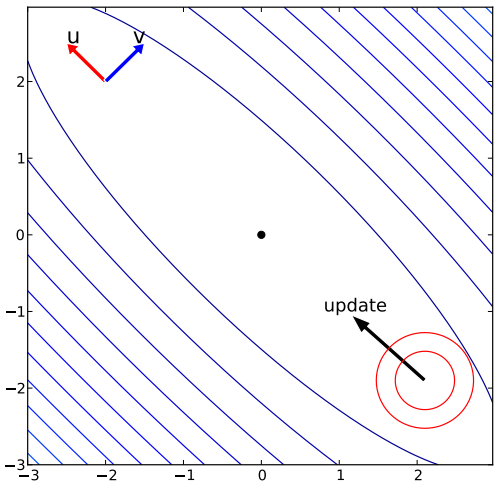

Figure 4.2: A 2D toy example of how using $D = \text{diag}(B)$ results in an overly restricted update. Let $u = [-1 \;\; 1]^\top$ and $v = [1 \;\; 1]^\top$, and let B be $uu^\top + avv^\top$ where a is large (e.g. 10^4, although we take $a = 15$ for display purposes). This matrix is full rank, and its diagonal entries are given by $[a+1 \;\; a+1]^\top$, representing the fact that the quadratic is highly sensitive to independent changes to the 2 parameters. The small circular region is where the update will be effectively restricted to when we make λ large enough.

approaches. While the Tikhonov approach arguably makes too few assumptions about the local properties of h, damping approaches based on $D = B$ or its diagonal may make too many. In particular, they make the same modeling assumptions as the original undamped quadratic approximation M itself. For example, B may not even be full-rank, and in such a situation it may be the case that M will predict unbounded improvement along some direction in B's nullspace, a problem which will not be handled by damping with $D = B$ for any value of λ, no matter how big. Even if B is full-rank, there may be directions of near-zero curvature which can cause a less extreme version of the same problem. Since the diagonal B will usually be full-rank even when B isn't, or just better conditioned in general, using it instead may give some limited immunity to these kinds of problems, but it is far from an ideal solution, as demonstrated in Figure 4.2.

In order to explain such degeneracies and understand why choices like $D = B$ can be bad, it is useful to more closely examine and critique our original choices for making them. The quadratic approximation breaks down due to higher-order effects (and even certain unmodeled 2nd-order effects in the case of the GGN matrix) and it is the goal of damping to help compensate for this. By taking $D = B$ we are penalizing directions according to their curvature, and so are in some sense assuming that the relative strength of the contribution to h from the high-order terms (and thus the untrustworthiness of M) along two different directions can be predicted reasonably well by looking at the ratio of their respective curvatures. And while there is a tendency for this to be true for certain objective functions, making this assumption too strongly may be dangerous.

Unfortunately, in the absence of any other information about the semi-local behavior of the function h, it may not always be clear what kind of assumption we should fall back on. To move towards the uniform scale assumption implied by the Tikhonov approach by choosing D to be some interpolation between the diagonal of B and a multiple of the identity (e.g. using the methods discussed in Section 4.9.2) seems like an arbitrary choice, since in general there may not be anything particularly special or natural about the default parameterization of h that we are given to work with. Despite this, such a strategy can be reasonably effective in some situations, and a

reasonable compromise between Tikhonov damping and damping with B.

A conceivably better approach would be to collect information about higher-order derivatives, or to use information collected and aggregated from previous iterations of the optimization process to build a simple model of the coarse geometric structure of h. Or perhaps some useful information could be gleaned from examining the structure of the computational graph of h. Unfortunately we are unaware of the existence of general methods for building D based on such ideas, and so this remains a direction for future research.

In the next section we discuss a method called "structural damping" which constructs D using knowledge of the particular structure of deep and recurrent neural networks, in order to construct damping matrices which may be better at selectively penalizing directions for the purposes of damping.

4.6.4 Structural Damping

Recurrent Neural Networks (RNNs) are known to be difficult to train with gradient descent, so it is conceivable that problematic variations in scale and curvature are responsible. Indeed, a direct application of the implementation of HF presented by Martens (2010) to RNNs can yield reasonable results, performing well on a family of synthetic pathological problems (Hochreiter and Schmidhuber, 1997; Martens and Sutskever, 2011) designed to have very long-range temporal dependencies of up to 100 time-steps. However Martens and Sutskever (2011) found that performance could be made substantially better and more robust using an idea called "structural damping".

Martens and Sutskever (2011) found that a basic Tikhonov damping approach performed poorly when applied to training RNNs. In particular, in order to avoid very large and untrustworthy update proposals, they found that λ needed to be very high, and this in turn would lead to much slower optimization. This need for a large λ can be explained by the extreme sensitivity of the RNN's long sequence of hidden states to changes in the parameters and in particular the hidden dynamics matrix $W_{1,1}^{(1)}$. While these sorts of problems exist with deep feedforward neural networks like the autoencoders considered in Hinton and Salakhutdinov (2006) and Martens (2010), the situation with RNNs is much more extreme, since they have many more effective "layers", and their parameters are applied repeatedly at every time-step and can thus have a dramatic effect on the entire hidden state sequence (Bengio et al., 1994; Hochreiter, 1991). Due to this extreme and highly non-linear sensitivity, local quadratic approximations are likely to be highly inaccurate in certain directions in parameter space, even over very small distances. A Tikhonov damping approach can only compensate for this by imposing a strict penalty against changes in all directions, since it lacks any mechanism to be more selective.

Structural damping addresses this problem by imposing a quadratic penalty not just to changes in parameters, but also to certain intermediate quantities that appear in the evaluation of h, such as the hidden state activities of an RNN. This allows us to be more selective in the way we penalize directions of change in parameter space, focusing on those that are more likely to lead to large changes in the hidden state sequence (which will mostly likely be a change that is not well modeled by M, due to the highly non-linear relationship of the hidden states to the parameters).

Speculatively, structural damping may have another more subtle benefit for RNN learning. It is known that good random initializations give rise to nontrivial hidden state dynamics that can carry useful information about the past inputs even before any learning has taken place (Jaeger and Haas, 2004; Martens and Sutskever, 2011). So if an RNN is initialized carefully to contain such random dynamics, the inclusion of structural damping may encourage the updates to preserve them at least until the hidden-to-output weights have had some time to be adapted to the point where the long-range information contained in the hidden activities actually gets used to inform future predictions. After such a point, a locally greedy optimizer like HF will have more obvious reasons

to preserve the dynamics.

To formalize structural damping, let $c(x, \theta)$ be some vector-valued function of θ consisting of intermediate quantities arising during the computation of $f(x, \theta)$. For example, $c(x, \theta)$ might contain the activities (the a_i's) or the unit inputs (the s_i's) of certain layers of hidden units in the network. We can prevent large changes in c by penalizing the distance between $c(x, \theta)$ and $c(x, \theta_k)$ according to the penalty function

$$K(\theta) = \frac{1}{|S|} \sum_{(x,y) \in S} d(c(x, \theta), c(x, \theta_k))$$

where $d(\cdot, \cdot)$ is a distance function or loss function such as the squared error or the cross-entropy[5].

Ideally, we would define the damped local objective as

$$M(\delta) + \mu K(\delta + \theta_k) + \frac{\lambda}{2} \delta^\top \delta$$

where μ is a strength constant similar to λ, which might be dynamically adjusted. But since we cannot minimize a non-quadratic objective with CG, we must resort to using a local quadratic approximation to $K(\theta)$.

As with the standard objective, we will use $\frac{1}{2} \delta^\top D_k \delta$ where D_k is the generalized Gauss-Newton matrix (taking d to be the "loss function" and c to be the network function) associated with penalty function $K(\theta)$ at $\delta = \theta - \theta_k = 0$. Note that the quadratic approximation to $K(\theta)$ does not have a linear term because $\delta = 0$ is a minimum of K.

Fortunately, it is straightforward to multiply by the generalized Gauss-Newton matrix of K using the techniques outlined in Section 3.4. Thus we could compute the products $G_k v$ and $\mu D_k v$ using two separate matrix-vector products, and add together the results, while approximately doubling the computational cost. In order to avoid this increase in cost, we can instead compute the sum $G_k v + \mu D_k v = (G_k + \mu D_k) v$ directly by exploiting the fact that $K(\delta)$ can be computed much more efficiently by reusing the $c(x, \theta_k + \delta)$ that gets computed as an intermediate quantity during the computation of $f(x, \theta_k + \delta)$.

One way to do this is to formally construct a new neural network whose output includes c as well as the usual output $f(x, \theta)$. In particular, we can define the new network function $f^\dagger(x, \theta)$ by

$$f^\dagger(x, \theta) = [f(x, \theta) \;\; c(x, \theta)]^\top$$

and the new loss function $L^\dagger$ by

$$L^\dagger(y, z) = L(y, f(x, \theta)) + \mu d(c(x, \theta), c(x, \theta_k))$$

It follows that the objective function $h^\dagger(\theta)$ corresponding to this new network satisfies $h^\dagger(\theta) = h(\theta) + \mu K(\theta)$.

It is not difficult to show that using these definitions yields a GGN matrix which will be given by $G_k + \mu D_k$. Thus, applying the standard techniques for computing matrix-vector products with the GGN to a computational graph/program which computes $h^\dagger(\theta)$ efficiently (i.e. by reusing the computations in $f(x, \theta)$ to compute $c(x, \theta)$) yields an efficient method for computing $(G_k + \mu D_k) v$, which will not be significantly more expensive than the basic method for computing matrix-vector products with just G_k).

[5]The cross-entropy is suitable when the hidden units use a logistic sigmoid nonlinearity

4.6.5 The Levenberg-Marquardt heuristic

For the penalty-based damping methods such as those described above to work well, λ (and perhaps also μ, as defined in the previous subsection) must be constantly adapted to keep up with the changing local curvature of h.

Fortunately, we have found that the well-known Levenberg-Marquardt (LM) heuristic, which is usually used in the context of the LM method (Moré, 1978), is effective at adapting λ in a sensible way even in the context of the truncated CG runs that are used in HF.

The key quantity behind the LM heuristic is the "reduction ratio", denoted by ρ, which is given by

$$\rho \equiv \frac{h(\theta_k + \delta_k) - h(\theta_k)}{M_k(\delta_k) - M_k(0)} \tag{4.9}$$

The reduction ratio measures the ratio of the reduction in the objective $h(\theta_k + \delta_k) - h(\theta_k)$ produced by the update δ_k, to the amount of reduction predicted by the quadratic model. When ρ is much smaller than 1, the quadratic model overestimates the amount of reduction and so λ should be increased, encouraging future updates to be smaller and more likely to be contained in the region where the quadratic model accurately predicts the reduction. In contrast, when ρ is close to 1, the quadratic approximation is likely to be fairly accurate near δ^*, and so we can afford to reduce λ, thus relaxing the constraints on δ_k and allowing for "larger" and more substantial updates.

The Levenberg-Marquardt heuristic is given by the rule

if $\rho > 3/4$ then $\lambda \leftarrow (2/3)\lambda$

if $\rho < 1/4$ then $\lambda \leftarrow (3/2)\lambda$

Although the constants in the above description of the LM are somewhat arbitrary, we found them to work well in our experiments.

While the definition of ρ in eqn. 4.9 uses the undamped quadratic in the denominator, the damped quadratic approximation $\hat{M}_k(\delta_k)$ can also be used, and in our experience will give similar results, favoring only slightly lower values of λ.

Because of this tendency for M to lose accuracy as CG iterates (see Subsection 4.6.7), the value of ρ tends to decrease as well (sometimes after an initial up-swing caused by using a non-zero initialization as in Section 4.8). If CG were to run to convergence, the Levenberg-Marquardt heuristic would work just as it does in the classical Levenberg-Marquardt algorithm, which is to say very effectively and with strong theoretical justifications. But the situation becomes more complicated when this heuristic is used in conjunction with updates produced by unconverged runs of CG, because the "optimal" value of λ, which the LM heuristic is trying to find, will be a function of how much progress CG tends to make when optimizing M.

Fortunately, as long as the local properties of h change slowly enough, terminating CG according to a fixed maximum number of steps should result in a relatively stable and well-chosen value of λ.

But unfortunately, well intentioned methods which, for example, attempt to be smarter and terminate CG based on the value of h, can run into problems caused by this dependency of the optimal λ on the performance of CG. In particular, this "smart" decision of when to stop CG will have an affect on ρ, which will affect the choice of λ via the LM heuristic, which will affect the damping and hence how the value of h evolves as CG iterates (at the next HF iteration), which will finally affect the decision of when to stop CG, bringing us full circle. It is this kind of feedback which may result in unexpected and undesirable behaviors when using the LM heuristic, such as λ and the number of the length of the CG runs both going to zero as HF iterates, or both quantities creeping upwards to inappropriately large values.

4.6.6 Trust-region methods

In contrast to damping methods that are based on penalty terms designed to encourage updates to be "smaller" according to some measure, trust region methods impose an explicit constraint on the optimization of the quadratic model M. Instead of performing unconstrained optimization on the (possibly damped) quadratic $\hat{M}$, a constrained optimization is performed over M. This is referred to as the trust-region sub-problem, and is defined as the minimization problem

$$\delta_R^* = \arg\min_{\delta:\delta\in R} M(\delta)$$

where $R \subseteq \mathbb{R}^n$ is some region localized around $\delta = 0$ called the "trust-region". Ideally, R has the property that M remains a reasonable approximation to h for any $\delta \in R$, without being overly restrictive. Or perhaps more weakly (and practically), that whatever update δ_R^* is produced by solving the trust-region sub-problem will produce a significant reduction in h. Commonly, R is chosen to be a ball of radius r centered at 0, although it could just as easily be an ellipsoid or something more exotic, as long as the required constrained optimization can be performed efficiently enough.

There is a formal result connecting trust region methods and penalty-based damping methods such as Tikhonov damping, which states that when R is an elliptical ball around 0 of radius r given $R = \{x : \|x\|_Q \leq r\}$ for some positive definite matrix Q, then for each quadratic function $M(\delta)$ there exists a λ such that

$$\arg\min_{\delta:\delta\in R} M(\delta) = \arg\min_{\delta} M(\delta) + \frac{\lambda}{2}\|\delta\|_Q^2$$

This result is valid for all quadratic functions M, even when B is indefinite, and can be proved using Lagrange multipliers (e.g. Nocedal and Wright, 2006).

Trust-region methods have some appeal over the previously discussed penalty-based damping methods, because it may be easier to reason intuitively, and perhaps also mathematically, about the effect of an explicit trust-region (with a particular radius r) on the update than a quadratic penalty. Indeed, the trust region R is invariant to changes in the scale of the objective[6], which may make it easier to tune, either manually or by some automatic method.

However, the trust region optimization problem is much more difficult than the unconstrained quadratic optimization of $\hat{M}$. It cannot be directly solved either by CG or by matrix inversion. Even in the case where a spherical trust-region with radius r is used, the previously discussed result is non-constructive and merely guarantees the existence of an appropriate λ that makes the exact minimizer of the Tikhonov damped $\hat{M}$ equal to the solution of the trust region sub-problem. Finding such a λ is a hard problem, and while there are algorithms that do this (Moré and Sorensen, 1983), they involve expensive operations such as full decompositions of the B matrix, or finding the solutions of multiple Tikhonov-damped quadratics of the form $\hat{M}$. When CG is used to perform partial optimization of the quadratic model, there are also good approximation algorithms (Gould et al., 1999) based on examining the tridiagonal decomposition of B (restricted to the Krylov subspace), but these require either storing a basis for the entire Krylov subspace (which may be impractical when n is large), or will require a separate run of CG once λ has been found via the tridiagonal decomposition, effectively doubling the amount of matrix-vector products that must be computed.

Even if we can easily solve (or partially optimize) the trust-region sub-problem, we are still left with the problem of adjusting r. And while it is likely that the "optimal" r will remain a more stable quantity than the

[6] If the objective is multiplied by 2, then λ also needs to be multiplied by 2 to achieve the same effect when using a penalty method. By contrast, the trust region R is unaffected by such a scale change.

"optimal" λ over the parameter space, it still needs to be adjusted using some heuristic. Indeed, one heuristic which is advocated for adjusting r (Nocedal and Wright, 2006) is precisely the Levenberg-Marquardt heuristic discussed in Section 4.6.5 which is already quite effective (in our experience) at adjusting λ directly. This leaves us with the question: why not just adjust λ directly and avoid the extra work required to compute λ by way of r?

One method which is effective at finding reasonable *approximate* solutions of the trust-region sub-problem, for the case where R is a ball of radius r, is to run CG (initialized at zero) until the norm of the solution exceeds r (i.e. the solution leaves the trust-region) and truncate CG at that iteration, with a possible modification to the α multiplier for the final conjugate direction to ensure a solution of norm exactly r. This is known as the Steihaug-Toint method (Toint, 1981; Steihaug, 1983), and it can be shown that, provided CG is initialized from a zero starting solution and no preconditioning is used, the norm of the solution will increase monotonically and that if CG is truncated in the manner described above, the resultant solution $\delta_k^\dagger$ will satisfy $M(\delta_k^\dagger) - M(0) \leq \frac{1}{2}(M(\delta_k^*) - M(0))$ where δ_k^* is the optimal solution to the trust-region sub-problem (Yuan, 2000). This seems like a good compromise, and it is more economical than approaches that try to solve the trust region problem exactly (or using better approximations, as discussed above). Unfortunately, the restriction that CG must be initialized from zero cannot be easily removed, and in our experience such initializations turn out to be very beneficial, as we will discuss in Section 4.8.

Another possible argument against using the Steihaug-Toint method is that it cannot be used with *preconditioned* CG. However, as long as we are willing to enforce an elliptical trust-region of the form $\{x : \|x\|_P < r\}$ where P is the preconditioning matrix, instead of a spherical one, the method still works and its theory remains valid. And depending on the situation, this kind of elliptical trust-region may actually be a very natural choice.

4.6.7 CG truncation as damping

Within HF, the main reason for terminating CG before it has converged is one of a practical nature, which is that the matrix-vector products are expensive and additional iterations will eventually provide diminishing returns as far as optimizing the quadratic model M. But there is another more subtle reason we may want to truncate early, which is that CG truncation may be viewed as a special type of damping which may be used in conjunction with (or as an alternative to) the various other damping methods discussed in this section.

As CG iterates, the accuracy of $M(\delta)$ as a model of $h(\theta_k + \delta)$ tends to go down, even while $h(\theta_k + \delta)$ may still be improving. Assuming CG is initialized from 0, one way to explain this is that the norm of δ increases monotonically with each CG iteration, and thus in later iterations will be more likely to leave the region where where M is a reasonable approximation of h. There is also theory which suggests that CG will converge to δ^* first along directions of mostly higher curvature, and only later along directions of mostly low curvature (see Section 4.7). While pursuing low curvature directions seems to be important for optimization of deep networks and RNNs, it also tends to be associated with large changes in θ which can lead to more serious breakdowns in the accuracy of the local quadratic model.

The Steihaug-Toint method described in Section 4.6.6 already makes use of this phenomenon and relies exclusively on truncating the solution early to enforce trust-region constraints. And it is well-known that truncating CG, which effectively limits the size of the Krylov subspace, has certain regularizing properties in the context of solving noisy and ill-posed systems of linear equations (Hansen and O'Leary, 1993).

Although it wasn't emphasized in the paper, Martens (2010) supplemented the progress-based CG termination criteria with a maximum limit of 250 steps. In practice, we have found that this maximum is consistently reached after the first 100 (or so) iterations of HF when the approach is applied to problems like the "CURVES" auto-encoder from Hinton and Salakhutdinov (2006), and that it plays an important role which is not limited to saving

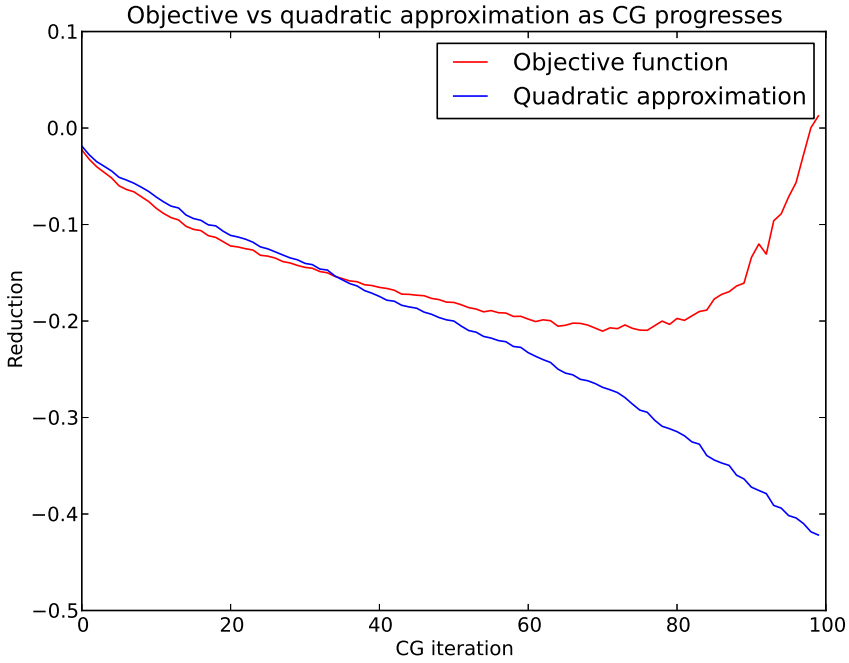

Figure 4.3: A plot of the objective function (red) versus the quadratic approximation (blue) as a function of the number of CG steps. This was selected as a typical example of a single run of CG performed by HF on a typical deep auto-encoder training task.

computation time. Instead, we can observe that the improvement in the value of the objective h is non-monotonic in the number of CG steps, and may peak long before condition 4.2 is satisfied (see Figure 4.3).

Martens (2010) proposed "CG-backtracking" as a method to take advantage of this tendency for earlier iterates to be more favorable as updates, by selecting the "best" iterate among some manageable subset, as measured by the associated reduction in h. One possible approach which is reasonably efficient is to compute the objective h only on a small subset of the current mini-batch or training set, and only at every multiple of c steps for some fixed c, or for every step whose number is equal to some power of ν (rounding to the nearest integer) for some fixed ν, and then terminate once it appears that there will be no further possible improvement in the objective.

An important thing to keep in mind is that the damping effect of CG truncation will depend on both the preconditioning scheme used within CG (as discussed in Section 4.9), and on the particular manner in which CG is initialized (as discussed in Section 4.8). In the extreme case where $P = B$, the eigenvalues will all be equal and CG will converge in one step, rendering CG truncation trivial/useless. And in general, for "strong" preconditioners that let CG converge faster it can be argued that truncation at a particular iteration i will have a smaller effect than with "weak" preconditioners. But this perspective oversimplifies a very complex issue.

Preconditioning is effectively reparameterizing the quadratic optimization problem, which has the effect of creating new eigenvectors with new eigenvalues and of changing the Krylov subspace (as discussed further in Section 4.9.1). This in turn affects the "order" in which CG tends to prioritize convergence along different directions (see Section 4.7). Thus, when CG is terminated long before convergence, preconditioning will have an important effect on the nature of the implicit damping and thus on the quality of the update.

From the perspective of the Steihaug-Toint method and trust-regions, CG preconditioning can be thought of as determining the shape of the trust-region that is being implicitly enforced through the use of truncation. In

particular, the trust region will be given by $R = \{x : \|x\|_P \leq r\}$, for some r, where P is the preconditioning matrix. Similarly, initializing CG from some non-zero point x_0 can be thought of as shifting the center of said trust-region away from 0. In both cases, the guarantee remains true that the solution found by truncated CG will be at least half as good (in terms of the value of M), subject to the trust-region radius implied by the current iterate, as long as we modify definition of the trust region appropriately.

4.6.8 Line searching

The most basic form of damping, which is present in almost every 2nd-order optimization algorithm, is standard line searching applied to the update proposal δ_k. In particular, we select a scalar α_k to multiply the update δ_k before adding it to θ_k so that $\theta_{k+1} = \theta_k + \alpha\delta_k$. Usually α is set to 1 as long as certain conditions hold (see Nocedal and Wright, 2006), and decreased only as necessary until they do. Doing this guarantees that certain local convergence theorems apply.

Provided that δ_k is a descent direction ($\delta_k^\top \nabla h(\theta_k) < 0$) we know that for a sufficiently small (but non-zero) value of α, we will have

$$h(\theta_k + \alpha\delta_k) = h(\theta_{k+1}) < h(\theta_k) \tag{4.10}$$

δ_k will be descent direction as long as $\hat{B}_k$ is positive definite, ∇h is computed on the entire training set, and $\hat{M}$ is optimized either exactly, or partially with CG (provided that we achieve $M(\delta) < 0$). Thus, in many practical scenarios, a line search will ensure that the update results in a decrease in h, although it may be very small. And if we are content with the weaker condition that only the terms of h corresponding to the mini-batch used to estimate $\nabla h(\theta_k)$ must decrease, then we can drop the requirement that ∇h be computed using the whole training set.

One easy way to implement line searching is via the back-tracking approach, which starts at $\alpha = 1$ and repeatedly multiplies this by some constant $\beta \in (0, 1)$ until the "sufficient-decrease condition" applies. This is given by

$$h(\theta_k + \alpha\delta_k) \leq h(\theta_k) + c\alpha\nabla h(\theta_k)^\top \delta_k$$

where c is some small constant like 10^{-2}. It can be shown that this following approach will produce an update which has strong convergence guarantees [7].

Unlike 1st-order methods where the total number of updates to θ can often be on the order of $10^5 - 10^6$ for a large neural network, powerful approximate Newton methods like HF may only require on the order of $10^2 - 10^3$, so the expense of a line-search is much easier to justify.

Note also that it is also possible to view line searching as a special type of a penalty based damping method, where we use a penalty term as in Section 4.6.3 with $D_k = (1/\alpha - 1)B$. In other words, we simply increase the scale of curvature matrix B by $1/\alpha$. This interpretation is valid as long as we solve $\hat{M}$ exactly (or partially by CG as long as it is initialized from zero).

The line search is best thought of as a last line of defense to compensate for occasional and temporary maladjustment of the various non-static parameters of the other damping methods, such as λ or r, and not as a replacement for these methods. If the line search becomes very active (i.e. it very often chooses an α less than 1) there are two probable causes, which should be addressed directly instead of with the line-search. The first is poorly designed/inappropriate damping methods and/or poor heuristics for adjusting their non-static meta-parameters.

[7] But note that other possible ways of choosing an α that satisfies this condition may make α too small.

The second probable cause is that the data used to estimate the curvature and/or gradient is insufficient and the update has effectively "overfitted" the current mini-batch.

The argument for not relying on line searches to fix whatever problems might exist with the updates produced by optimizing M is that they work by rescaling each eigen-component (or conjugate direction) of the update *equally* by the multiplicative factor α, which is a very limited and inflexible approach. It turns out that in many practical situations, the type of scaling modification performed by a penalty method like Tikhonov damping is highly preferable, such as when B is very ill-conditioned. In such a situation, minimizing M results in an update proposal δ_k which is scaled much too strongly in certain, possibly spurious, low-curvature eigen-directions, by a factor of possibly 10^3 or more, and the line-search will have to divide *all components* by this factor in order to make the update viable, which results in an extremely small update. Meanwhile, a Tikhonov approach, because it effectively modifies the curvatures by the additive constant λ (as discussed in Section 4.6.1) can easily suppress these very low-curvature directions while leaving the higher curvature directions relatively untouched, which makes the update much bigger and more useful as a result.

4.7 Convergence of CG

In this section we will examine the theoretical convergence properties of CG and provide justifications for various statements regarding its convergence made throughout this report, such as how δ converges along different "directions" in parameter space, and how CG prioritizes these directions according to their "associated curvature". This analysis has particularly important implications for preconditioning (see Section 4.9) and CG truncation damping (see Section 4.6.7).

Before we begin it should be noted that due to inexact computer arithmetic, the conclusions of this analysis (which implicitly assume exact arithmetic) will only hold approximately in practice. Indeed, CG, unlike many other numerical algorithms, is highly sensitive to numerical issues and after only $5 - 10$ iterations on a high-precision machine will produce iterates that can differ significantly from those which would be produced by a hypothetical implementation using exact arithmetic.

We will analyze CG applied to a general quadratic objective of the form $q(x) = \frac{1}{2}x^\top Ax - b^\top x$ for a symmetric positive definite matrix $A \in \mathbb{R}^{n\times n}$, and a vector $b \in \mathbb{R}^n$. This can be related to HF by taking $A = B$ (or $A = \hat{B}$ in the case of a damped quadratic approximation) and $b = -\nabla h$.

Note that x as defined here is an n-dimensional vector and should not be confused with its use elsewhere in this chapter as the input to the network.

Let $\{\lambda_j\}_{j=1}^n$ be the eigenvalues of A and $\{v_j\}_{j=1}^n$ the corresponding (unit) eigenvectors, x_0 be the initialization of CG, and x^* the minimizer of q. Since the v_j's are an orthonormal basis for $\mathbb{R}^n$ (because A is symmetric and invertible) for any x we can express $x_0 - x^*$ in terms of the v_j's, giving $x_0 - x^* = \sum_{j=1}^n \xi_j v_j$ for $\xi_j = v_j^\top (x_0 - x^*)$.

It can be shown (Nocedal and Wright, 2006; Shewchuk, 1994) that

$$\|x_i - x^*\|_A^2 = \min_p \sum_{j=1}^n \lambda_j p(\lambda_j)^2 \xi_j^2 \tag{4.11}$$

where $\|z\|_A^2 \equiv \frac{1}{2}z^\top Az$ and where the minimum is taken over all polynomials of degree i with constant term 1. This result can be used to prove various convergence theorems for CG (Nocedal and Wright, 2006). For example, CG will always converge to x^* after a number of iterations less than or equal to the number m of distinct eigenvalues of A, since it is easy to design a polynomial of degree m with constant term 1 that satisfies $p(\lambda_j) = 0$

for all j.

To gain more insight into eqn. 4.11 we will re-derive and re-express it in a way that implies an intuitive interpretation for each term. It is easy to show that

$$q(z + x_0) - q(x_0) = \frac{1}{2} z^\top A z + r_0^\top z$$

where $r_0 = Ax_0 - b$ (i.e. the initial residual). And so $q(z + x_0) - q(x_0)$ is a quadratic in z with constant term equal to 0, and linear term r_0.

Defining $\eta_j = r_0^\top v_j$ to be the size of eigenvector v_j in direction $r_0 = Ax_0 - b$ (which is the initial residual), and observing that $v_j^\top A v_j = \lambda_j v_j^\top v_j = \lambda_j$, we have for any $\alpha \in \mathbb{R}$

$$q(\alpha v_j + x_0) - q(x_0) = \frac{1}{2}\alpha^2 v_j^\top A v_j + \alpha r_0^\top v_j = \frac{1}{2}\alpha^2 \lambda_j + \alpha \eta_j$$

Note that the v_j's are also mutually conjugate (which follows from them being both orthonormal and eigenvectors) and that $q(z+x_0)-q(x_0)$ is quadratic in z with constant term 0. Thus we have that for any two conjugate vectors u and w

$$q(u + w + x_0) - q(x_0) = (q(u + x_0) - q(x_0)) + (q(w + x_0) - q(x_0))$$

which is straightforward to show.

Since the v_j's are an orthonormal basis for $\mathbb{R}^n$ (because A is symmetric and invertible), we can express $x - x_0$ (for any x) in terms of the v_j's, giving $x = x_0 + \sum_j \beta_j v_j$ where $\beta_j = v_j^\top (x - x_0)$. Thus we have

$$\begin{aligned} q(x) - q(x_0) = q((x - x_0) + x_0) - q(x_0) &= \sum_{j=1}^{n} (q(\beta_j v_j + x_0) - q(x_0)) \\ &= \sum_{j=1}^{n} \left(\frac{1}{2}\beta_j^2 \lambda_j + \beta_j \eta_j \right) \end{aligned}$$

What this says is that the sizes of the contribution of each eigenvector/eigen-direction to x (given by the β_j's), have an independent influence on the value of $q(x) - q(x_0)$, and so we can meaningfully talk about how each one of them independently "converges" as CG iterates.

In a sense, each β_j is being optimized by CG, with the ultimate goal of minimizing the corresponding 1-D quadratic $q(\beta_j v_j + x_0) - q(x_0)$, whose minimizer is $\beta_j^* = -\dfrac{\eta_j}{\lambda_j} = v_j^\top (x^* - x_0)$ with associated minimum value

$$q(\beta_j^* v_j + x_0) - q(x_0) = -\frac{1}{2}\frac{\eta_j^2}{\lambda_j} = -\omega_j$$

where we define $\omega_j \equiv \frac{1}{2}\frac{\eta_j^2}{\lambda_j}$. The difference between the current value of $q(\beta_j v_j + x_0)$ and its minimum has a particularly nice form:

$$q(\beta_j v_j + x_0) - q(\beta_j^* v_j + x_0) = \frac{1}{2}\beta_j^2 \lambda_j + \beta_j \eta_j + \omega_j = \omega_j \left(\frac{\lambda_j}{\eta_j}\beta_j + 1 \right)^2 = \omega_j \left(1 - \frac{\beta_j}{\beta_j^*} \right)^2$$

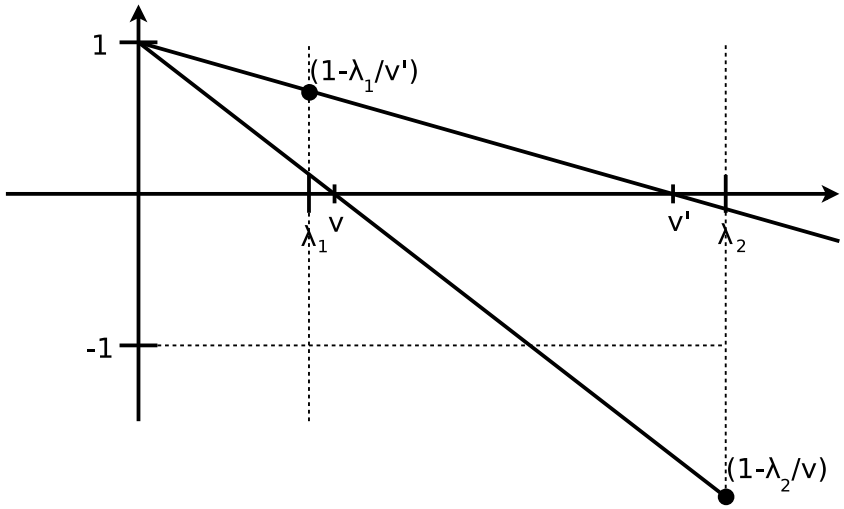

Figure 4.4: This figure demonstrates geometrically how the contribution to the polynomial $p(z)$ of an additional root ν or ν' in the vicinity of a small eigenvalue λ_1 or a large eigenvalue λ_2 (resp.) affects the loss term associated with the other eigenvalue. In particular, the distance of the lines above or below the horizontal axis is equal to the factor whose square effectively multiplies the loss term associated with the given eigenvalue.

Now suppose that $x - x_0 \in K_i(A, r_0)$ so that there exists some $(i-1)$-degree polynomial s s.t. $x - x_0 = s(A)r_0$ and note that $s(A)v_j = s(\lambda_j)v_j$. Then,

$$\beta_j = v_j^\top (x - x_0) = v_j^\top s(A) r_0 = (s(A)v_j)^\top r_0 = (s(\lambda_j)v_j)^\top r_0 = s(\lambda_j)\eta_j$$

so that

$$q(\beta_j v_j + x_0) - q(\beta_j^* v_j + x_0) = \omega_j \left(\frac{\lambda_j}{\eta_j} s(\lambda_j)\eta_j + 1 \right)^2 = \omega_j (\lambda_j s(\lambda_j) + 1)^2 = \omega_j p(\lambda_j)^2$$

where we define $p(z) = zs(z) + 1$ (which is a general polynomial of degree i with constant coefficient 1).

Summarizing, we have

$$\begin{aligned} q(x) - q(x^*) = (q(x) - q(x_0)) - (q(x^*) - q(x_0)) &= \sum_{j=1}^{n} (q(\beta_j v_j + x_0) - q(\beta_j^* v_j + x_0)) \\ &= \sum_{j=1}^{n} \omega_j p(\lambda_j)^2 \end{aligned} \tag{4.12}$$

We now apply this results to CG. We know that after i steps, CG applied to q with initialization x_0, finds the iterate x_i which minimizes $q(x_i)$ subject to restriction $x_i - x_0 \in K_i(A, r_0)$. This is equivalent to minimizing over all possible polynomials p with the requirement that p has degree i with constant coefficient 1, or in other words

$$q(x_i) - q(x^*) = \min_p \sum_{j=1}^{n} \omega_j p(\lambda_j)^2 \tag{4.13}$$

Thus we see that CG is effectively picking a polynomial p to minimize the weighted sum of p^2 evaluated at the different eigenvalues.

As mentioned before, there are various results that make use of expressions similar to this one in order to argue about how quickly CG can make progress optimizing q based on the distribution of the eigenvalues of A. One particularly interesting result states that if the eigenvalues cluster into m groups, then since we can easily design a degree-m polynomial p which is relatively small in the vicinity of each cluster by placing a root of p at each cluster center, the error will be quite low by the m-th iteration (Nocedal and Wright, 2006).

However, the particular form of eqn. 4.13 and its derivation allow us to paint a more intuitive picture of how CG operates. Each of the terms in the sum 4.12 correspond to a direction-restricted objective $q(\beta_j v_j + x_0) - q(\beta_j^* v_j + x_0) = \omega_j p(\lambda_j)^2$ which are indirectly optimized by CG w.r.t. the β_j's. The size each of these "loss" terms negatively correlates with how much progress CG has made in optimizing x along the corresponding eigen-directions, and by examining the form of these terms, we can talk about how CG will "prioritize" these different terms (and hence the optimization of their associated eigen-directions) through its choice of an optimal polynomial p.

Firstly, consider the "weights" $\omega_j = -(q(\beta_j^* v_j + x_0) - q(x_0)) = \frac{1}{2}\frac{\eta_j^2}{\lambda_j}$, which measure the total decrease in q that can be obtained by fully optimizing along the associated direction v_j. Their effect is thus to shift the focus of CG towards those eigen-directions which will give the most reduction. They are inversely proportional to the curvature λ_j, and proportional to $\eta_j^2 = (v_j^\top r_0)^2$, which is the square of the size of the contribution of the eigen-direction within the initial residual (which in HF will be the gradient of h when $x_0 = 0$). This makes the ω_j's "scale-invariant", in the sense that any linear reparameterization which preserves the eigenvectors of A, while possibly rescaling the eigenvalues, will have no effect on the ω_j's.

Secondly, we note the effect of the size of the λ_j's, or in other words, the curvatures associated with v_j's. If it weren't for the requirement that p must have a constant term of 1, the λ_j's probably wouldn't have any influence on CG's prioritizing of directions (beyond how they modulate the weights ω_j). But note that general polynomials of degree i with constant coefficient 1 must have the form

$$p(z) = \prod_{k=1}^{i} \left(1 - \frac{z}{\nu_k}\right)$$

for $\nu_k \in \mathbb{C}$. We will argue by illustrative example that this fact implies that CG will favor high-curvature directions, everything else being equal. Suppose there are two tight clusters of eigenvalues of A, a low-magnitude one located close to zero and a large-magnitude one located further away. Suppose also that they have equal total loss as measured by the sum of the associated $\omega_j p(\lambda_j)^2$'s (for the current p). Placing an additional root ν_k close to the large-magnitude cluster will greatly reduce the associated $\omega_j p(\lambda_j)^2$ loss terms in that cluster, by effectively multiplying each by $\left(1 - \frac{\lambda_j}{\nu_k}\right)^2$ which will be small due to the closeness of ν_k to each λ_j. Meanwhile, for the λ_j's in the small-magnitude cluster, the associated loss terms will be multiplied by $\left(1 - \frac{\lambda_j}{\nu_k}\right)^2$ which won't be greater than 1 since $0 < \lambda_j < \nu_k$, implying that these loss terms will not increase (in fact, they will very slightly decrease).

Now contrast this with what would happen if we placed a root ν_k close to the small magnitude cluster. As before, the loss terms associated with that cluster will be greatly reduced. However, because $\lambda_j \gg \nu_k$ for λ_j's in the large-magnitude cluster, we will have $\left(1 - \frac{\lambda_j}{\nu_k}\right)^2 \gg 1$, and so the associated loss terms will greatly increase, possibly even resulting in a net increase in q. Thus CG, being optimal, will place the root near to the large-magnitude cluster in this situation, versus the small magnitude-one, despite convergence of either one yielding the same improvement in q.

4.8 Initializing CG

As in the previous section we will use the generic notation $q(x) = \frac{1}{2}x^\top Ax - b^\top x$ to refer to the quadratic objective being optimized by CG.

A useful property of CG is that it is able to make use of arbitrary initial guesses x_0 for x. This choice can have a strong effect on its performance, which is not surprising since x_i depends strongly on the Krylov subspace $K_i(A, r_0)$ (where $r_0 = Ax_0 - b$), which in turn depends strongly on x_0. From the perspective of the previous section, an initial x_0 may be "more converged" than $x = 0$ along some eigen-directions and less converged along others, thus affecting the corresponding weights ω_j, which measure the total reduction that can be obtained by fully optimizing eigen-direction v_j (versus leaving it as it is in x_0). This "redistribution" of weights caused by taking a different x_0 may make the quadratic optimization easier or harder for CG to optimize.

Since the local geometry of the error surface of h (and hence the local damped quadratic model $q = \hat{M}$) changes relatively slowly between updates (at least along some eigen-directions), this suggests using the previous update δ_{k-1} as the starting solution x_0 for CG, as was done by Martens (2010).

In practice, this choice can result in an initial value of q which is higher than zero, and thus seemingly worse than just using $x_0 = 0$, which satisfies $q(x_0) = 0$. x_0 may not even be a descent direction, implying that $q(\epsilon x_0) > 0$ for all $\epsilon > 0$. But these objections are based on the naive notion that the value of q tells us everything there is to know about the quality of potential initialization. What we observe in practice is that while CG runs initialized with $x_0 = \delta_{k-1}$ "start slow" (as measured by the value of $q(x)$), they eventually catch up and then surpass runs started from $x_0 = 0$.

To make sense of this finding, we first note it is easy to design initializations which will have arbitrarily high values of q, but which require only one CG step to reach the minimum. To do this, we simply take the minimizer of q and add a large multiple of one of the eigenvectors of A to it. This corresponds to the situation where only one eigenvalue λ_j has non-zero weight ω_j, so that to make $q(x) - q(x^*) = 0$ CG can simply select the degree-1 polynomial which places a root at λ_j.

More generally, x_0 may be more converged than 0 along eigen-directions which are more numerous or which have eigenvalues that are small and spread-out, while being less converged than 0 only along eigen-directions which are fewer in number, or which have eigenvalues that are large and more tightly clustered. If the latter group has a larger total weight (given by the sum of the ω_j's as defined in the previous section) this will cause $q(x_0) > 0$. But since the latter directions will be easier for CG to optimize than the former, this implies that the given x_0 will still be a highly preferable initialization over "safer" choice of 0, as long as CG is given enough iterations to properly optimize the latter group of badly initialized but "easy-to-fix" eigen-directions.

We surmise that the choice $x_0 = \delta_{k-1}$ used in HF fits into the situation described above, where the former group of eigen-directions correspond to the slowly optimized low-curvature directions that tend to remain descent-directions across many HF iterations. Consistent with this theory, is our observation that the number of CG steps required to achieve $q(x) < 0$ from the initialization $x_0 = \delta_{k-1}$ tends to grow linearly with the number of CG steps used at the previous HF iteration[8].

Analogously to how the current update vector is "decayed" by a scalar constant when using gradient descent with momentum, we have found that it is helpful to slightly decay the initialization, taking $x_0 = \zeta\delta_{k-1}$ for some constant $0 \leq \zeta \leq 1$, such as 0.95.

Choosing this decay factor for HF carefully is not nearly as important as it can be for momentum methods. This is because while momentum methods modify their current update vectors by a single gradient-descent step,

[8]This is, incidentally, one reason why it is good to use a fixed maximum number of CG iterations at each HF step

HF uses an entire run of CG, which can make much more significant changes. This allows HF to more easily scale back x_0 along eigen-directions which are no longer appropriate to follow from the current θ.

Our experience suggests that properly tuning the decay constant becomes more important as aggressive CG truncation, and/or other factors like weak preconditioning, limit CG's ability either to modify the update from its initial value x_0, or to make good progress along important low-curvature directions. While the former problem calls for lowering ζ, the latter calls for raising it. The optimal value will likely depend on the amount of truncation, the type of preconditioning, and the local geometry of the objective being optimized. $\zeta = 0.95$ seems to be a good default value, but it may help to reduce it when using an approach which truncates CG very early. It may also be beneficial to increase it in the later stages of optimization where CG struggles much harder to optimize q along low curvature directions (although in very stochastic optimizations this might not be a good idea).

4.9 Preconditioning

As powerful as CG is, there are quadratics optimization problems which can be easily solved using more direct methods, which CG may struggle with. For example, if the curvature matrix is diagonal, CG will in general require i iterations to converge (where i is the number of distinct values on the diagonal). Meanwhile, we could easily solve the entire system by straightforward inversion of the diagonal curvature matrix in time $\mathcal{O}(n)$. CG is, in a sense, unaware of this special structure and unable to exploit it.

While the curvature matrix will in general not be diagonal or have any other special form that makes it easy to invert, there may nevertheless be cheap operations which can exploit information about the course structure of the curvature matrix to do some of the work in optimizing q, reducing the burden on CG.

In the context of HF, preconditioning refers to the reparameterization of $\hat{M}$ according to some linear transformation relatively easy to invert, with the idea that CG will make more rapid progress per iteration optimizing within the new parameterization.

Formally, given some invertible transformation defined by a matrix C, we transform the quadratic objective $\hat{M}(\delta)$ by a reparameterization defined by $\delta = C^{-1}\gamma$ and optimize w.r.t. γ instead of δ.

$$\hat{M}(C^{-1}\gamma) = \frac{1}{2}\gamma^\top C^{-\top}\hat{B}C^{-1}\gamma + \nabla h^\top C^{-1}\gamma$$

Applying preconditioning to CG is very easy and amounts to computing transformed residual vectors y_i at each iteration, by solving $Py_i = r_i$, where $P = C^\top C$ (see Algorithm 2). This can be accomplished, say, by multiplication of r_i by P^{-1}, which for many common choices of P (such as diagonal approximations of $\hat{B}$) is a cheap operation.

Preconditioning can be applied to other optimization methods, such as gradient descent, where it corresponds to a non-static linear reparameterization of the objective h that typically varies with each iteration, and amounts simply to multiplication of the gradient update by P^{-1}. In fact, one way to view 2nd-order optimization is as a particular non-static preconditioning approach for gradient descent, where P is given by the curvature matrix B (or some approximation or Krylov subspace restriction of it).

4.9.1 The effects of preconditioning

In Section 4.7, we saw how the eigenvalues of the curvature matrix and the corresponding sizes of the contributions to the initial residual of each eigen-direction effectively determine the convergence characteristics of CG, in terms of the "order" in which the eigen-directions tend to converge, and how quickly. It was found that each

eigen-direction has an effective "weight" ω_j (corresponding to the total decrease in q which can be obtained by completely optimizing it), and that CG prioritizes convergence along the eigen-directions both according to their weights and their associated curvature/eigenvalue, preferring larger values of both. Moreover, CG works better when the eigenvalues are clustered together into a few groups, as it can essentially make progress on the associated eigen-directions of a particular cluster all at once.

Thus to understand how a potential preconditioning scheme affects the convergence of CG we can look at the eigen-distribution of the transformed curvature matrix $C^{-\top}\hat{B}C^{-1}$, and the associated weights, which depend on the transformed initial residual $C^{-\top}(\hat{B}x_0 - \nabla h)$. Choices for C (or equivalently P) which yield tight clusters of eigenvalues should lead to overall faster convergence, at least along the directions which are contained in such clusters.

But note that since the eigenvalue distribution will affect the order in which various directions converge, preconditioning will interact in a non-trivial way with CG truncation damping (see Section 4.6.7). For example, preconditioned CG may optimize certain directions which would otherwise never be touched by CG in the original parameterization, such as those with very small eigenvalues. This is a potential problem, since it is our experience that certain directions of very low curvature in neural network objective functions (in the default parameterization) tend to be highly non-trustworthy. In particular, they often tend to correspond to degeneracies in the quadratic model, such as those introduced by using different sets of data to compute the gradient and curvature-matrix vector products (see Section 4.10.1), or to directions which yield small reductions in q for the current mini-batch but large increases on other training data (an issue called "mini-batch overfitting", which is discussed in Section 4.10.2).

4.9.2 Designing a good preconditioner

Designing a good preconditioner is an application specific art, especially for HF, and it is unlikely that any one preconditioning scheme will be the best in all situations. Typically, there is a trade-off between the computational cost associated with a preconditioner and its effectiveness.

While the previous section describes how a preconditioner can help in theory, in practice it is not obvious how to design one based directly on insights about eigen-directions and their prioritization.

One approach, which is popular and often very effective in various domains where CG is used, is to design P to be some kind of easily inverted approximation of the curvature matrix (in our case, $\hat{B}$). While the ultimate purpose of preconditioning is to help CG optimize more effectively, which may conceivably be accomplished by less obvious choices for P, approximating $\hat{B}$ may be an easier goal to approach directly. Justifying this idea is the fact that when $P = \hat{B}$, the preconditioned matrix is I, so CG will converge in one step.

Adopting the perspective that P should approximate $\hat{B}$, the task of designing a good preconditioner becomes one of balancing approximation quality with practical concerns, such as the cost of multiplying by P^{-1}.

Of course, "approximation quality" is a problematic concept, since the various ways we might want to define it precisely, such as via various matrix norms, may not correlate well with the effectiveness of P as a preconditioner. Indeed, CG is invariant to the overall scale of the preconditioner, and so while $\beta\hat{B}$ would be an optimal preconditioner for any $\beta > 0$, it could be considered an arbitrarily poor approximation to $\hat{B}$ as β grows, depending on how we measure this.

Diagonal P's are a very convenient choice due to the many nice properties they naturally possess, such as being full rank, easy to invert, and easy to store. They also tend to be quite effective for optimizing deep feed-forward neural networks, due to how the scale of the gradient and curvature of the objective h with respect to the hidden activities grows or shrinks exponentially as we proceed backwards through the layers (Bengio et al.,

1994; Hochreiter, 1991). Without compensating for this with diagonal preconditioning, the eigenvalues of the effective curvature matrix will likely be much more "spread out" and thus harder for CG to deal with. By contrast, RNN optimization does not seem to benefit as much from diagonal preconditioning, as reported by Martens and Sutskever (2011). Despite how RNNs can possess per-timestep scale variations analogous to the per-layer scale variations sometimes seen with feedforward networks, these won't manifest as differences in scales between any particular parameters (i.e. diagonal scale differences), due to the way each parameter is used at *every* time-step.

Many obvious ways of constructing non-diagonal preconditioners end up resulting in P's which are expensive and cumbersome to use when n is large. For example, if P or P^{-1} is the sum of a k-rank matrix and a diagonal, it will require $\mathcal{O}((k+1)n)$ storage, which for very large n will be a problem (unless k is very small).

A well-designed diagonal preconditioner P should represent a conservative estimate of the overall scale of each parameter, and while the diagonal of the curvature matrix is a natural choice in many situations, such as when the curvature matrix is diagonally dominant, it is seriously flawed for curvature matrices with a strong non-diagonal component. Nonetheless, building a diagonal preconditioner based on $d = \operatorname{diag}(\hat{B})$ (or an approximation of this) is a sensible idea, and forms the basis of the approaches taken by Martens (2010) and Chapelle and Erhan (2011). However, it may be beneficial, as Martens (2010) found, not to use d directly, but to choose P to be somewhere between $\operatorname{diag}(d)$ and a scalar multiple of the identity matrix. This has the effect of making it more gentle and conservative, and it works considerably better in practice. One way to accomplish this is by raising each entry of d (or equivalently, the whole matrix P) to some power $0 < \xi < 1$, which will make P tend to the identity as ξ approaches 0.

In situations where diagonal damping penalty terms like the Tikhonov term are weak or absent, it may also be beneficial to include an additional additive constant κ, which also has the effective of making P tend to a scalar multiple of the identity as κ grows so that we have

$$P = (\operatorname{diag}(d) + \kappa I)^{\xi}$$

If there is information available about the coarse relative scale of the parameters, in the form of some vector $s \in \mathbb{R}^n$, such as the reparameterized neural network example discussed in Section 4.6.2, it may better to use $\kappa \operatorname{diag}(s)$ instead of κI.

It is important to emphasize that d should approximate $\operatorname{diag}(\hat{B})$ and not $\operatorname{diag}(B)$, since it is the former curvature matrix which is used in the quadratic which CG actually optimizes. When D is a diagonal matrix, one should take

$$d = \operatorname{diag}(B) + \lambda D$$

where the second term (λD) can be computed independently and exactly (and not via the methods for approximating $\operatorname{diag}(B)$ which we will discuss next). Meanwhile, if the damping matrix D is non-diagonal, then one should take

$$d = \operatorname{diag}(B + \lambda D)$$

where we might in fact use the aforementioned methods in order to approximate the diagonal of $B + \lambda D$ together.

So far, the discussion ignored the cost of obtaining the diagonal of a curvature matrix. Although it is easy to compute Hessian-vector products of arbitrary functions, there exists no efficient exact algorithm for computing the diagonal of the Hessian of a general nonlinear function (Martens et al., 2012, sec. 4), so approximations must be used. Becker and LeCun (1989) report an efficient method for computing the diagonal of the Gauss-Newton matrix, but close examination reveals that it is mathematically unsound (although it can still be viewed as a heuristic approximation).

In case of the Gauss-Newton matrix, it is possible to obtain the exact diagonal at the cost of k runs of back-propagation, where k is the number of output units (Bishop, 1992). This approach can be generalized in the obvious way to compute the diagonal of the generalized Gauss-Newton matrix, and is feasible for classification problems with small numbers of classes, although not feasible for problems such as deep autoencoders or RNNs which have high-dimensional outputs. In the next sections, we describe two practical methods for approximating the diagonal of the GGN matrix regardless of the dimension of the output.

4.9.3 The Empirical Fisher Diagonal

One approach to approximating the diagonal of the GGN matrix G is to instead compute the diagonal of a related matrix for which exact computation of the diagonal is easier. For this purpose Martens (2010) selected the empirical Fisher information matrix $\bar{F}$, which is a crude (and biased) approximation to the well-known Fisher information matrix (Amari, 1998) (which is itself related to the generalized Gauss-Newton matrix). The empirical Fisher is given by

$$\bar{F} = \frac{1}{|S|} \sum_{(x,y) \in S} \nabla L(y, f(x, \theta)) \nabla L(y, f(x, \theta))^\top$$

where $\nabla L(y, f(x, \theta))$ is the gradient of $L(y, f(x, \theta))$ w.r.t. θ.

Because of its special low-rank form, its diagonal is readily computable as

$$\text{diag}(\bar{F}) = \frac{1}{|S|} \sum_{(x,y) \in S} \text{sq}(\nabla L(y, f(x, \theta)))$$

where $\text{sq}(x)$ denotes coordinate-wise square.

Since the $\nabla L(y, f(x, \theta))$'s are available from the gradient computation of ∇h over the mini-batch, additionally computing $\text{diag}(\bar{F})$ over the same mini-batch in parallel incurs no extra cost, save for the possible requirement of storing the individual $\nabla L(y, f(x, \theta))$'s for each $(x, y) \in S$.

This extra cost can be avoided for FNNs, although seemingly not for RNNs. To see how, let A_i be the matrix with $|S|$ columns consisting of the $\mathcal{D}s_i$'s (as in Algorithm 1) for each case in S, and similarly let B_i be the matrix with columns consisting of the $\bar{a}_i$'s. Then it is not hard to see that the gradient of h w.r.t. W_i is given by $\frac{1}{|S|} A_i B_{i-1}^\top$, where $\mathbf{1}_{|S|}$ is the $|S|$-dimensional vector of ones. From this it follows that the entries of $\text{diag}(\bar{F})$ corresponding to W_i are given by $\frac{1}{|S|} \text{sq}(A_i) \text{sq}(B_{i-1})^\top$.

In general, it is possible to compute the sum of squares of gradients in a mini-batch in parallel without storing the squares of the individual gradients whenever the computational graph of the gradient makes only one additive contribution to every parameter for each case in S. This is because we can add $[\nabla L(y, f(x, \theta))]_j^2$ to the appropriate entry of $\text{diag}(F)$ as soon as the single contribution to $[\nabla L(y, f(x, \theta))]_j$ is computed.

However, when the computational graph of the derivative (for a given case) makes multiple additive contributions to a given $[\nabla L(y, f(x, \theta))]_j$, it is necessary to allocate temporary storage for this quantity since we must square the sum of its contributions *before* averaging over the cases.

4.9.4 An unbiased estimator for the diagonal of G

Chapelle and Erhan (2011) give a randomized algorithm for computing an unbiased estimate of the diagonal of the generalized Gauss-Newton matrix, which requires the same amount of work as computing the gradient. And just as with any unbiased estimate, this approach can be repeatedly applied, and the results averaged, to achieve more precise estimates.

The method of Chapelle and Erhan is described in algorithm 6 below.

Algorithm 6 Computing an unbiased estimate of the diagonal of $G(\theta; (x, y))$

1: Sample $v \in \mathbb{R}^m$ from a distribution satisfying $\mathbb{E}[vv^\top] = I$
2: **output** $\text{sq}\left(J^\top {H_L}^{1/2} v\right)$

Here $H_L^{1/2}$ can be any matrix which satisfies ${H_L^{1/2}}^\top H_L^{1/2} = H_L$. Note that since H_L is usually diagonal or diagonal plus rank-1, $H_L^{1/2}$ is an easy matrix to compute or multiply vectors with.

As discussed in Section 3.4, multiplication of an arbitrary vector (in this case, ${H_L}^{1/2} v$) by the transposed Jacobian $J^\top$ of f can be performed efficiently by the usual back-propagation algorithm. The correctness of algorithm 6 is easy to prove:

$$\begin{aligned}\mathbb{E}\left[\text{sq}\left(J^\top {H_L}^{1/2^\top} v\right)\right] &= \mathbb{E}\left[\text{diag}\left((J^\top {H_L}^{1/2^\top} v)(J^\top {H_L}^{1/2^\top} v)^\top\right)\right] = \text{diag}\left(J^\top {H_L}^{1/2^\top} \mathbb{E}[vv^\top] {H_L}^{1/2} J\right) \\ &= \text{diag}\left(J^\top {H_L}^{1/2^\top} {H_L}^{1/2} J\right) = \text{diag}(J^\top H_L J) = \text{diag}(G)\end{aligned}$$

Martens et al. (2012), following the work of Chapelle and Erhan (2011), introduced an efficient unbiased approximation method for estimating the entire Hessian or GGN matrix of a given function (or just their diagonals, if this is desired) with a cost also comparable to computing the gradient. In the special case of estimating the diagonal of the GGN matrix, the two methods are equivalent. Of practical import, Martens et al. (2012) also proved that sampling the components of v from $\{-1, 1\}$ will produce lower variance estimates than those obtained by sampling from $N(0, 1)$.

Computationally, the method of Chapelle and Erhan (2011) is very similar to the method for computing the diagonal of the empirical Fisher that was described in the previous section. Indeed, while the latter computes $\text{sq}\left(\nabla L(y, f(x, \theta))\right) = \text{sq}\left(J^\top \nabla_z L(y, z)\right)$, this method computes $\text{sq}\left(J^\top {H_L}^{1/2} v\right)$ for random v's, and so the methods have similar implementations. In particular, both estimates can be computed in parallel over cases in the mini-batch, and they share the same issue with temporary storage (discussed in the previous subsection), which can be overcome for FNNs but not RNNs.

In our experience, both methods tend to produce preconditioners with similar properties and performance characteristics, although Chapelle and Erhan (2011) found that in certain situations this unbiased estimate gave better results. One clear advantage of this method is that it can correctly account for structural damping, which is not done by using the empirical Fisher, as the gradients of the structural damping objective are equal to zero. The disadvantage of this method is that due to the stochastic nature of the curvature estimates, there could be parameters with non-zero gradients whose diagonal estimates could nonetheless be very small or even zero (which will never happen with the diagonal of the empirical Fisher). The diagonal of the empirical Fisher also has the additional advantage that it can be computed in tandem with the gradient at virtually no extra cost.

4.10 Mini-batching

In modern machine learning applications, training sets can be very large, and a learning algorithm which processes all the examples in the training set to compute each parameter update (called "batch processing") will likely be very slow or even totally impractical (Bottou and Bousquet, 2011). Some training datasets may even be infinite and so it may not even make any sense to talk about an algorithm operating in batch-mode.

"Online" or "stochastic" gradient algorithms like stochastic gradient descent (SGD) can theoretically use gradient information computed on arbitrarily small subsets of the training set, called "mini-batches", as long as the learning rate is sufficiently small. HF, on the other hand, uses mini-batches to estimate the curvature matrix B, which is used to produce large and sometimes aggressive parameter updates that can be very sensitive to noise in B. These curvature matrix estimates may become increasingly low-rank and degenerate as the mini-batch size shrinks (assuming no contribution from damping or weight-decay), which in some cases (see Subsection 4.10.1) may lead to unbounded and nonsensical updates, although the damping mechanisms discussed in Section 4.6 can compensate for this to some extent.

But even without these more obvious degeneracy issues, it can be argued that, intuitively, the matrix B captures "soft-constraints" about how far we can go in any one direction before making things worse, and if the constraints relevant to a particular training case are not well estimated from the mini-batch, the update δ obtained from optimizing M could easily make the objective h worse on such a case, perhaps severely so.

Thus 2nd-order methods like HF which must estimate the curvature only from the current mini-batch, may not work nearly as well with very small mini-batches. And while there are several strategies to deal with mini-batches that are "too small", (as we will discuss in Subsection 4.10.2), the benefits of using a 2nd-order method like HF may be diminished by their use.

Fortunately, in the case of neural networks, there are elegant and natural implementations which exploit data-parallelism and vectorization to efficiently compute gradients and curvature matrix-vector products over large mini-batches (see Section 4.5). For highly parallel architectures like GPUs, these tend to give an increase in computational cost which remains sublinear (as a function of mini-batch size) up to and beyond mini-batch sizes which are useful in HF.

4.10.1 Higher quality gradient estimates

Unlike 1st order optimization schemes like SGD where the number of updates to θ required to approach a good solution can reach as high as $10^5 - 10^7$, the number of updates required by a strong 2nd-order optimizer like HF is in our experience orders of magnitude smaller, and usually around $10^2 - 10^3$. However, while the linear term $b = -\nabla h(\theta_k)$ passed to CG needs to be computed only once for each update, CG may require on the order of $10^2 - 10^3$ matrix-vector products with the curvature matrix $A = B$ to produce each update. These matrix-vector products are by far the most computationally costly part of any truncated Newton approach.

It may therefore be cost-effective to compute the gradient on a larger mini-batch than is used to compute the matrix-vector products. Martens (2010) recommended using this technique (as does Byrd et al. (2011)), and in our experience it can often improve optimization speed if used carefully and in the right contexts. But despite this, there are several good theoretical reasons why it might be better, at least in some situations, to use the same mini-batch to compute gradient and curvature matrix-vector products. These have been corroborated by our practical experience.

We will refer to the mini-batches used to compute the gradient and curvature matrix-vector products as the "gradient mini-batch" and "curvature mini-batch", respectively.

When the gradient and curvature mini-batches are equal, and the GGN curvature matrix is used, the quadratic model M maintains its interpretation as the local Taylor series approximation of a convex function, which will simply be the approximation of h obtained by linearizing f (see eqn. 3.4), but restricted to the data in the current mini-batch. In such a situation, with some additional reasonable assumptions about the convex function L (strong

convexity would be sufficient, but is more than what is needed), the quadratic model M can be written as

$$\begin{aligned}
M(\delta) &= \frac{1}{2}\delta^\top B\delta + \nabla h_k^\top \delta + f(\theta_k) = \frac{1}{2}\delta^\top J^\top H_L J\delta + (J^\top \nabla_z L(y,z))^\top \delta + f(\theta_k) \\
&= \frac{1}{2}(J\delta)^\top H_L(J\delta) + \nabla_z L(y,z)^\top {H_L}^{-1} H_L(J\delta) + \frac{1}{2}\nabla_z L(y,z)^\top {H_L}^{-1} H_L {H_L}^{-1}\nabla_z L(y,z) \\
&\qquad - \frac{1}{2}\nabla L^\top {H_L}^{-1} H_L {H_L}^{-1}\nabla_z L(y,z) + f(\theta_k) \\
&= \frac{1}{2}(J\delta + {H_L}^{-1}\nabla_z L(y,z))^\top H_L(J\delta + {H_L}^{-1}\nabla_z L(y,z)) + c \\
&= \frac{1}{2}\|J\delta + {H_L}^{-1}\nabla_z L(y,z)\|^2_{H_L} + c
\end{aligned}$$

where $c = h(\theta_k) - \frac{1}{2}\nabla_z L(y,z)^\top {H_L}^{-1}\nabla_z L(y,z)$ and all quantities are computed only on the current mini-batch. Here we have used the fact that H_L is invertible, which follows from the fact that L is strongly convex.

This result is interesting because it applies only when B is the generalized Gauss-Newton matrix (instead of the Hessian), and it establishes a bound on the maximum improvement in h that the quadratic model M can ever predict, which is given by $\frac{1}{2}\nabla_z L(y,z)^\top {H_L}^{-1}\nabla_z L(y,z)$. This quantity does not depend on the properties of the network, but rather only on its current predictions and the associated convex loss function.

Such a boundedness result does not exist whenever M is estimated using different mini-batches for the gradient and curvature. In this case, the estimated gradient may easily lie outside the column space of the estimated curvature matrix in the sense that there may exist directions d s.t. $g^\top d < 0$, $\|d\| = 1$, but $d^\top B d = 0$. In such a case it is easy to see that the quadratic model is unbounded and $M(\alpha d) \to -\infty$ as $\alpha \to \infty$. While boundedness can be guaranteed with the inclusion of damping penalty terms which ensure that the damped curvature matrix $\hat{B}$ is positive definite, and will also be guaranteed when the curvature matrix B is full rank, it may be the case that the boundedness is "weak". In particular, we have have that $d^\top B d$ may be extremely small, leading to a nearly degenerate update δ. For example, when using Tikhonov damping then we know that $d^\top \hat{B} d \geq \lambda$, but in order for this to sufficiently constrain the update along direction d, λ may have to be large enough that it would impose unreasonably high constraints on optimization in *all* directions.

More intuitively, the gradient represents a linear reward for movement in certain directions d (the strength of which is given by $g^\top d$) while the curvature matrix represents a quadratic penalty. If we include the linear rewards associated with a subset of cases without also including the corresponding quadratic penalties, then there is a chance that this will lead to a degenerate situation where some directions will have lots of reward (predicted linear reduction) without any corresponding penalty (curvature). This can result in an update which makes h worse even on cases contained in both the gradient and curvature mini-batches, for reasons that have nothing directly to do with a breakdown in the reliability of the quadratic approximation to h. On the other hand, if the curvature and gradient mini-batches are equal *and* the quadratic approximation to h is otherwise reliable (or properly damped), then using equal gradient and curvature mini-batches provides a minimal guarantee that h will improve on the current mini-batch after the proposed update is applied.

Another more subtle way in which using a smaller-sized curvature mini-batch than gradient mini-batch could be counterproductive, is that in addition to causing a dangerous underestimation of the curvature associated with the left-out cases, it may also lead to an *over*estimation of the curvature for the cases actually in the curvature-mini-batch. This is because when we compute estimates of the curvature by averaging, we must divide by the number of cases in the mini-batch, and since this number will be smaller for the curvature estimate than for the gradient, the gradient contributions from these cases will be smaller in proportion to the corresponding curvature terms.

Byrd et al. (2011) showed that if the eigenvalues of the curvature matrices (estimated using any method) are uniformly bounded from below in the sense that there exists $\mu > 0$ s.t. $\hat{B} - \mu I$ is PSD for all possible $\hat{B}$'s which we might produce, then assuming the use of a basic line-search and other mild conditions, a truncated Newton algorithm like HF which estimates the gradient on the full training set will converge in the sense that the gradients will go to zero. But this result makes no use of the particular form of B and says nothing about the rate of convergence, or how small the updates will have to be. As far as the theorem is concerned, B can be any λ dependent matrix with the required boundedness property, and need not have anything to do with local quadratic models of h at all.

Despite all of these objections, the higher quality estimates of the gradient may nonetheless provide superior convergence properties in some situations. For example, in later stages of a stochastic optimization, it is usually the "noise" in the gradient which is the primary reason most gradient-based optimizers slow down, and so increasing the size of the gradient mini-batch can be the best way to accelerate convergence in such situations. The best trade-off between these various factors is likely to be highly dependent on the particular problem, the stage of the optimization (early versus late), and the damping mechanisms being used. Our experience is that penalty and CG-truncation damping become more active when there is a significant qualitative mismatch between the gradient and curvature estimates, which is more likely to happen when the cases in the training dataset, or the network's associated patterns of activation, are particularly "diverse".

4.10.2 Mini-batch overfitting and methods to combat it

As mentioned in the previous section, the updates produced by HF may be effectively "overfitted" to the current mini-batch of training data. While a single update of SGD has the same problem, this is less of an issue because the updates are extremely cheap and numerous. HF, by contrast, performs a run of CG with anywhere between 10 to 300+ iterations, which is a long and expensive process, and must be performed using the same fixed estimates of the gradient and curvature from a single mini-batch. Ideally, we could use a stochastic algorithm when optimizing the local quadratic models which would be able to see much more data at no extra cost. Unfortunately we are not aware of any batch methods which possess the same strongly optimal performance for optimizing quadratics as CG does, while also working well as stochastic methods.

The simplest solution to dealing with the mini-batch overfitting problem is to increase the size of the mini-batches, thus providing CG with more accurate estimates of the gradient and curvature. When optimizing neural networks, we have observed that the mini-batch overfitting problem becomes gradually worse as optimization proceeds, and so implementing this solution will require continually growing the mini-batches, possibly without bound. Fortunately, there are other ways of dealing with this problem.

The damping methods discussed in Section 4.6 were developed to compensate for the untrustworthiness of the local quadratic approximations being made to h. These methods work by imposing various soft or hard constraints on the update δ in order to keep it "closer" to 0 according to some metric (which is a region where M is trustworthy by construction).

Using mini-batches to compute the gradient and curvature imposes a different kind of untrustworthiness on the quadratic model, arising from the fact that the function being approximated is not actually the true objective but rather just an unbiased sample-based approximation of it. But despite the differing nature of the source of this untrustworthiness, the previously developed damping methods turn out to be well suited to the task of compensating for it, in our experience.

Of the various ways to deal with mini-batch overfitting, decreasing the maximum number of CG steps, using larger minbatches for the gradient, and decreasing the default learning rate (which is equivalent to damping by

adding multiples of B, as discussed in Section 4.6.3) according to some SGD-like schedule, seem to be the most effective approaches in practice. If standard Tikhonov damping is used and its strength λ is increased to compensate for mini-batch overfitting, this will make HF behave asymptotically like SGD with a dynamically adjusted learning rate.

There is a compelling analogy between the mini-batch overfitting which occurs when optimizing δ with CG, and the general overfitting of a non-linear optimizer applied to a conventional learning problem. And some of the potential solutions to both of these problems turn out to be analogous as well. Tikhonov damping, for example, is analogous to an ℓ_2 prior or "weight-decay" penalty (but centered at $\delta = 0$), and CG truncation is analogous to "early-stopping".

Recall that damping approaches are justified as methods for dealing with quadratic approximation error because as the "size" of the update shrinks (according to any reasonable metric), it will eventually lie inside a region where this source of error must necessarily be negligible. This is due to the simple fact that any terms in the Taylor series expansion of h which are unmodeled by M will approach zero much more rapidly than the size of the update. It is important to keep in mind that a similar justification *does not* apply to the handling of mini-batch–related estimation errors with update damping methods. Indeed, the negative gradient computed on a given mini-batch may not even be a descent direction for the total objective (as computed on the complete training set), and even an infinitesimally small update computed from a given mini-batch may actually make the total objective worse.

Thus, when tuning damping mechanisms to handle mini-batch overfitting (either by hand, or dynamically using an automatic heuristic), one shouldn't aim for obtaining a reduction in the total h that is a fixed multiple of that which is predicted by the mini-batch–computed M (as is done in Section 4.6.5), but rather to simply obtain a more modest reduction which is proportional to the relative contribution of the current mini-batch to the entire training dataset.

It is also worthwhile noting that the practice of initializing CG from the update computed at the previous iteration of HF (as discussed in Section 4.8) seems to bias CG towards finding solutions that generalize better to data outside of the current mini-batch. We don't have a good understanding for why this happens, but one possible explanation is that by carrying δ over to each new run of CG and performing an incomplete optimization on it using new data, δ is allowed to slowly grow (as HF iterates) along low-curvature directions[9] that by necessity, must generalize across lots of training data. The reasoning is that if such slowly optimized directions didn't generalize well, then they would inevitably be detected as high-curvature ascent directions for some new mini-batch and quickly zeroed out by CG before ever having a chance grow large in δ.

Finally, Byrd et al. (2012) has developed methods to deal with the mini-batch overfitting problem, which are based on heuristics that increase the mini-batch size and also terminate CG early, according to estimates of the variance of the gradient and curvature-matrix vector products. While this is a potentially effective approach (which we don't have experience with), there are several problems with it, in theory. First, variance is measured according to highly parameterization-dependent metrics which are not particularly meaningful. Second, increasing the size of the mini-batch, which is only one method to deal with mini-batch overfitting, is not a strategy which will remain practical for very long. Thirdly, aggressive early termination heuristics for CG tend to interact badly with non-zero CG initializations[10] and other forms of damping. And finally, there are other more direct ways of measuring how well updates will generalize, such as simply monitoring h on some training data outside of the current mini-batch.

[9] which get optimized much more slowly by CG than high-curvature directions, as shown in Section 4.7

[10] because termination of CG may be forced before $\hat{M}(\delta) < 0$ is achieved

4.11 Tricks and recipes

There are many things to keep in mind when designing an HF-style optimization algorithm for a particular application and it can be somewhat daunting even to those of us that have lots of experience with the method. So in order to make things easier in this regard, in this section we relate some of our experience in designing effective methods, and describe several particular setups that seem to work particularly well for certain deep neural network learning problems, assuming the use of a standard parameterization.

Common elements to all successful approaches we have tried are:

- use of the GGN matrix instead of the Hessian
- the CG initialization technique described in Section 4.8
- a well-designed preconditioner. When using Tikhonov damping, a reasonable choice is an estimate of the diagonal of the GGN matrix, modified using the technique described in Section 4.9.2 with $\kappa = 0$ and $\xi = 0.75$. When using one of the scale-sensitive methods described in Section 4.6.3, it *may* be necessary to increase κ to something like 10^{-2} times the mean of the diagonal entries of D
- the use of one of diagonal damping methods, possibly in conjunction with structure damping for certain RNN learning problems. For feedforward network learning problems under the default parameterization, Tikhonov damping often works well, and usually so does using a modified estimate of the diagonal of the GGN matrix, provided that κ is large enough (as in the previous point)
- the use of the progress-based termination criterion for CG described in Section 4.2 in addition to some other condition which may stop CG sooner, such as a fixed iteration limit
- dynamic adjustment of damping constants (e.g. λ) according to the Levenberg-Marquardt (LM) heuristic or a similar method (see Section 4.6.5)

Now, we describe the particulars of each of these successful methods.

First, there is the original algorithm described by Martens (2010), which works pretty well. Here, the "CG-backtracking" approach is used to select an iterate based on the objective function value (see Section 4.6.7), the gradient is computed on a larger subset of the training data than the curvature, and CG is always terminated before reaching a fixed maximum number of steps (around $50 - 250$, depending on the problem).

Second, there is a subtle but powerful variation on the above method which differs only in how CG is terminated, how the iterate used for the update δ is selected, and how CG is initialized at the next iteration of HF. In this variation, CG is terminated as soon as the objective function, as evaluated on the data in the "curvature mini-batch" (see Section 4.10.1) gets significantly worse than its value from some number of steps ago (e.g. 10). The iterate used as the parameter update δ is selected to minimize M (or perhaps h) as evaluated on some data which is *not* contained in the curvature mini-batch. Lastly, CG is initialized at the next iteration $k + 1$, not from the previous update δ_k, but instead from the CG iterate which gave the highest objective function value on the curvature mini-batch (which will be close to the last). In practice, the quantities used to determine when to terminate CG, and how to select the best iterate, do not need to be computed at every step of CG, and can also be computed on much smaller (but representative) subsets of data.

Finally, an approach which has emerged recently as perhaps the most efficient and effective, but also the most difficult to use, involves modifying HF to behaving more like a tradition momentum method, thus making stronger use of the CG initializations (see Section 4.8) to better distribute work involved in optimizing the local quadratics

Table 4.2: Experimental parameters

NAME	SIZE	K	ENCODER DIMS
CURVES	20000	5000	784-400-200-100-50-25-6
MNIST	60000	7500	784-1000-500-250-30
FACES	103500	5175	625-2000-1000-500-30

across many iterations. To do this, we use a smaller maximum number of CG steps (around 25 to 50), smaller mini-batches of training-data, and we also pay more attention to the CG initialization decay-constant ζ, which usually means increasing it towards the end of optimization. Using shorter runs of CG helps with mini-batch overfitting, and makes it feasible to use smaller mini-batches and also compute the gradient and curvature using the same data. And as discussed in Section 4.10.1, computing the gradient on the same data as the curvature has numerous theoretical advantages, and in practice seems to result in a reduced need for extra damping, thus resulting in a λ that shrinks reliably towards 0 when adjusted by the LM heuristic. However, this method tends to produce "noisy" updates, which while arguably beneficial from the standpoint of global optimization, make it more difficult to obtain finer convergence on some problems. So when nearing the end of optimization, we adopt some of the methods described in Section 4.10.2, such as lowering the learning rate, using shorter CG runs, increasing the mini-batch size, and/or switching back to using larger mini-batches to compute gradients (making sure to raise the damping constant λ to compensate for this) in order to achieve fine convergence.

One more piece of general advice we have is that using small amounts of weight-decay regularization can be highly beneficial from the standpoint of global optimization. In particular, to get the lowest training error possible, we have observed that it helps to use such regularization at the beginning of optimization only to disable it near the end. Also, using a good initialization is extremely important in regards to global optimization, and methods which work well for deep networks include the sparse initialization scheme advocated by Martens (2010), and the method of Glorot and Bengio (2010), and of course the pre-training techniques of Hinton and Salakhutdinov (2006).

4.12 Experiments on deep autoencoders

In this section we present the results from a series of experiments from the original publication on HF (Martens, 2010) which are designed to test the effectiveness of our HF approach on the deep auto-encoder problems considered by Hinton and Salakhutdinov (2006) (abbr. H&S). We adopt precisely the same model architectures, datasets, loss functions and training/test partitions that they did, so as to ensure that our results can be directly compared with theirs.

Each dataset consists of a collection of small grey-scale images of various objects such as hand-written digits and faces. Table 4.2 summarizes the datasets and associated experimental parameters, where **size** gives the size of the training set, **K** gives the size of minibatches used, and **encoder dims** gives the encoder network architecture. In each case, the decoder architecture is the mirror image of the encoder, yielding a "symmetric autoencoder". This symmetry is required in order to be able to apply H&S's pre-training approach. Note that CURVES is the synthetic curves dataset from H&S's paper and FACES is the augmented Olivetti face dataset.

Despite how the optimal mini-batch size can vary over the course of the optimization, we kept the mini-batch sizes constant. We used the same meta-parameters settings across the experiments. We used a standard back-tracking line-search algorithm (in addition to, and after, the "CG-backtracking") on the directions produced by

Table 4.3: Results (training and test errors)

	PT + NCG	**RAND+HF**	**PT + HF**
CURVES	0.74, 0.82	0.11, 0.20	0.10, 0.21
MNIST	2.31, 2.72	1.64, 2.78	1.63, 2.46
MNIST*	2.07, 2.61	1.75, 2.55	1.60, 2.28
FACES	-, 124	55.4, 139	-,-
FACES*	-,-	60.6, 122	-,-

CG, although this would only rarely select a distance less than 1 since they were already well-scaled.

We implemented our approach using the GPU-computing MATLAB package Jacket. We also re-implemented, using Jacket, the precise approach considered by H&S, using their provided code as a basis, and then re-ran their experiments using *many* more training epochs than they did, and for far longer than we ran our HF approach on the same models. With these extended runs we were able to obtain slightly better results than they reported for both the CURVES and MNIST experiments. Unfortunately, we were not able to reproduce their results for the FACES dataset, as each network we pre-trained had very high generalization error, even before fine-tuning. We ran each method until it either seemed to converge, or started to overfit (which happened for MNIST and FACES, but not CURVES). We found that since our method was much better at fitting the training data, it was thus more prone to overfitting, and so we ran additional experiments where we introduced an ℓ_2 prior on the connection weights.

While we did not spend any time optimizing it, this implementation was nonetheless very fast thanks to massively parallel processing capabilities of the GPU. Running times for each experiment varied between roughly 1 and 18 hours, depending on the size of the model and whether pre-training was used. Moreover, since most of the time spent was during the final stages of optimization, competitive results were often available much sooner.

Table 4.3 summarizes our results, where **PT+NCG** is the pre-training + non-linear CG fine-tuning approach of H&S, **RAND+HF** is our Hessian-free method initialized randomly, and **PT+HF** is our approach initialized with pre-trained parameters. The numbers given in each entry of the table are the average sum of squared reconstruction errors on the training-set and the test-set. The *'s indicate that an ℓ_2 prior was used, with strength 10^{-4} on MNIST and 10^{-2} on FACES. Error numbers for FACES which involve pre-training are missing due to our failure to reproduce the results of H&S on that dataset (instead we just give the test-error number they reported).

Figure 4.5 demonstrates the performance of our implementations on the CURVES dataset. Pre-training time is included where applicable. This plot is not meant to be a definitive performance analysis, but merely a demonstration that our method is indeed quite efficient.

4.13 Experiments on RNNs

In this section we describe the experiments (from Martens and Sutskever (2011)) we performed training RNNs using a basic version of HF on various pathological synthetic problems which are believed to be difficult to standard gradient descent (e.g., Hochreiter et al., 2000). These problems were taken directly from Hochreiter and Schmidhuber (1997), which used the specialized RNN architecture called Long Short-term Memory (LSTM) to solve them.

Each of these problems has a similar form, where the inputs are long sequences whose start (and possibly middle) are relevant, and the goal of the network 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

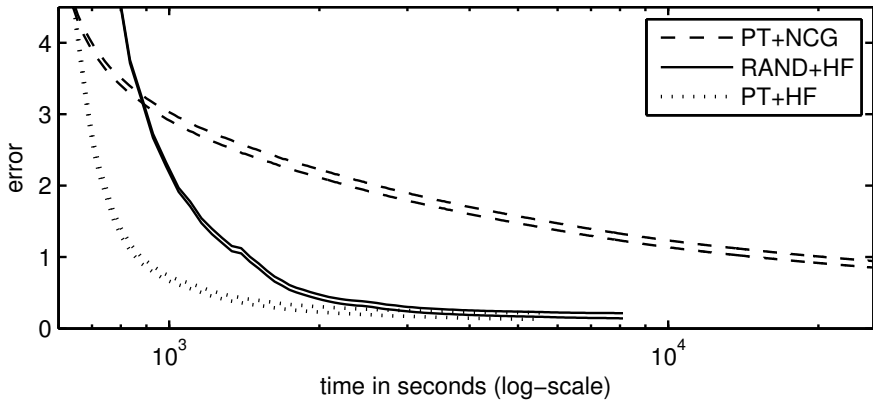

Figure 4.5: Error (train and test) vs. computation time on CURVES

problem is illustrated in Figure 4.6. The difficulty of these problems increases with T (the sequence length), since longer sequences exhibit longer range dependencies.

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 ($W_{1,0}^{(0)}$, $W_{1,1}^{(1)}$, and $W_{2,1}^{(0)}$) is sparsely initialized so that each unit is connected to 15 other units. The nonzero connections of $W_{1,1}^{(1)}$, of $W_{2,1}^{(0)}$ and the biases are sampled independently from a Gaussian with mean 0 and variance $\frac{1}{15}$, while the nonzero connections of $W_{1,0}^{(0)}$ are independently sampled from a Gaussian with mean 0 and a variance of 1. These constants were chosen so that the initial gradient wouldn't vanish or explode too strongly.

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 iteration of HF used a fresh set of sequences and the test set also consists of 10,000 sequences. We set the maximum number CG steps to be 300, and stopped CG early once the magnitude of the reduction in the objective decreased below 90% of the maximum so far obtained 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 training 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 GGN matrix can be used.

The addition, multiplication, and XOR problem

The addition problem is described in Figure 4.6. 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.

The multiplication and the XOR problem are completely analogous.

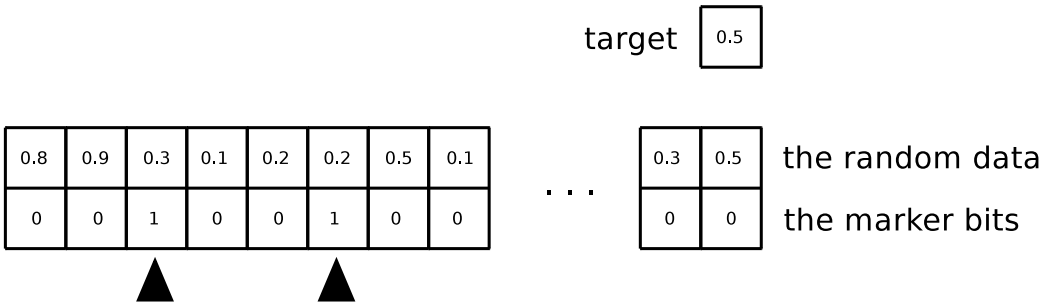

Figure 4.6: An illustration of the addition problem, a typical problem with pathological long term dependencies. The target is the sum of the two marked numbers (indicated with the black arrows). The "..." represent a large number of timesteps.

Note that these problems are challenging for the RNN not only because of their significant long term dependencies, but also because the RNN's hidden state is exposed to both the relevant and the irrelevant inputs at each timestep and must learn to accurately remember the former whilst ignoring the latter.

The temporal order problem

In this task (task 6a in Hochreiter and Schmidhuber, 1997), 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.

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 (see Hochreiter and Schmidhuber, 1997, task 6b, for more details).

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 uniformly from $\{1, 2\}$, while the rest of the symbols are drawn uniformly from $\{3, \ldots, 100\}$. 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.

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.

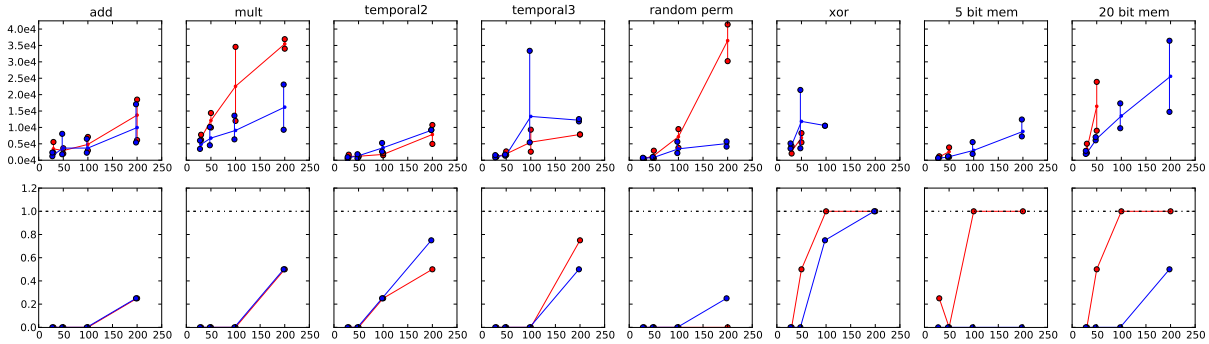

Figure 4.7: For each problem, the plots in the top row give the mean number of minibatches processed before the success condition was achieved (amongst trials that didn't fail) for various values of T (x-axis), with the min and max given by the vertical bars. Here, a minibatch is a set of 1000 sequences 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 defined as a failure. The failure rates are shown in the bottom row. Red is HF with only standard adaptive Tikhonov damping and blue is HF with structural damping included.

4.13.1 Results and discussion

The results from our experiments training RNNs with HF on the pathological synthetic problems are shown in Figure 4.7. We refer the reader to Hochreiter and Schmidhuber (1997) for a detailed analysis of the performance of LSTMs on these tasks. Our results establish that our HF approach is capable of training RNNs to solve problems with very long-term dependencies that are believed to be very difficult for standard gradient descent. 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 4.7). This is a problem that could potentially be addressed with a more careful random initialization than the one we used, as evidenced by the significant amount of problem-specific hand tweaking done by Hochreiter and Schmidhuber (1997) (see their Table 10).

4.14 Epilogue

In the years following the publication of the results presented in the previous couple sections, it has become apparent that with a reasonably well designed random initialization, the use of momentum (whose importance varies by problem), and the application of a certain amount of care regarding the tuning of learning rate and momentum decay constants, SGD can work well for optimizing deep autoencoders, RNNs, and other related models. Extreme curvature remains an important problems in neural network optimization, but the use of strong forms of momentum (with decay parameters set close to 1) seem to adequately address it (Sutskever, Martens, Dahl, and Hinton, 2013). A possible exception to this are the pathological long-term dependency problems considered in Section 4.13, where SGD with momentum is very slow and brittle compared to HF, and will outright fail unless certain very specific initialization and data pre-processing routines are employed (Sutskever, Martens, Dahl, and Hinton, 2013).

The experimental findings we presented are best understood within their original historical context of 2010-2011, when it was believed that randomly initialized SGD was inadequate to the task of optimizing very deep neural networks like these deep autoencoders and RNNs. These beliefs were based on early observations about vanishing gradients (Hochreiter et al., 2000), as well as negative results on SGD published in various works (e.g. Hinton and Salakhutdinov, 2006).

These experiments, and ones performed using more light-weight versions of HF that more closely resemble SGD with momentum (see Section 4.11, and the experiments reported in Sutskever, Martens, Dahl, and Hinton (2013)), challenged these beliefs by demonstrating that a classical gradient based local-search optimization approach could work very well for deep learning problems. This motivated later work revisiting SGD.

Another important finding from these experiments is that high-quality update steps computed using 2nd-order information (and subject to damping of course), when applied to deep neural network objectives, can make a couple orders of magnitude more progress than the simple gradient descent steps. This fact motivates the search for an approach that can compute such updates more efficiently than HF does. In Chapter 6 we develop an optimization method based on such an approach that performs much better in practice than SGD with momentum does on the deep autoencoder problems. This method makes use of many of the ideas discussed in this chapter, such as damping, the Generalized Gauss-Newton matrix, optimization of local quadratic models, etc.

Chapter 5

New insights and perspectives on the natural gradient method

5.1 KL divergence objective

The natural gradient method of Amari (1998) can be potentially applied to any objective function which measures the performance of some statistical model. However, it enjoys richer theoretical properties when applied to objective functions based on the KL divergence between the model's distribution and the target distribution, or certain approximations/surrogates of these.

In this section we will establish the basic notation and properties of these objective functions, and discuss the various ways in which they can be formulated. Each of these formulations will be analogous to a particular formulation of the Fisher information matrix and natural gradient (as defined in Section 5.2), which will differ in subtle but important ways.

In the idealized setting, input vectors x are drawn independently from a *target* distribution Q_x with density function $q(x)$, and the corresponding (target) outputs y from a conditional *target* distribution $Q_{y|x}$ with density function $q(y|x)$. For the purposes of this work we will assume that both of these distributions exist, although in more general contexts they may not.

We define the goal of learning as the minimization of the KL divergence from target joint distribution $Q_{x,y}$, whose density is $q(y, x) = q(y|x)q(x)$, to the learned distribution $P_{x,y}(\theta)$, whose density is $p(x, y|\theta) = p(y|x, \theta)q(x)$. Note that the second $q(x)$ is not a typo here, since we are not learning the distribution over x, only the conditional distribution of y given x. Our objective function is thus

$$\mathrm{KL}(Q_{x,y} \| P_{x,y}(\theta)) = \int q(x, y) \log \frac{q(x, y)}{p(x, y|\theta)} dx dy$$

This is equivalent to the expected KL divergence

$$\mathrm{E}_{Q_x}[\mathrm{KL}(Q_{y|x} \| P_{y|x}(\theta))] \tag{5.1}$$

since we have

$$\begin{aligned}\mathrm{E}_{Q_x}[\mathrm{KL}(Q_{y|x}\|P_{y|x}(\theta))] &= \int q(x)\int q(y|x)\log\frac{q(y|x)}{p(y|x,\theta)}dydx \\ &= \int q(x,y)\log\frac{q(y|x)q(x)}{p(y|x,\theta)q(x)}dxdy \\ &= \mathrm{KL}(Q_{x,y}\|P_{x,y}(\theta))\end{aligned}$$

It is often the case that we only have samples from Q_x and no direct knowledge of its density function. Or the expectation w.r.t. Q_x in eqn. 5.1 may be too difficult to compute. In such cases, we can substitute an empirical *training* distribution $\hat{Q}_x$ in for Q_x, which is given by a set S of samples from Q_x. This gives the objective

$$\mathrm{E}_{\hat{Q}_x}[\mathrm{KL}(Q_{y|x}\|P_{y|x}(\theta))] = \frac{1}{|S|}\sum_{x\in S}\mathrm{KL}(Q_{y|x}\|P_{y|x})$$

Provided that $q(y|x,\theta)$ is known for each x in S and that $\mathrm{KL}(Q_{y|x}\|P_{y|x})$ can be efficiently computed, we can use the above expression as our objective.

Otherwise, as is often the case, we might only have access to a single sample from $Q_{y|x}$ for each $x \in S$, giving an empirical *training* distribution $\hat{Q}_{y|x}$. Substituting this in for $Q_{y|x}$ gives the objective function

$$\mathrm{E}_{\hat{Q}_x}[\mathrm{KL}(\hat{Q}_{y|x}\|P_{y|x})] = \frac{1}{|S|}\sum_{(x,y)\in S} 1\log\frac{1}{p(y|x,\theta)} = -\frac{1}{|S|}\sum_{(x,y)\in S}\log p(y|x,\theta)$$

which is the same as the objective minimized in standard maximum likelihood learning. Note that we have extended S to be the set of the (x, y) pairs, which agrees with how S was defined in Section 2.3.

This kind of objective function fits into the general supervised learning framework described in Section 2.3 as follows. We define the learned conditional distribution $P_{y|x}$ to be the composition of the deterministic neural network function $f(x,\theta)$, and an "output" conditional distribution $R_{y|z}$ (with associated density function $r(y|z)$), so that

$$P_{y|x} = R_{y|f(x,\theta)}$$

We then define the loss function as $L(y,z) = -\log r(y|z)$.

Note that given some loss L not necessarily defined in this way, one can find a corresponding R where this definition does apply, provided that L satisfies certain properties. In particular, if $\exp(-L(y,z))$ has the same finite integral w.r.t. y for each z, then one can define R by taking $r(y|z) \propto \exp(-L(y,z))$, where the proportion is w.r.t. both y and z.

5.2 Various definitions of the natural gradient and the Fisher information matrix

The usual definition of the natural gradient (Amari, 1998) which appears in the literature is

$$\tilde{\nabla}h = F^{-1}\nabla h$$

where F is the Fisher information matrix of $P_{x,y}(\theta)$ w.r.t. θ. F is given by

$$F = \mathrm{E}_{P_{x,y}}\left[\nabla \log p(x,y|\theta) \nabla \log p(x,y|\theta)^\top\right] \tag{5.2}$$

$$= -\mathrm{E}_{P_{x,y}}\left[H_{\log p(x,y|\theta)}\right] \tag{5.3}$$

where gradients and Hessians are taken w.r.t. θ. For the purposes of brevity we will often refer to the Fisher information matrix simply as the "Fisher".

It can be immediately seen from the first of these expressions for F that is is PSD (since it's the expectation of something which is trivially PSD, a vector outer-product). And from the second expression we can see that it also has the interpretation of being the negative expected Hessian of $\log p(x,y|\theta)$.

Because $p(x,y|\theta) = p(y|x,\theta)q(x)$ where $q(x)$ doesn't depend on θ, we have

$$\nabla \log p(x,y|\theta) = \nabla \log p(y|x,\theta) + \nabla \log q(x) = \nabla \log p(y|x,\theta)$$

and so F can also be written as the expectation w.r.t. to Q_x of the Fisher information matrix of $P_{y|x}(\theta)$ as follows:

$$F = \mathrm{E}_{Q_x}\left[\mathrm{E}_{P_{y|x}}\left[\nabla \log p(y|x,\theta) \nabla \log p(y|x,\theta)^\top\right]\right] \quad \text{or} \quad F = -\mathrm{E}_{Q_x}\left[\mathrm{E}_{P_{y|x}}\left[H_{\log p(y|x,\theta)}\right]\right]$$

In Amari (1998), this version of F is computed explicitly for a basic perceptron model (basically a neural network with 0 hidden layers) in the case where Q_x is given by $N(0,I)$.

However, in practice the real $q(x)$ may be not directly available, or it may be difficult to integrate $H_{\log p(y|x,\theta)}$ over Q_x. For example, the conditional Hessian $H_{\log p(y|x,\theta)}$ corresponding to a multilayer neural network may be far too complicated to be analytically integrated, even for a very simple Q_x. In such situations, Q_x may be replaced with its empirical version $\hat{Q}_x$, giving

$$F = \frac{1}{|S|}\sum_{x \in S} \mathrm{E}_{P_{y|x}}\left[\nabla \log p(y|x,\theta) \nabla \log p(y|x,\theta)^\top\right] \quad \text{or} \quad F = -\frac{1}{|S|}\sum_{x \in S} \mathrm{E}_{P_{y|x}}\left[H_{\log p(y|x,\theta)}\right]$$

This is the version of F considered in Park et al. (2000).

Note that when $L(y,z) = -\log r(y|z)$ as in Section 5.1, the Fisher has the interpretation of being the expectation under $P_{x,y}$ of the Hessian of $L(y, f(x,\theta))$. Meanwhile, the Hessian H of h is given by the expectation under $\hat{Q}_{x,y}$ of the Hessian of $L(y, f(x,\theta))$ (where $\hat{Q}_{x,y}$ is given by the density $\hat{q}(x,y) = \hat{q}(y|x)\hat{q}(x)$), and so F can be seen as an approximation of H in this sense. Moreover, we can see that by computing F using $\hat{Q}_x$ instead of Q_x as described above, the quality of this approximation will arguably be better since H is also computed using $\hat{Q}_x$ instead of Q_x.

5.3 Geometric interpretation

The negative gradient $-\nabla h$ can be interpreted as the steepest descent direction for h in the sense that it yields the most reduction in h per unit of change in θ, where change is measured using the standard Euclidean norm $\|\cdot\|$. More formally we have

$$\frac{-\nabla h}{\|\nabla h\|} = \lim_{\epsilon \to 0} \frac{1}{\epsilon} \operatorname*{arg\,min}_{d:\|d\| \leq \epsilon} h(\theta + d)$$

This interpretation exposes the strong dependence of the gradient on the Euclidean geometry of the parameter

space (as defined by the norm $\|\cdot\|$).

One way to motivate the natural gradient is to show that it can be viewed as a steepest descent direction, much like the negative gradient can be, except with respect to a metric that is intrinsic to the distributions being modeled as opposed to the default Euclidean metric in parameter space. In particular, the natural gradient can be derived by adapting the steepest descent formulation to use an alternative definition of (local) distance based on the "information geometry" (Amari and Nagaoka, 2000) of the space of probability distributions (as induced by the parameters). The particular distance function[1] which gives rise to the natural gradient turns out to be

$$\mathrm{KL}(P_{x,y}(\theta + d)\|P_{x,y}(\theta))$$

To make this formal, we will first show how the KL divergence and the Fisher are fundamentally connected. The Taylor series expansion of the above distance is

$$\mathrm{KL}(P_{x,y}(\theta + d)\|P_{x,y}(\theta)) = \frac{1}{2}d^\top F d + O(d^3)$$

where "$O(d^3)$" is short-hand to mean terms that are order 3 or higher in the entries of d. Thus F defines the local quadratic approximation of this distance, and so gives the mechanism of *local* translation between the geometry of the space of distributions, and that of the original parameter space with its default Euclidean geometry.

To make use of this connection we first observe that for a general positive definite matrix A we have

$$\frac{-A^{-1}\nabla h}{\|\nabla h\|_{A^{-1}}} = \lim_{\epsilon \to 0} \frac{1}{\epsilon} \operatorname*{arg\,min}_{d:\|d\|_A \leq \epsilon} h(\theta + d)$$

where the notation $\|v\|_B$ is defined by $\|v\|_B = \sqrt{v^\top B v}$.

Then taking $A = \frac{1}{2}F$ and using the above Taylor series expansion of the KL divergence to show that $\mathrm{KL}(P_{x,y}(\theta + d)||P_{x,y}(\theta)) \to \frac{1}{2}d^\top F d = \|d\|^2_{\frac{1}{2}F}$ as $\epsilon \to 0$, with some extra work (e.g. Arnold et al., 2011) it follows that

$$-\sqrt{2}\frac{\tilde{\nabla} h}{\|\nabla h\|_{F^{-1}}} = \lim_{\epsilon \to 0} \frac{1}{\epsilon} \operatorname*{arg\,min}_{d\,:\,\mathrm{KL}(P_{x,y}(\theta+d)\|P_{x,y}(\theta)) \leq \epsilon^2} h(\theta + d)$$

Thus the negative natural gradient is indeed the steepest descent direction in the space of distributions where distance is (approximately) measured in local neighborhoods by the KL divergence. While this might seem impossible since the KL divergence is in general not symmetric in its two arguments, it turns out that $\mathrm{KL}(P_{x,y}(\theta + d)\|P_{x,y}(\theta))$ is locally/asymptotically symmetric as d goes to zero, and so will be (approximately) symmetric in a local neighborhood [2].

Note that both F and $\tilde{\nabla} h$ are defined in terms of the standard basis in θ-space and so obviously depend on the parameterization of h. But the KL divergence does not, and instead only depends on the form of the predictive distribution $P_{y|x}$. Thus, the direction in distribution space defined implicitly by $\tilde{\nabla} h$ will be invariant to our choice of parameterization (whereas the direction defined by ∇h will not be).

By using the smoothly varying positive semi-definite (PSD) matrix F to locally define a metric tensor at every point in parameter space, a Riemannian manifold can be generated over the space of distributions. Note that the associated metric of this space won't be the KL divergence (this isn't even a valid metric), although it will be

[1] Note that this is not a formal "distance" function in the usual sense since it is not symmetric.

[2] This follows from the fact the second order term of the Taylor series of $\mathrm{KL}(P_{x,y}(\theta)\|P_{x,y}(\theta + d))$ is *also* given by $\frac{1}{2}d^\top F d$.

"locally equivalent" to the square root of the KL divergence in the sense that the two will approximate each other within a small neighborhood.

When we use the KL divergence objective function discussed in Section 5.1, the geometric interpretation of the natural gradient becomes particularly nice. This is because the objective function will locally measure distance in distribution space the same way that it is locally measured in the steepest descent interpretation of the natural gradient. In this view, smoothly following the natural gradient is equivalent to following the geodesic path in the Riemannian manifold from the current distribution towards the target distribution (which may never be reached due to the presence of singularities).

5.4 Computational aspects of the natural gradient and connections to the generalized Gauss-Newton matrix

Note that

$$\nabla \log p(y|x,\theta) = J_f^\top \nabla_z \log r(y|z)$$

where J_f is the Jacobian of $f(x,\theta)$ w.r.t. θ, and $\nabla_z \log r(y|z)$ is the gradient of $\log r(y|z)$ w.r.t. z, evaluated at $z = f(x,\theta)$ (r is defined at the end of Section 5.1).

As was first shown by Park et al. (2000), the Fisher information matrix is thus given by

$$\begin{aligned} F &= \mathrm{E}_{Q_x}\left[\mathrm{E}_{P_{y|x}}\left[\nabla \log p(y|x,\theta) \nabla \log p(y|x,\theta)^\top\right]\right] \\ &= \mathrm{E}_{Q_x}[\mathrm{E}_{P_{y|x}}[J_f^\top \nabla_z \log r(y|z) \nabla_z \log r(y|z)^\top J_f]] \\ &= \mathrm{E}_{Q_x}[J_f^\top \, \mathrm{E}_{P_{y|x}}[\nabla_z \log r(y|z) \nabla_z \log r(y|z)^\top] J_f] = \mathrm{E}_{Q_x}[J_f^\top F_R J_f] \end{aligned}$$

where F_R is the Fisher information matrix of the predictive distribution $R_{y|z}$ at $z = f(x,\theta)$.

F_R is given by

$$F_R = \mathrm{E}_{P_{y|x}}[\nabla_z \log r(y|z) \nabla_z \log r(y|z)^\top] = \mathrm{E}_{R_{y|f(x,\theta)}}[\nabla_z \log r(y|z) \nabla_z \log r(y|z)^\top]$$

or,

$$F_R = -\,\mathrm{E}_{R_{y|f(x,\theta)}}[H_{\log r}]$$

where $H_{\log r}$ is the Hessian of $\log r(y|z)$ w.r.t. z, evaluated at $z = f(x,\theta)$.

Note that even if Q_x's density function $q(x)$ is known, and is relatively simple, only for certain choices of $R_{y|z}$ and $f(x,\theta)$ will it be possible to analytically evaluate the expectation w.r.t. Q_x in the above expression for F.

For example, if we take $Q_x = \mathcal{N}(0, I)$, $R_{y|z} = \mathcal{N}(z, \sigma^2)$, and f to be a simple neural network with no hidden units and a single tan-sigmoid output unit, then both F and its inverse can be computed efficiently (Amari, 1998). This situation is exceptional however, and for even slightly more complex models, such as neural networks with one or more hidden layers, it has never been demonstrated how to make such computations feasible in high dimensions.

Fortunately, the situation improves significantly if Q_x is replaced by $\hat{Q}_x$, as this gives

$$F = \mathrm{E}_{\hat{Q}_x}[J_f^\top F_R J_f] = \frac{1}{|S|} \sum_{x \in S} J_f^\top F_R J_f \tag{5.4}$$

which is easy to evaluate when F_R is. Moreover, this is essentially equivalent to the expression in eqn. 3.3 for the generalized Gauss-Newton matrix (GGN), except that we have F_R instead of H_L as the "inner" matrix.

It also suggests a straightforward and efficient way of computing matrix-vector products with F, using an approach similar to the one in Schraudolph (2002) for computing matrix-vector products with the GGN. In particular, one can multiply by J_f using a linearized forward pass, then multiply by F_R (which will be easy if $R_{y|z}$ is sufficiently simple), and then finally multiply by $J_f^\top$ using a standard backwards pass.

As we shall see in the remainder of this section, the connections between the GGN and the Fisher run deeper than just similar expressions and similar algorithms for computing matrix-vector products.

In Park et al. (2000) it was shown that if the density function of $R_{y|z}$ has the form $r(y|z) = \prod_{j=1}^m c(y_j - z_j)$ where $c(a)$ is some univariate density function over $\mathbb{R}$, then F is equal to a re-scaled[3] version of the classical Gauss-Newton matrix for non-linear least squares, with regression function given by f. And in particular, the choice $c(a) = \frac{1}{2}a^2$ turns the learning problem into exactly non-linear least squares, and F into precisely the classical Gauss-Newton matrix.

Heskes (2000) showed that the Fisher and the classical Gauss-Newton matrix are equivalent in the case of the squared error loss and proposed using the Fisher as an approximation to the Hessian in more general contexts.

Concurrently with this work, Pascanu and Bengio (2014) showed that for several common loss functions like cross-entropy and squared error, the GGN and the Fisher are equivalent.

We will show that in fact there is a much more general equivalence between the two matrices, starting from observation that the expressions for the GGN in eqn. 3.3 and the Fisher in eqn. 5.4 are identical up to the equivalence of H_L and F_R.

First, note that $L(y, z)$ may not even be convex in z, and so the GGN won't necessarily be well defined. But even if $L(y, z)$ is convex in z, it won't be true in general that $F_R = H_L$, and so the GGN and the Fisher will differ. However, there is an important class of $R_{y|z}$'s for which $F_R = H_L$ will hold, provided that we have $L(y, z) = -\log r(y|z)$ (putting us in the framework of Section 5.1).

Notice that $F_R = -\mathrm{E}_{R_{y|f(x,\theta)}}[H_{\log r}]$, and $H_L = -H_{\log r}$ (which follows from $L(y, z) = -\log r(y|z)$). Thus, the two matrices being equal is equivalent to the condition

$$\mathrm{E}_{R_{y|f(x,\theta)}}[H_{\log r}] = H_{\log r} \tag{5.5}$$

While this condition may seem arbitrary, it is actually very natural and holds in the important case where $R_{y|z}$ corresponds to an exponential family model with "natural" parameters given by z. That is, when we have

$$\log r(y|z) = z^\top T(y) - \log Z(z)$$

for some function T, where $Z(z)$ is the normalizing constant/partition function. In this case we have $H_{\log r} = -H_{\log Z}$ which doesn't depend on y, and so eqn. 5.5 holds trivially.

Examples of such $R_{y|z}$'s include:

- multivariate normal distributions where z parameterizes only the mean μ

[3] Where the re-scaling constant is determined by properties of $c(a)$.

- multivariate normal distributions where z is the concatenation of $\Sigma^{-1}\mu$ and the vectorization of Σ^{-1}
- multinomial distributions where the softmax of z is the vector of probabilities for each class

Note that the loss function L corresponding to the multivariate normal is the familiar squared error, and the one corresponding to the multinomial distribution is the familiar cross-entropy.

As discussed in Section 3.4, when constructing the GGN one must pay attention to how f and L are defined with regards to what parts of the neural network's computation are performed by each function. For example, the softmax computation performed at the final layer of a classification network is usually considered to be part of the network itself and hence to be part of f. The output $f(x, \theta)$ of this computation are normalized probabilities, which are then fed into a cross-entropy loss of the form $L(y, z) = -\sum_j y_j \log z_j$. But the other way of doing it, which Schraudolph (2002) recommends, is to have the softmax function be part of L instead of f, which results in a GGN which is slightly closer to the Hessian due to "less" of the computational pipeline being linearized before taking the 2nd-order Taylor series approximation. The corresponding loss function is $L(y, z) = -\sum_j y_j z_j + \log(\sum_j \exp(z_j))$ in this case. As we have established above, doing it this way also has the nice side effect of making the GGN equivalent to the Fisher, provided that $R_{y|z}$ is a exponential family model with z as its natural parameters.

This (qualified) equivalence between the Fisher and the GGN suggests how the GGN can be generalized to cases where it might not otherwise be well defined. In particular, it suggests formulating the loss as the negative log density for some distribution and then taking the Fisher of this distribution. Sometimes, this might be as simple as defining $r(y|z) \propto \exp(-L(y, z))$ as per the discussion at the end of Section 5.1.

5.5 Constructing practical natural gradient methods, and the role of damping

Assuming that it is easy to compute, the simplest way to use the natural gradient in optimization is to substitute it in place of the standard gradient within a basic gradient descent approach. This gives the iteration

$$\theta_{k+1} = \theta_k - \alpha_k \tilde{\nabla} h(\theta_k) \tag{5.6}$$

where $\{\alpha_i\}_i$ is a schedule of learning rates.

Choosing the learning rate schedule can be difficult. There are adaptive schemes which are largely heuristic in nature (Amari, 1998) and some non-adaptive prescriptions such as $\alpha_k = c/k$, which have certain theoretical convergence guarantees in the stochastic setting, but which won't necessarily work well in practice. And traditional line-search methods used in deterministic optimization, such as those based on the sufficient decrease condition, won't generally work well in the stochastic setting as we cannot efficiently estimate the objective function to the degree of precision required.

Ideally, we would want to apply the natural gradient method with infinitesimally small steps and produce a smooth idealized path through the space of realizable distributions. But since this is usually impossible in practice, and we don't have access to any other simple description of the class of distributions parameterized by θ that we could work with more directly, we must take non-negligible discrete steps in the given parameter space[4].

[4]In principle, we could move to a much more general class of distributions, such as those given by some non-parametric formulation, where we could work directly with the distributions themselves. But even assuming such an approach would be practical from a computational efficiency standpoint, we would lose the various advantages that we get from working with powerful parametric models like neural networks. In particular, we would lose their ability to generalize to unseen data by modeling the "computational process" which explains the data, instead

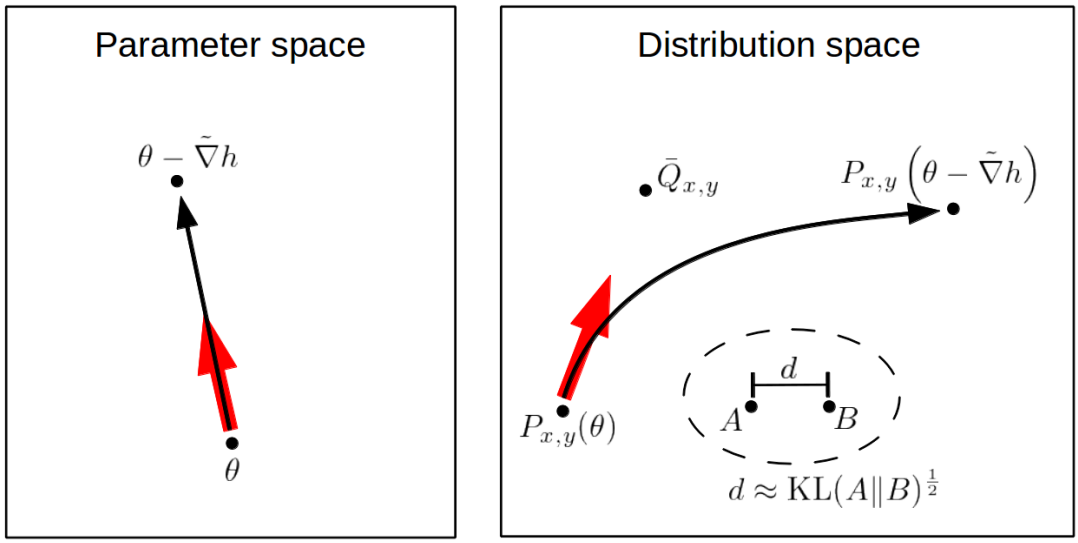

Figure 5.1: A typical situation encountered when performing large discrete updates in the original parameter space. The red arrow is the natural gradient direction (given by the vector $\tilde{\nabla} h$ in parameter space) and the black arrow is the path generated by taking $\theta - \alpha \tilde{\nabla} h$ for $\alpha \in [0, 1]$.

The fundamental problem with simple schemes such as the one in eqn. 5.6 is that they implicitly assume that the natural gradient is a good direction to follow over non-negligible distances in the original parameter space, which will not be true in general. Traveling along a straight line in the original parameter space will not yield a straight line in distribution space, and so the resulting path may instead veer far away from the target that the natural gradient originally pointed towards. This is illustrated in Figure 5.1.

Fortunately, we can exploit the (qualified) equivalence between the Fisher and the GGN in order to produce natural gradient-like updates which will often be appropriate to take with $\alpha_k = 1$. In particular, we know from the discussion in Section 3.4 that the GGN G can serve as a reasonable proxy for the Hessian H of h, and will often produce smaller and more "conservative" updates as it tends to model the curvature as being higher in most directions than the Hessian does. Meanwhile, the update δ produced by minimizing the GGN-based local quadratic model $M(\delta) = \frac{1}{2}\delta^\top G \delta + \nabla h^\top \delta + h(\theta)$ is given by $-G^{-1}\nabla h$, which will be equal to the natural gradient when $F = G$. Thus, the natural gradient, with scaling factor $\alpha = 1$, can be seen as the optimal update according to an approximate, and perhaps slightly conservative, 2nd-order model of h.

But just as in the case of approximate Newton methods, the break-down in the accuracy of the 2nd-order approximation of h, combined with the potential for the natural gradient to be very large (e.g. when F contains some very small eigenvalues), can often lead to very large and very poor update proposals. And simply re-scaling the update by reducing α may be too crude a mechanism to deal with this subtle problem, as it will affect all eigen-directions (of F) equally, including those in which the natural gradient is already sensible or even overly conservative.

Instead, the connection between natural gradient descent and 2nd-order methods suggests the use of some of the various update "damping" techniques that have been developed for the latter, which work by constraining or penalization the solution for δ in various ways during the optimization of $M(\delta)$. Examples include Tikhonov regularization/damping and the closely related trust-region method (e.g. Nocedal and Wright, 2006), and other more sophisticated ones such as the "structural damping" approach of Martens and Sutskever (2011) (Section

of merely using smoothness and locality to generalize.

4.6.4), or the approach present in Krylov Subspace Descent (Vinyals and Povey, 2012). See Section 4.6 for an in-depth discussion of these and other damping techniques.

This idea is well supported by practical experience since, for example, the Hessian-free optimization approach of Martens (2010) generates its updates using an Tikhonov damping scheme applied to the GGN matrix (which in the experiments considered are equivalent to the Fisher), and these updates are used effectively with $\alpha_k = 1$ and make a lot more progress on the objective than optimally re-scaled updates computed without damping (i.e. the raw natural gradient).

5.6 The empirical Fisher

An approximation of the Fisher known as the "empirical Fisher" (denoted $\bar{F}$), which is often used in practical natural gradient methods, is obtained by taking the inner expectation of eqn. 5.2 over the target distribution $Q_{x,y}$ (or its empirical surrogate $\hat{Q}_{x,y}$) instead of the model's distribution $P_{x,y}$.

In the case that one uses $\hat{Q}_{x,y}$, this yields the following simple form:

$$\begin{aligned}\bar{F} &= \mathrm{E}_{\hat{Q}_{x,y}}\left[\nabla \log p(x,y|\theta) \nabla \log p(x,y|\theta)^\top\right] \\ &= \mathrm{E}_{\hat{Q}_x}\left[\mathrm{E}_{\hat{Q}_{y|x}}\left[\nabla \log p(y|x,\theta) \nabla \log p(y|x,\theta)^\top\right]\right] \\ &= \frac{1}{|S|}\sum_{(x,y)\in S} \nabla \log p(y|x,\theta) \nabla \log p(y|x,\theta)^\top\end{aligned}$$

This matrix is often incorrectly referred to as the Fisher, or even the Gauss-Newton, although it is in general not equivalent to those matrices.

5.6.1 Comparisons to the standard Fisher

Like the Fisher F, the empirical Fisher $\bar{F}$ is PSD. But unlike F, it is essentially free to compute, provided that one is already computing the gradient of h. It can also be applied to objective functions which might not involve a probabilistic model in any obvious way.

Compared to F, which is of rank $\leq |S| \operatorname{rank}(F_r)$, $\bar{F}$ has a rank of $\leq |S|$, which can make it easier to work with in practice. For example, the problem of computing the diagonal (or various blocks) is easier for the empirical Fisher than it is for higher rank matrices like the standard Fisher (Martens et al., 2012). This has motivated its use in optimization methods such as TONGA (Le Roux et al., 2008), and as the diagonal preconditioner of choice in the Hessian-free optimization method (Martens, 2010). Interestingly however, there are stochastic estimation methods (Chapelle and Erhan, 2011; Martens et al., 2012) which can be used to efficiently estimate the diagonal (or various blocks) of the standard Fisher F, and these work quite well in practice.

Despite the various advantages of using $\bar{F}$, there are good reasons to use F instead of $\bar{F}$ whenever possible. In addition to Amari's extensive theory developed for the exact natural gradient (which uses F), perhaps the best reason for using F over $\bar{F}$ is that F turns out to be a reasonable approximation to the Hessian H of h in certain important special cases, which is a property that $\bar{F}$ lacks in general.

For example, as discussed in Section 5.2, when the loss is given by $-\log p(y|x)$ (as in Section 5.1), F can be seen as an approximation of H, because both matrices have the interpretation of being the expected Hessian of the loss under some distribution. Due to the similarity of the expression for F in eqn. 5.2 and the one above for $\bar{F}$ it might be tempting to think that $\bar{F}$ is given by the expected Hessian of the loss under $\hat{Q}_{x,y}$ (which is actually

the formula for H) in the same way that F is given by eqn. 5.3, however this is not the case in general.

And as we saw in Section 5.4, given certain assumptions about how the GGN is computed, and some additional assumptions about the form of the loss function L, F turns out to be equivalent to the GGN. This is very useful since the GGN can be used to define a local quadratic approximation of h, whereas F normally doesn't have such an interpretation. Moreover, Schraudolph (2002) and later Martens (2010) compared $\bar{F}$ to the GGN and observed that the latter performed much better as a curvature matrix within various neural network optimization methods.

As concrete evidence for why the empirical Fisher is, at best, a questionable choice for the curvature matrix, consider the following example. We will set $n = 1$, $f(x, \theta) = \theta$, $R_{y|z} = \mathcal{N}(z, 1)$, and $S = \{(0, 0)\}$, so that $h(\theta)$ is a simple convex quadratic function of θ given by $h(\theta) = \frac{1}{2}\theta^2$. In this example we have that $\nabla h = \theta$, $\bar{F} = \theta^2$, while $F = 1$. If we use $\bar{F}^\xi$ as our curvature matrix for some exponent $\frac{1}{2} \leq \xi \leq 1$, then it is easy to see that an iteration of the form

$$\theta_{k+1} = \theta_k - \alpha_k(\bar{F}(\theta_k)^\xi)^{-1}\nabla h(\theta_k) = \theta_k - \alpha_k(\theta_k^2)^{-\xi}\theta_k = (1 - \alpha_k|\theta_k|^{-2\xi})\theta_k$$

will fail to converge to the minimizer ($\theta = 0$) unless $\xi < 1$ and the learning rate α_k goes to 0 sufficiently fast. And even when it does converge, it will only be at a rate comparable to the speed at which α_k goes to 0, which in typical situations will be either $\mathcal{O}(1/k)$ or $\mathcal{O}(1/\sqrt{k})$. Meanwhile, a similar iteration of the form

$$\theta_{k+1} = \theta_k - \alpha_k F^{-1}\nabla h(\theta_k) = \theta_k - \alpha_k\theta_k = (1 - \alpha_k)\theta_k$$

which uses the exact Fisher F as the curvature matrix, will experience very fast linear convergence[5] with rate $|1 - \alpha|$, for any fixed learning rate $\alpha_k = \alpha$ satisfying $0 < \alpha < 2$.

It is important to note that this example uses a noise-free version of the gradient, and that this kind of linear convergence is (provably) impossible in most realistic stochastic/online settings. Nevertheless, we would argue that a highly desirable property of any stochastic optimization method should be that it can, in principle, revert to an optimal (or nearly optimal) behavior in the deterministic setting. This might matter a lot in practice, since the gradient may end up being sufficiently well estimated in earlier stages of optimization from only a small amount of data (which is a common occurrence in our experience), or in later stages provided that larger mini-batches or other variance-reducing procedures are employed (e.g. Le Roux et al., 2012; Johnson and Zhang, 2013).

5.6.2 Recent diagonal methods based on the empirical Fisher

Recently, a spate of stochastic optimization methods have been proposed that are all based on diagonal approximations of the empirical Fisher $\bar{F}$. These include the diagonal version of AdaGrad (Duchi et al., 2011), RMSProp (Tieleman and Hinton, 2012), Adam (Ba and Kingma, 2015), etc. Such methods use iterations of the following form (possibly with some slight modifications):

$$\theta_{k+1} = \theta_k - \alpha_k(B_k + \lambda I)^{-\xi}g_k(\theta_k) \tag{5.7}$$

where the curvature matrix B_k is taken to be a diagonal matrix $\mathrm{diag}(p_k)$ with p_k adapted to maintain some kind of estimate of the diagonal of $\bar{F}$ (possibly using information from previous iterates/mini-batches), $g_k(\theta_k)$ is an estimate of $\nabla h(\theta_k)$ produced from the current mini-batch, α_k is a schedule of learning rates, and $0 < \lambda$ and $0 < \xi \leq 1$ are "fudge factors" (discussed later in this section).

[5]Here we mean "linear" in the classical sense that $|\theta_k - 0| \leq |\theta_0 - 0||1 - \alpha|^k$ and *not* in the sense that $|\theta_k - 0| \in \mathcal{O}(1/k)$

There are also slightly more sophisticated methods (Schaul et al., 2013; Zeiler, 2013) which use preconditioners that combine the diagonal of $\hat{F}$ with other quantities (such as an approximation of the diagonal of the Gauss-Newton/Fisher in the case of Schaul et al. (2013)) in order to correct for how the empirical Fisher doesn't have the right "scale" (which is ultimately the reason why it does poorly in the example given at the end of Section 5.6.1).

A diagonal preconditioner of the form used in eqn. 5.7 was also used to accelerate the CG sub-optimizations performed within HF (Martens, 2010) (see Section 4.9.3). In the context of CG, the improper scale of $\bar{F}$ is not as serious an issue due to the fact that CG is invariant to the overall scale of its preconditioner (since it computes an optimal "learning rate" at each step which automatically adjusts for the scale). However, it still makes more sense to use the diagonal of F as a preconditioner, and thanks to the method proposed by Chapelle and Erhan (2011), this can be estimated efficiently and accurately.

While the idea of using the diagonal of F, $\bar{F}$, or the Gauss-Newton as a preconditioner for stochastic gradient descent (SGD) is sometimes incorrectly attributed to Duchi et al. (2011), it actually goes back *much* earlier, and was likely first applied to neural networks with the work of Lecun and collaborators (Becker and LeCun, 1989; LeCun et al., 1998), who proposed an iteration of the form in eqn. 5.7 with $\xi = 1$ where p_k approximates the diagonal of the Hessian or the Gauss-Newton matrix (which as shown in Section 5.4, is actually equivalent to F for the common squared-error loss).

Following the early pioneering work of Lecun, Amari, and their collaborators, various neural network optimization methods have been developed over the last couple of decades that use diagonal, block-diagonal, low-rank, or Krylov-subspace based approximations of F or $\bar{F}$ as a curvature matrix/preconditioner. In addition to methods based on diagonal approximations already mentioned, some methods based on non-diagonal approximations include the method of Park et al. (2000), TONGA (Le Roux et al., 2008), Natural Newton (Le Roux and Fitzgibbon, 2010), HF (Martens, 2010), KSD (Vinyals and Povey, 2012) and many more.

The idea of computing an estimate of the (empirical) Fisher using a history of previous iterates/mini-batches appeared in various early works. The particular way of doing this advocated by Duchi et al. (2011), which was to use an equally weighted average of all past gradients and which was done in order to make it possible to prove a particular regret bound[6], is – not surprisingly – a poor choice in practice (Tieleman and Hinton, 2012) compared to the older and more intuitive approach of using an exponentially decayed running average (e.g. LeCun et al., 1998; Park et al., 2000), which naturally "forgets" very old contributions to the estimate which are based on stale parameter values.

It is important to observe that the way $\bar{F}$ is estimated can affect the convergence characteristics of an iteration like eqn. 5.7 in subtle and important ways. For example, if $\bar{F}$ is estimated using gradients from previous iterations, and especially if it is the average of *all* past gradients as in AdaGrad, it may shrink sufficiently slowly that the convergence issues seen in the example at the end of Section 5.6.1 are avoided. Moreover, for reasons related to this phenomenon, it seems likely that the proofs of regret bounds in Duchi et al. (2011) and the related work of Hazan et al. (2007) could *not* be modified work if the exact $\bar{F}$, computed only at the current θ, were used. Developing a better understanding of this issue, and the relationship between methods and theories such as AdaGrad developed in the convex optimization literature, and classical stochastic 2nd-order methods and theories (e.g Murata, 1998; Bottou and LeCun, 2005) remains an interesting direction for future research.

The constants λ and ξ present in eqn. 5.7 are often thought of as "fudge factors" designed to correct for the "poor conditioning" (Becker and LeCun, 1989) of the curvature matrix, or to guarantee boundedness of the

[6]The total regret at iteration K is defined as $\sum_{k=1}^{K}(h(\theta_k) - h(\theta^*)$ for the optimal θ^*, and provides a measure of the speed of convergence of an online optimization algorithm which is particularly popular in the convex optimization community.

updates and prevent the optimizer from "blowing up" (LeCun et al., 1998). However, these explanations are severe oversimplifications, at best. A much more compelling and *useful* explanation, at least in the case of λ, comes from viewing the update in eqn. 5.7 as being the minimizer of a local quadratic approximation $M(\delta) = \frac{1}{2}\delta^\top B_k \delta + \nabla h(\theta)^\top \delta + h(\theta)$ to $h(\theta_k + \delta)$, as discussed in Section 5.5. In this view, λ plays the role of a Tikhonov damping parameter (see Section 4.6.1) which is added to B_k in order to ensure that the proposed update stays within a certain radius around zero in which $M(\delta)$ remains a reasonable approximation to $h(\theta + \delta)$. Note that this explanation implies that no single fixed value of λ will be appropriate throughout the entire course of optimization (since the local properties of the objective will change), and so an adaptive adjustment scheme, such as the one present in HF (as described in Section 4.6.5) should be used.

The use of the exponent $\xi = 3/4$ first appeared in HF as part of its diagonal preconditioner for CG, and was justified as a way of making the curvature estimate "more conservative" by making it closer to a multiple of the identity, to compensate for the diagonal approximation being made (among other things). Around the same time, Duchi et al. (2011) proposed to use $\xi = 1/2$ within an update of the form of eqn. 5.7, which was important in proving a certain regret bound both for the diagonal and non-diagonal versions of the method. However, it is noteworthy that while the use of $\xi = 1$ would invalidate this particular bound (or at least its existing proof), it is relatively simple to prove a different bound for the $\xi = 1$ case by a minor modification of the argument given by Duchi et al. (2011) for the $\xi = 1/2$ case (provided one appropriately modifies the schedule for the learning rate α_k so that the updates scale as $1/\sqrt{k}$).

To shed some light on the question of ξ, we can consider the work of Hazan et al. (2007), who like Duchi et al. (2011) developed and analyzed an online approximate Newton method within the framework of online convex optimization. Like the non-diagonal version of AdaGrad, the method proposed by Hazan et al. (2007) uses an estimate of the empirical Fisher $\bar{F}$ computed as the average of gradients from all previous iterations. While impractical for high dimensional problems like any non-diagonal method is (or at least, one that doesn't make some other strong approximation of the curvature matrix), this method achieves a superior upper bound on the regret than Duchi et al. (2011) was able to show for AdaGrad ($\mathcal{O}(\log(k))$ instead of $\mathcal{O}(\sqrt{k})$, where k is the total number of iterations), which was possible in part due to the use of slightly stronger hypotheses about the properties of h (e.g. that for each x and y, $L(y, f(x, \theta))$ is a strongly convex function of θ). Notably, this method uses $\xi = 1$, just as in standard natural gradient descent, which provides support for such a choice, especially since the h used in neural networks usually satisfies these stronger assumptions (at least when ℓ_2 regularization is used).

However, it is important to note that Hazan et al. (2007) also proves a $\mathcal{O}(\log(k))$ bound on the regret for a basic version of SGD, and that what actually differentiates the various methods they analyze is the constant hidden in the big-O notation, which is much larger for the version of SGD they consider than for their approximate Newton method. In particular, the former depends on a quantity which grows with the condition number of the Hessian H at θ^*, while the latter does not, in a way that echos the various analyses performed on stochastic gradient descent and stochastic approximations of Newton's method in the more classical "local-convergence" setting (e.g. Murata, 1998; Bottou and LeCun, 2005).

5.7 Asymptotic convergence speed

5.7.1 Amari's Fisher efficiency result

A property of natural gradient descent which is frequently referenced in the literature is that it is "Fisher efficient". In particular, Amari (1998) showed that an iteration of the form

$$\theta_{k+1} = \theta_k - \alpha_k \tilde{g}_k(\theta_k) \tag{5.8}$$

when applied to an objective of the form discussed in Section 5.1, with α_k shrinking as $1/k$, and with $\tilde{g}_k(\theta_k) = F^{-1} g_k(\theta_k)$ where $g_k(\theta_k)$ is a stochastic estimate of $\nabla h(\theta_k)$ (from a single training case), will produce an estimator θ_k which is asymptotically "Fisher efficient". This means that θ_k will tend to an unbiased estimator of the global optimum θ^*, and that its expected squared error matrix (which is asymptotically equal to its variance) will satisfy

$$\mathrm{E}[(\theta_k - \theta^*)(\theta_k - \theta^*)^\top] = \frac{1}{k} F(\theta^*)^{-1} + \mathcal{O}\left(\frac{1}{k^2}\right) \tag{5.9}$$

which is (asymptotically) the smallest[7] possible variance matrix that any unbiased estimator based k training cases can have, according to the Cramér-Rao lower bound.

This result can also be straightforwardly extended to handle the case where $g_k(\theta_k)$ is computed using a mini-batch of size m (which uses m independently sampled cases at each iteration), in which case the above asymptotic variance bound becomes

$$\frac{1}{mk} F(\theta^*)^{-1} + \mathcal{O}\left(\frac{1}{k^2}\right)$$

which again matches the Cramér-Rao lower bound.

Note that this result, as stated above, applies to the version of natural gradient descent where F is computed using the training distribution $\hat{Q}_x$ (see Section 5.2). If we instead consider the version where F is computed using the true data distribution Q_x, then a similar result will still apply, provided that we sample of x from Q_x and y from $Q_{y|x}$ when computing the stochastic gradient $g_k(\theta_k)$, and that θ^* is defined as the minimum of the idealized objective $\mathrm{KL}(Q_{x,y} \| P_{x,y}(\theta))$ (see Section 5.1).

While this Fisher efficiency result would seem to suggest that natural gradient descent is the best possible optimization method in the stochastic setting, it unfortunately comes with several important caveats, which we discuss below.

Firstly, the proof assumes that the iteration in eqn. 5.8 eventually converges to the global optimum θ^* (at an unspecified speed). While this assumption can be justified when the objective h is convex (provided that α_k is chosen appropriately), it won't be true in general for non-convex objectives, such as those encountered in neural network training. In practice, a reasonable local optimum θ^* might be a good surrogate for the global optimum, in which case a property roughly analogous to asymptotic Fisher efficiency may still hold, at least approximately.

Secondly, it is assumed in Amari's proof that F is computed using the full training distribution $\hat{Q}_x$, which in the case of neural network optimization usually amounts to an entire pass over the training set S. So while the proof allows for the gradient ∇h to be stochastically estimated from a mini-batch, it doesn't allow this for the Fisher F. This is a serious challenge to the idea that (stochastic) natural gradient descent gives an estimator which

[7] With the usual definition of $\leq$ for PSD matrices: $A \leq B$ iff $B - A$ is PSD.

makes optimal use of the training data that it sees. And note that while one can approximate F using samples of x from S, which is a solution that often works well in practice (depending on how much data is used, and whether the estimate can feasibly be "accumulated" across multiple iterations), a Fisher efficiency result like the one proved by Amari (1998) will likely no longer hold. Investigating the manner and degree in which it may hold *approximately* when F is estimated in this way is an interesting direction for future research.

A third issue with Amari's result is that it is given in terms of the convergence of θ_k according to its own (arbitrary) euclidean geometry instead of the arguably more relevant objective function value. Fortunately, it is straightforward to obtain the former from the latter. In particular, by applying Taylor's theorem and using $\nabla h(\theta^*) = 0$ we have

$$\begin{aligned} h(\theta_k) - h(\theta^*) &= \frac{1}{2}(\theta_k - \theta^*)^\top H^*(\theta_k - \theta^*) + \nabla h(\theta^*)^\top(\theta_k - \theta^*) + \mathcal{O}\left((\theta_k - \theta^*)^3\right) \\ &= \frac{1}{2}(\theta_k - \theta^*)^\top H^*(\theta_k - \theta^*) + \mathcal{O}\left((\theta_k - \theta^*)^3\right) \end{aligned} \tag{5.10}$$

where $H^* = H(\theta^*)$ and $\mathcal{O}\left((\theta_k - \theta^*)^3\right)$ is short-hand to mean a function which is cubic in the entries of $\theta_k - \theta^*$. From this it follows that

$$\begin{aligned} \mathrm{E}[h(\theta_k)] - h(\theta^*) &= \frac{1}{2}\,\mathrm{E}\left[(\theta_k - \theta^*)^\top H^*(\theta_k - \theta^*)\right] + \mathrm{E}\left[\mathcal{O}\left((\theta_k - \theta^*)^3\right)\right] \\ &= \frac{1}{2}\,\mathrm{tr}\left(H^*\,\mathrm{E}\left[(\theta_k - \theta^*)(\theta_k - \theta^*)^\top\right]\right) + \mathrm{E}\left[\mathcal{O}\left((\theta_k - \theta^*)^3\right)\right] \\ &= \frac{1}{2k}\,\mathrm{tr}\left(H^* F(\theta^*)^{-1}\right) + \mathrm{E}\left[\mathcal{O}\left((\theta_k - \theta^*)^3\right)\right] = \frac{n}{2k} + o\left(\frac{1}{k}\right) \end{aligned} \tag{5.11}$$

where we have used $H^* = F(\theta^*)$ which follows from the "realizability" hypothesis used to prove the Fisher efficiency result (see below). Note that we have also used $\mathrm{E}\left[\mathcal{O}\left((\theta_k - \theta^*)^3\right)\right] = o(1/k)$, which is an (unjustified) assumption that is used in Amari's proof. This assumption has intuitive appeal since $\mathrm{E}\left[\mathcal{O}\left((\theta_k - \theta^*)^2\right)\right] = \mathcal{O}(1/k)$, and so it makes sense that $\mathrm{E}\left[\mathcal{O}\left((\theta_k - \theta^*)^3\right)\right]$ would shrink faster. However, extreme counterexamples are possible which involve very heavy-tailed distributions on θ_k over unbounded regions. By adding some mild hypotheses such as θ_k being restricted to some bounded region, which is an assumption frequently used in the convex optimization literature, it is possible to justify this assumption rigorously. Rather than linger on this issue we will refer the reader to Bottou and LeCun (2005), which provides a more rigorous treatment of these kind of asymptotic results, using various generalizations of the big-O notation.

Note that while this is the same convergence rate ($\mathcal{O}(1/k)$) as the one which appears in Hazan et al. (2007) (see our Section 5.6), the constant is much better. However, the comparison is slightly unfair, since Hazan et al. (2007) doesn't require that the curvature matrix be estimated on the entire dataset (as discussed above).

The forth and final caveat of Amari's Fisher efficiency result is that Amari's proof assumes that the training distribution $\hat{Q}_{x,y}$ and the optimal model distribution $P_{x,y}(\theta^*)$ coincide, a condition called "realizability" (which is also required in order for the Cramér-Rao lower bound to apply). This means, essentially, that the model perfectly captures the training distribution at $\theta = \theta^*$. This assumption is used in Amari's proof of the Fisher efficiency result to show that the Fisher F, when evaluated at $\theta = \theta^*$, is equal to both the empirical Fisher $\bar{F}$ and the Hessian H of h.

It is not clear from Amari's proof what happens when this correspondence fails to hold at $\theta = \theta^*$, and whether a (perhaps) weaker asymptotic upper bound on the variance might still be provable. Fortunately, various authors (Murata, 1998; Bottou and LeCun, 2005; Bordes et al., 2009) building on the early work of Amari (1967), provide

some further insight into this question by studying asymptotic behavior of general iterations of the form[8]

$$\theta_{k+1} = \theta_k - \alpha_k B_k^{-1} g_k(\theta_k) \tag{5.12}$$

where $B_k = B$ is a fixed[9] curvature matrix (which is independent of θ_k and k), and where $g_k(\theta_k)$ is a stochastic estimate of $\nabla h(\theta_k)$ (which must be unbiased, have finite variance, and have the property that $\{g_i(\theta)\}_i$ are i.i.d. variables).

In particular, Murata (1998) gives exact (although implicit) expressions for the asymptotic mean and variance of θ_k in the above iteration for the case where $\alpha_k = 1/(k+1)$ or α_k is constant, thus generalizing Amari's Fisher efficiency result. These expressions describe the (asymptotic) behavior of this iteration in cases where the curvature matrix B is not the Hessian H or the Fisher F, covering the non-realizable case, as well as the case where the curvature matrix is only an approximation of the Hessian or Fisher. Bordes et al. (2009) meanwhile gives expressions for $\mathrm{E}[h(\theta_k)]$ in the case where α_k shrinks as $1/k$, thus generalizing eqn. 5.11 in a similar manner.

In the following subsection we will examine these results in more depth, and significantly improve on those of Bordes et al. (2009) (at least in the quadratic case) by giving an *exact* asymptotic solution for $\mathrm{E}[h(\theta)]$. To do this we will apply a simple generalization of eqn. 5.11 to Murata's expressions for the asymptotic mean and covariance of θ_k, thus expressing $\mathrm{E}[h(\theta)]$ in terms of the trace of the solution of a certain matrix equation, and then apply some methods from the control theory literature for computing the trace of such solutions.

Some interesting consequences of this analysis are discussed in Sections 5.7.3 and 5.7.7. Among these are the observation that when an annealed learning rate $\alpha_k = 1/(k+1)$ is used, the application of stochastic 2nd-order optimization with $B = H$, while not improving the asymptotic dependency on k of the convergence rate vs SGD, will in realistic scenarios significantly improve the multiplicative constant on the asymptotically dominant $\Omega(1/k)$ term in the expression for $\mathrm{E}[h(\theta)] - h(\theta_0)$. We also show that when a iterate averaging scheme is used with a fixed learning rate, the constant on the $\Omega(1/k)$ term is *not* improved by the application of 2nd-order optimization, while the constant on the $\Omega(1/k^2)$ term improves significantly. We argue that this second term, which is independent of the stochastic gradient noise and instead depends on the initial value of the objective ($h(\theta_0)$), may matter more practice given a limited iteration budget.

5.7.2 Some new results concerning asymptotic convergence speed of general stochastic 2nd-order methods

In this subsection we build on the results of Murata (1998) in order to prove Theorem 6, which is a result that gives detailed expressions for the convergence speed of stochastic 1st and 2nd-order methods based on iterations of the form in eqn. 5.12. Along the way, we will develop techniques for computing the asymptotic mean and covariance of θ_k (as produced by the stochastic iteration in eqn. 5.12) using tools from control theory. Readers only interested in the results themselves may wish to skip directly to the statement of the theorem and the subsequent discussion of consequences, which appear at the end of this subsection.

For an n-dimensional symmetric matrix A we will denote by $\lambda_i(A)$ its i-th largest eigenvalue, so that $\lambda_1(A) \geq$

[8] Note that some authors define B_k to be the matrix that multiplies the gradient, instead of its inverse (as we do instead).

[9] Note that for a non-constant B_k where B_k^{-1} converges sufficiently quickly to a fixed B^{-1} as θ_k converges to θ^*, these analyses will likely still apply, at least approximately.

$\lambda_2(A) \geq \ldots \geq \lambda_n(A)$. Further, we will denote

$$\begin{aligned} V_k &= \mathrm{var}(\theta_k) = \mathrm{cov}(\theta_k, \theta_k) = \mathrm{E}\left[(\theta_k - \mathrm{E}[\theta_k])(\theta_k - \mathrm{E}[\theta_k])^\top\right] \\ \Xi_A(X) &= AX + (AX)^\top \\ \Omega_A(X) &= AXA^\top \\ \Sigma_g(\theta) &= \mathrm{var}(g(\theta)) = \mathrm{cov}(g(\theta), g(\theta)) = \mathrm{E}\left[(g(\theta) - \mathrm{E}[g(\theta)])(g(\theta) - \mathrm{E}[g(\theta)])^\top\right] \end{aligned}$$

where Ξ_A and Ω_A are linear operators on $n \times n$ matrices [10], and $g(\theta)$ denotes a random variable with the same distribution as each of the $g_i(\theta)$'s.

The following theorem summarizes the relevant results of Murata (1998) on the asymptotic behavior of stochastic iterations of the form in eqn. 5.12. Note that we use the symbols defined below in a somewhat inconsistent way from how they are used by Murata (1998) (e.g. "V_∞" has a different scaling).

Theorem 1. *(Adapted from Theorems 1 and 4 of Murata (1998)) Suppose that θ_k is generated by the stochastic iteration in eqn. 5.12 while optimizing a quadratic objective $h(\theta) = \frac{1}{2}(\theta - \theta^*)^\top H^*(\theta - \theta^*)$.*

If $\alpha_k = \alpha$ is constant then we have that

$$\begin{aligned} \mathrm{E}[\theta_k] &= \theta^* + (I - \alpha B^{-1}H^*)^k(\theta_0 - \theta^*) \\ V_k &= \left(I - (I - \Xi_{\alpha B^{-1}H^*})^k\right)(V_\infty) + (I - \Xi_{\alpha B^{-1}H^*})^k\left((\theta_0 - \theta^*)(\theta_0 - \theta^*)^\top\right) \end{aligned}$$

where $V_\infty = \alpha\left(\Xi_{B^{-1}H^*}\right)^{-1}\left(B^{-1}\Sigma_g(\theta^*)B^{-1}\right)$

If on the other hand $\alpha_k = 1/(k+1)$ and $\lambda_n\left(B^{-1}H^*\right) > \frac{1}{2}$ then we have that

$$\begin{aligned} \mathrm{E}[\theta_k] &= \theta^* + \prod_{j=0}^{k-1}\left(I - \alpha_j B^{-1}H^*\right)(\theta_0 - \theta^*) \\ V_k &= \frac{1}{k}\left(\Xi_{B^{-1}H^*} - I\right)^{-1}\left(B^{-1}\Sigma_g(\theta^*)B^{-1}\right) - \frac{1}{k^2}\left(\Xi_{B^{-1}H^*} - 2I\right)^{-1}\left(B^{-1}\Sigma_g(\theta^*)B^{-1}\right) + \mathcal{O}\left(\frac{1}{k^3}\right) \end{aligned}$$

Remark 2. *Theorem 1 deviates from the original presentation of Murata (1998) by assuming that $h(\theta)$ is exactly quadratic. This is done because Murata (1998) proved their more general results in a somewhat non-rigorous way by first assuming this hypothesis, and then appealing to the fact that more general objectives are well-approximated by such a convex quadratic in a close proximity to the local minimum θ^*, as long as they are sufficiently smooth. While their proof made a cursory attempt to rigorously deal with the resulting approximation error, at least in the case where $\alpha_k = 1/(k+1)$, this produced an analysis that was at best flawed but repairable, and at worst wrong. These issues could likely be repaired in the case where $\alpha_k = 1/(k+1)$ with a more careful approach that uses the fact that the error vanishes asymptotically as θ_k converges (both with high probability). However, in the case where $\alpha_k = \alpha$ is constant, the theorem does not strictly hold without assuming that $h(\theta)$ is a convex quadratic, even if we only require that the expressions for the mean and variance are asymptotically accurate. Moreover, without assuming that $h(\theta^*)$ is quadratic, it is unlikely that any closed-form expression could be obtained for the asymptotic covariance V_∞ in this case. See Appendix A.1 for some additional discussion of this issue.*

Remark 3. *We have taken a few additional liberties in interpreting the results in Murata (1998). For example,*

[10] Note that they are *not* $n \times n$ matrices themselves, although they can be represented as $n^2 \times n^2$ matrices using Kronecker product notation. Note that these operators can be linearly combined and composed, where we will use the standard $\pm$ notation for linear combination, and multiplication for composition, where I will be the identity operator. So, for example, $(I + \Xi_A^2)(X) = X + \Xi_A(\Xi_A(X))$.

we give a slightly different result (which can be obtained by a minor modification of the original arguments) where we assume that the covariance of the stochastic gradients, $\Sigma_g(\theta)$, is constant, as opposed to assuming that $\mathrm{E}[g(\theta)g(\theta)^\top] = \Sigma_g(\theta) + \mathrm{E}[g(\theta)]\,\mathrm{E}[g(\theta)]^\top$ is constant as Murata (1998) does[11]*. Note that this change doesn't affect the terms that are dominant (in k) in any of the resulting asymptotic expressions or those which we derive from them, although it does affect the non-dominant terms, and in a way that makes the resulting expressions arguably more accurate. We have also given a more detailed expression for V_k in the $\alpha_k = 1/(k+1)$ case which is accurate up to order $1/k^2$ (instead of just $1/k$). See Appendix A.1 for further details.*

One interesting observation we can immediately make from Theorem 1 is that, at least in the case where the objective is a convex quadratic, $\mathrm{E}[\theta_k]$ progresses in a way that is fully independent of the amount/shape of noise which exists in the estimate of the gradient (which is captured by the $\Sigma_g(\theta^*)$ matrix). Indeed, it proceeds as θ_k itself would in the case of fully deterministic optimization. It is only the variance of θ_k around $\mathrm{E}[\theta_k]$ that depends on the gradient noise.

To see why this happens, note that if $h(\theta)$ is quadratic $\nabla h(\theta)$ will be an affine function of θ, and so

$$\mathrm{E}[g(\theta_k)] = \mathrm{E}[\nabla h(\theta_k)] = \nabla h(\mathrm{E}[\theta_k])$$

and thus, provided that α_k doesn't depend on θ_k in any way (as we are implicitly assuming), we have

$$\mathrm{E}[\theta_{k+1}] = \mathrm{E}[\theta_k - \alpha_k B^{-1} g(\theta_k)] = \mathrm{E}[\theta_k] - \alpha_k B^{-1} \nabla h(\mathrm{E}[\theta_k])$$

which is precisely the deterministic version of eqn. 5.12 (where we treat $\mathrm{E}[\theta_k]$ as the optimized quantity).

While Theorem 1 provides a detailed picture of how well θ^* is estimated by θ_k, it doesn't tell us anything directly about how quickly progress is being made on the objective, which is arguably a much more relevant concern in practice. Fortunately, as observed by Murata (1998), we have the basic identity

$$\begin{aligned}\mathrm{E}\left[(\theta_k - \theta^*)(\theta_k - \theta^*)^\top\right] &= \mathrm{E}\left[(\theta_k - \mathrm{E}[\theta_k])(\theta_k - \mathrm{E}[\theta_k])^\top\right] + \mathrm{E}\left[(\mathrm{E}[\theta_k] - \theta^*)(\mathrm{E}[\theta_k] - \theta^*)^\top\right] \\ &= V_k + (\mathrm{E}[\theta_k] - \theta^*)(\mathrm{E}[\theta_k] - \theta^*)^\top\end{aligned}$$

and so we have

$$\begin{aligned}\mathrm{E}[h(\theta_k)] - h(\theta^*) &= \frac{1}{2}\,\mathrm{tr}\left(H^*\,\mathrm{E}\left[(\theta_k - \theta^*)(\theta_k - \theta^*)^\top\right]\right) \\ &= \frac{1}{2}\,\mathrm{tr}\left(H^*\left(V_k + (\mathrm{E}[\theta_k] - \theta^*)(\mathrm{E}[\theta_k] - \theta^*)^\top\right)\right) \\ &= \frac{1}{2}\,\mathrm{tr}\left(H^* V_k\right) + \frac{1}{2}\,\mathrm{tr}\left(H^*(\mathrm{E}[\theta_k] - \theta^*)(\mathrm{E}[\theta_k] - \theta^*)^\top\right) \qquad (5.13)\end{aligned}$$

which allows us to relate the convergence of the mean of θ_k and the size/shape of its variance to the convergence of $\mathrm{E}[h(\theta_k)]$. In particular, we see that in this simple case where $h(\theta)$ is quadratic, $\mathrm{E}[h(\theta_k)] - h(\theta^*)$ neatly decomposes as the sum of two independent terms that quantify the roles of these respective factors in the convergence of $\mathrm{E}[h(\theta_k)]$ to $h(\theta^*)$.

In the remainder of this section we will use the above expression and Theorem 1 to evaluate and bound the asymptotic convergence of $\mathrm{E}[h(\theta_k)]$. Note that while Murata (1998) gives expressions for this as well, these

[11] Note that Murata (1998) only makes this assumption up to an asymptotically negligible approximation factor, although this isn't dealt with in a completely rigorous way, as per the discussion in Remark 2. Insofar as Theorem 1 can be extended to handle the non-quadratic case, it can likely also be extended to handle a non-constant gradient covariance matrix, provided that said matrix becomes approximately constant sufficiently quickly as θ_k converges to θ^*.

expressions only include the asymptotically dominant terms, and cannot be directly evaluated except in certain special cases (such as when $B = H$).

First we will consider the case where $\alpha_k = \alpha$ is constant.

Note that for any matrix X we have

$$\begin{aligned}
H^{*1/2}\left(I - \Xi_{\alpha B^{-1}H^*}\right)(X)\, H^{*1/2} &= H^{*1/2}\left(X - \alpha B^{-1}H^*X - \alpha X^\top H^* B^{-1}\right) H^{*1/2} \\
&= H^{*1/2}XH^{*1/2} - \alpha H^{*1/2}B^{-1}H^{*1/2}(H^{*1/2}XH^{*1/2}) - \alpha(H^{*1/2}XH^{*1/2})^\top H^{*1/2}B^{-1}H^{*1/2} \\
&= \left(I - \Xi_{\alpha H^{*1/2}B^{-1}H^{*1/2}}\right)(H^{*1/2}XH^{*1/2}) = \left(I - \Xi_C\right)(H^{*1/2}XH^{*1/2})
\end{aligned}$$

where we have defined

$$C = \alpha H^{*1/2}B^{-1}H^{*1/2}$$

Applying this recursively we obtain

$$H^{*1/2}\left(I - \Xi_{\alpha B^{-1}H^*}\right)^k(X)\, H^{*1/2} = \left(I - \Xi_C\right)^k(H^{*1/2}XH^{*1/2})$$

Then using the expression for V_k from Theorem 1 it follows that

$$\begin{aligned}
H^{*1/2}V_kH^{*1/2} &= H^{*1/2}\left(V_\infty - \left(I - \Xi_{\alpha B^{-1}H^*}\right)^k(V_\infty) + \left(I - \Xi_{\alpha B^{-1}H^*}\right)^k\left((\theta_0 - \theta^*)(\theta_0 - \theta^*)^\top\right)\right)H^{*1/2} \\
&= \left(I - \left(I - \Xi_C\right)^k\right)(H^{*1/2}V_\infty H^{*1/2}) + \left(I - \Xi_C\right)^k\left(H^{*1/2}(\theta_0 - \theta^*)(\theta_0 - \theta^*)^\top H^{*1/2}\right)
\end{aligned} \tag{5.14}$$

And thus

$$\begin{aligned}
\frac{1}{2}\operatorname{tr}(H^*V_k) &= \frac{1}{2}\operatorname{tr}\left(H^{*1/2}V_kH^{*1/2}\right) \\
&= \frac{1}{2}\operatorname{tr}\left(H^{*1/2}V_\infty H^{*1/2}\right) - \frac{1}{2}\operatorname{tr}\left(\left(I - \Xi_C\right)^k(H^{*1/2}V_\infty H^{*1/2})\right) \\
&\qquad + \frac{1}{2}\operatorname{tr}\left(\left(I - \Xi_C\right)^k\left(H^{*1/2}(\theta_0 - \theta^*)(\theta_0 - \theta^*)^\top H^{*1/2}\right)\right)
\end{aligned} \tag{5.15}$$

Next, observe that for any matrix X, and any symmetric matrix Y

$$\begin{aligned}
\operatorname{tr}\left(Y(I - \Xi_C)(X)\right) &= \operatorname{tr}(YX - YCX - Y(CX)^\top) = \operatorname{tr}(YX) - \operatorname{tr}(YCX) - \operatorname{tr}(YX^\top C) \\
&= \operatorname{tr}(YX) - \operatorname{tr}(YCX) - \operatorname{tr}((YX^\top C)^\top) = \operatorname{tr}(YX) - \operatorname{tr}(YCX) - \operatorname{tr}(CXY) \\
&= \operatorname{tr}(YX) - \operatorname{tr}(YCX) - \operatorname{tr}(YCX) = \operatorname{tr}(YX - YCX - YCX)) = \operatorname{tr}(Y(I - 2C)X)
\end{aligned}$$

and thus it follows that for any non-negative integer i

$$\operatorname{tr}\left(Y(I - \Xi_C)^i(X)\right) = \operatorname{tr}(Y(I - 2C)^iX) \tag{5.16}$$

Noting that the eigenvalues of a product of square matrices is invariant under cyclic permutation of those matrices we have $2\lambda_1(C) = 2\lambda_1(\alpha H^{*1/2}B^{-1}H^{*1/2}) = 2\alpha\lambda_1(B^{-1}H^*) < 1$ (so that $I - 2C$ is PSD), and it thus follows that $\lambda_i((I - 2C)^k) = (1 - 2\lambda_{n-i+1}(C))^k$. Then assuming X is also PSD we can use Lemma 4 (given

below) to get

$$(1 - 2\lambda_1(C))^k \operatorname{tr}(X) \leq \operatorname{tr}((I - 2C)^k X) \leq (1 - 2\lambda_n(C))^k \operatorname{tr}(X)$$

Lemma 4. *(Adapted from Lemma 1 from Wang et al. (1986)) Suppose X and S are $n \times n$ matrices such that S is symmetric and X is PSD. Then we have*

$$\lambda_n(S) \operatorname{tr}(X) \leq \operatorname{tr}(SX) \leq \lambda_1(S) \operatorname{tr}(X)$$

Applying this to eqn. 5.15 we thus have the upper bound

$$\begin{aligned}\frac{1}{2}\operatorname{tr}\left(H^* V_k\right) &\leq \frac{1}{2}\operatorname{tr}\left(H^* V_\infty\right) - \left(1 - 2\alpha\lambda_1\left(B^{-1}H^*\right)\right)^k \frac{1}{2}\operatorname{tr}(H^* V_\infty) + \left(1 - 2\alpha\lambda_n\left(B^{-1}H^*\right)\right)^k h(\theta_0) \\ &= \left[1 - \left(1 - 2\alpha\lambda_1\left(B^{-1}H^*\right)\right)^k\right] \frac{1}{2}\operatorname{tr}\left(H^* V_\infty\right) + \left(1 - 2\alpha\lambda_n\left(B^{-1}H^*\right)\right)^k h(\theta_0) \qquad (5.17)\end{aligned}$$

and the lower bound

$$\begin{aligned}\frac{1}{2}\operatorname{tr}\left(H^* V_k\right) &\geq \frac{1}{2}\operatorname{tr}\left(H^* V_\infty\right) - \left(1 - 2\alpha\lambda_n\left(B^{-1}H^*\right)\right)^k \frac{1}{2}\operatorname{tr}\left(H^* V_\infty\right) + \left(1 - 2\alpha\lambda_1\left(B^{-1}H^*\right)\right)^k h(\theta_0) \\ &= \left[1 - \left(1 - 2\alpha\lambda_n\left(B^{-1}H^*\right)\right)^k\right] \frac{1}{2}\operatorname{tr}\left(H^* V_\infty\right) + \left(1 - 2\alpha\lambda_1\left(B^{-1}H^*\right)\right)^k h(\theta_0) \qquad (5.18)\end{aligned}$$

where we have used that

$$\begin{aligned}\frac{1}{2}\operatorname{tr}\left(H^{*1/2}(\theta_0 - \theta^*)(\theta_0 - \theta^*)^\top H^{*1/2}\right) &= \operatorname{tr}\left((\theta_0 - \theta^*)^\top H^*(\theta_0 - \theta^*)\right) \\ &= \frac{1}{2}(\theta_0 - \theta^*)^\top H^*(\theta_0 - \theta^*) = h(\theta_0)\end{aligned}$$

Next we will compute/bound the term $\operatorname{tr}\left(H^*(\mathrm{E}[\theta_k] - \theta^*)(\mathrm{E}[\theta_k] - \theta^*)^\top\right)$.

Theorem 1 tells us that

$$\mathrm{E}[\theta_k] - \theta^* = (I - \alpha B^{-1}H^*)^k(\theta_0 - \theta^*)$$

Then observing

$$H^{*1/2}(I - \alpha B^{-1}H^*) = \left(I - \alpha H^{*1/2}B^{-1}H^{*1/2}\right) H^{*1/2} = (I - C)\, H^{*1/2}$$

it follows that

$$\begin{aligned}H^{*1/2}(\mathrm{E}[\theta_k] - \theta^*) &= H^{*1/2}(I - \alpha B^{-1}H^*)^k(\theta_0 - \theta^*) \\ &= (I - C)^k\, H^{*1/2}(\theta_0 - \theta^*) \qquad (5.19)\end{aligned}$$

and thus we have

$$\begin{aligned}\frac{1}{2}\operatorname{tr}\left(H^*(\mathrm{E}[\theta_k] - \theta^*)(\mathrm{E}[\theta_k] - \theta^*)^\top\right) &= \frac{1}{2}\operatorname{tr}\left((I - C)^k\, H^{*1/2}(\theta_0 - \theta^*)(\theta_0 - \theta^*)^\top H^{*1/2}(I - C)^{k\top}\right) \\ &= \frac{1}{2}\operatorname{tr}\left((I - C)^{2k}\left(H^{*1/2}(\theta_0 - \theta^*)(\theta_0 - \theta^*)^\top H^{*1/2}\right)\right)\end{aligned}$$

Applying Lemma 4 in a similar manner to before we have the upper bound

$$\frac{1}{2}\operatorname{tr}\left(H^*(\mathrm{E}[\theta_k]-\theta^*)(\mathrm{E}[\theta_k]-\theta^*)^\top\right) \leq \left(1-\alpha\lambda_n\left(B^{-1}H^*\right)\right)^{2k} h(\theta_0) \tag{5.20}$$

and the lower bound

$$\frac{1}{2}\operatorname{tr}\left(H^*(\mathrm{E}[\theta_k]-\theta^*)(\mathrm{E}[\theta_k]-\theta^*)^\top\right) \geq \left(1-\alpha\lambda_1\left(B^{-1}H^*\right)\right)^{2k} h(\theta_0) \tag{5.21}$$

Combining eqn. 5.13, eqn. 5.17, eqn. 5.18, eqn. 5.20, and eqn. 5.21 we thus have

$$L(k) \leq \mathrm{E}[h(\theta_k)] - h(\theta^*) \leq U(k) \tag{5.22}$$

where

$$U(k) = \left[1-(1-2\epsilon_1)^k\right]\frac{1}{2}\operatorname{tr}\left(H^*V_\infty\right) + (1-2\epsilon_2)^k\, h(\theta_0) + (1-\epsilon_2)^{2k}\, h(\theta_0)$$

and

$$L(k) = \left[1-(1-2\epsilon_2)^k\right]\frac{1}{2}\operatorname{tr}\left(H^*V_\infty\right) + (1-2\epsilon_1)^k\, h(\theta_0) + (1-\epsilon_1)^{2k}\, h(\theta_0)$$

with $\epsilon_1 = \alpha\lambda_1\left(B^{-1}H^*\right)$ and $\epsilon_2 = \alpha\lambda_n\left(B^{-1}H^*\right)$.

It remains to compute $\frac{1}{2}\operatorname{tr}\left(H^*V_\infty\right)$.

From Theorem 1, V_∞ is given by $\alpha\Xi^{-1}_{B^{-1}H^*}(B^{-1}\Sigma_g(\theta^*)B^{-1})$, so that we have

$$\Xi_{B^{-1}H^*}(V_\infty) = \alpha B^{-1}\Sigma_g(\theta^*)B^{-1}$$

Written as a matrix equation this is

$$B^{-1}H^*V_\infty + V_\infty H^* B^{-1} = \alpha B^{-1}\Sigma_g(\theta^*)B^{-1} \tag{5.23}$$

Defining

$$\begin{aligned} A &= -B^{-1}H^* \\ P &= V_\infty \\ Q &= \alpha B^{-1}\Sigma_g(\theta^*)B^{-1} \end{aligned} \tag{5.24}$$

we can write this as

$$-A^\top P - PA = Q$$

which after rearrangement becomes

$$A^\top P + PA + Q = 0$$

and is known in the control theory literature as a continuous algebraic Lyapunov equation (CALE) whenever Q is PSD (as it is in our case). The control theory community has developed efficient algorithms for solving such

equations for P (e.g Bartels and Stewart, 1972).

Because the process for solving a CALE is somewhat opaque, it may be useful to bound the required traces with simpler expressions which are easier to understand and reason about. Fortunately, the control theory community has developed many such upper and lower bounds for various functions of P, including $\mathrm{tr}(P)$. See Kwon et al. (1996) for a good, although somewhat dated, review of these.

A pair of upper and lower bounds due to Wang et al. (1986), which are straightforward consequences of Lemma 4, are

$$\frac{\mathrm{tr}(Q)}{-\lambda_n(A+A^\top)} \leq \mathrm{tr}(P) \leq \frac{\mathrm{tr}(Q)}{-\lambda_1(A+A^\top)} \tag{5.25}$$

A more difficult and tighter upper bound due to Komaroff (1992) is

$$\mathrm{tr}(P) \leq \sum_{i=1}^{n} \frac{\lambda_i(Q)}{-\lambda_i(A+A^\top)} \tag{5.26}$$

And a lower bound due to Komaroff (1988), which is incomparable to the one in eqn. 5.25, is

$$\frac{\left(\sum_{i=1}^{n} \lambda_i(Q)^{1/2}\right)^2}{-\mathrm{tr}(A+A^\top)} \leq \mathrm{tr}(P) \tag{5.27}$$

Note that all of these bounds apply only in the case where $A + A^\top$ is negative definite.

As we are interested in $\mathrm{tr}(H^* V_\infty)$ and not just $\mathrm{tr}(V_\infty)$ we cannot directly apply these bounds to $\mathrm{tr}(P)$. Fortunately there are several ways around this issue.

By pre and post-multiplying both sides of eqn. 5.23 by $H^{*1/2}$ it becomes

$$(H^{*1/2}B^{-1}H^{*1/2})(H^{*1/2}V_\infty H^{*1/2}) + (H^{*1/2}V_\infty H^{*1/2})(H^{*1/2}B^{-1}H^{*1/2}) = \alpha H^{*1/2}B^{-1}\Sigma_g(\theta^*)B^{-1}H^{*1/2}$$

Then defining

$$\begin{aligned} A &= -H^{*1/2}B^{-1}H^{*1/2} \\ P &= H^{*1/2}V_\infty H^{*1/2} \\ Q &= \alpha H^{*1/2}B^{-1}\Sigma_g(\theta^*)B^{-1}H^{*1/2} \end{aligned} \tag{5.28}$$

and performing some simple rearrangement we get

$$A^\top P + PA + Q = 0$$

which is a CALE. This is useful in our present case since

$$\mathrm{tr}(P) = \mathrm{tr}\left(H^{*1/2}V_\infty H^{*1/2}\right) = \mathrm{tr}(H^{*1/2}H^{*1/2}V_\infty) = \mathrm{tr}(H^* V_\infty)$$

and thus we can bound $\mathrm{tr}(H^* V_\infty) = \mathrm{tr}(P)$ directly using any of eqn. 5.25, eqn. 5.26, and eqn. 5.27.

However, it turns out that in our particular case we can do better by taking better advantage of the special structure of eqn. 5.23 that isn't shared by general CALEs. To this end we will define a "pseudo-CALE" to be

matrix equations of the form

$$AP + P^\top A + Q = 0 \tag{5.29}$$

where A is invertible (with no restrictions on Q or P). The following lemma describes a useful property of pseudo-CALEs which we will exploit:

Lemma 5. *Suppose $AP + P^\top A + Q = 0$ is a pseudo-CALE. Then we have*

$$\operatorname{tr}(P) = -\frac{1}{2}\operatorname{tr}(A^{-1}Q)$$

Proof. Pre-multiplying both sides of $AP+P^\top A+Q = 0$ by A^{-1} and taking the trace yields $\operatorname{tr}(P)+\operatorname{tr}(A^{-1}P^\top A)+\operatorname{tr}(A^{-1}Q) = 0$. Then nothing that $\operatorname{tr}(A^{-1}P^\top A) = \operatorname{tr}(AA^{-1}P^\top) = \operatorname{tr}(P^\top) = \operatorname{tr}(P)$ we have the $2\operatorname{tr}(P) + \operatorname{tr}(A^{-1}Q) = 0$ and so the result follows.

□

To apply this result we observe that after simple rearrangement eqn. 5.23 can be written as a pseudo-CALE $AP + P^\top A + Q = 0$ where

$$\begin{aligned} A &= -B^{-1} \\ P &= H^* V_\infty \\ Q &= \alpha B^{-1}\Sigma_g(\theta^*)B^{-1} \end{aligned} \tag{5.30}$$

Thus applying Lemma 5 we have

$$\operatorname{tr}(H^*V_\infty) = \operatorname{tr}(P) = -\frac{1}{2}\operatorname{tr}(A^{-1}Q) = -\frac{1}{2}\operatorname{tr}\left((-B^{-1})^{-1}\alpha B^{-1}\Sigma_g(\theta^*)B^{-1}\right) = \frac{\alpha}{2}\operatorname{tr}\left(B^{-1}\Sigma_g(\theta^*)\right) \tag{5.31}$$

Next we will examine the second case considered in Theorem 1, where $\alpha_k = 1/(k+1)$ and $\lambda_n\left(B^{-1}H^*\right) > 1/2$. From eqn. 5.13 we have

$$\mathrm{E}[h(\theta_k)] - h(\theta^*) = \frac{1}{2}\operatorname{tr}\left(H^*V_k\right) + \frac{1}{2}\operatorname{tr}\left(H^*(\mathrm{E}[\theta_k]-\theta^*)(\mathrm{E}[\theta_k]-\theta^*)^\top\right) \tag{5.32}$$

And by the expression for V_k from Theorem 1 we have that

$$\frac{1}{2}\operatorname{tr}\left(H^*V_k\right) = \frac{1}{2}\operatorname{tr}\left(H^*\left(\frac{1}{k}C_1 + \frac{1}{k^2}C_2 + \mathcal{O}\left(\frac{1}{k^3}\right)\right)\right) = \frac{1}{2k}\operatorname{tr}\left(H^*C_1\right) + \frac{1}{2k^2}\operatorname{tr}\left(H^*C_2\right) + \mathcal{O}\left(\frac{1}{k^3}\right) \tag{5.33}$$

where $C_1 = \left(\Xi_{B^{-1}H^*} - I\right)^{-1}\left(B^{-1}\Sigma_g(\theta^*)B^{-1}\right)$ and $C_2 = -\left(\Xi_{B^{-1}H^*} - 2I\right)^{-1}\left(B^{-1}\Sigma_g(\theta^*)B^{-1}\right)$.

Since $\left(\Xi_{B^{-1}H^*} - I\right)(C_1) = B^{-1}\Sigma_g(\theta^*)B^{-1}$ we have that C_1 is given by the matrix equation

$$B^{-1}H^*C_1 + C_1H^*B^{-1} - C_1 = B^{-1}\Sigma_g(\theta^*)B^{-1} \tag{5.34}$$

In order to compute $\mathrm{tr}(H^*C_1)$ we will rewrite eqn. 5.34 as

$$\left(B^{-1} - \frac{1}{2}H^{*-1}\right) H^*C_1 + C_1H^* \left(B^{-1} - \frac{1}{2}H^{*-1}\right) = B^{-1}\Sigma_g(\theta^*)B^{-1} \tag{5.35}$$

which after simple rearrangement can be written as $AP + P^\top A + Q = 0$ where

$$\begin{aligned} A &= -\left(B^{-1} - \frac{1}{2}H^{*-1}\right) \\ P &= H^*C_1 \\ Q &= B^{-1}\Sigma_g(\theta^*)B^{-1} \end{aligned} \tag{5.36}$$

and then apply Lemma 5.

However we must first verify that our A is invertible. To this end we will show that $B^{-1} - \frac{1}{2}H^{*-1}$ is positive definite. This is equivalent to the condition that $H^{*1/2}(B^{-1} - \frac{1}{2}H^{*-1})H^{*1/2} = H^{*1/2}B^{-1}H^{*1/2} - \frac{1}{2}I$ is positive definite, or in other words that $\lambda_n(H^{*1/2}B^{-1}H^{*1/2}) = \lambda_n(H^*B^{-1}) > 1/2$, which is true by hypothesis.

Thus Lemma 5 is applicable, and yields

$$\begin{aligned} \mathrm{tr}(H^*C_1) = \mathrm{tr}(P) = -\frac{1}{2}\,\mathrm{tr}(A^{-1}Q) &= \frac{1}{2}\,\mathrm{tr}\left(\left(B^{-1} - \frac{1}{2}H^{*-1}\right)^{-1} B^{-1}\Sigma_g(\theta^*)B^{-1}\right) \\ &= \frac{1}{2}\,\mathrm{tr}\left(\left(I - \frac{1}{2}B^{1/2}H^{*-1}B^{1/2}\right)^{-1} B^{-1/2}\Sigma_g(\theta^*)B^{-1/2}\right) \end{aligned} \tag{5.37}$$

To compute $\mathrm{tr}(H^*C_2)$, we observe

$$C_2 = -\left(\Xi_{B^{-1}H^*} - 2I\right)^{-1}(B^{-1}\Sigma_g(\theta^*)B^{-1}) = -\frac{1}{2}\left(\Xi_{B^{-1}\frac{1}{2}H^*} - I\right)^{-1}(B^{-1}\Sigma_g(\theta^*)B^{-1})$$

and so we may adapt our previous analysis with H^* replaced by $\frac{1}{2}H^*$, which yields

$$\mathrm{tr}(H^*C_2) = -\frac{1}{4}\,\mathrm{tr}\left(\left(I - \frac{1}{4}B^{1/2}H^{*-1}B^{1/2}\right)^{-1} B^{-1/2}\Sigma_g(\theta^*)B^{-1/2}\right) \tag{5.38}$$

It remains to compute/bound the term $\mathrm{tr}\left(H^*(\mathrm{E}[\theta_k] - \theta^*)(\mathrm{E}[\theta_k] - \theta^*)^\top\right)$. From Theorem 1 we have

$$\mathrm{E}[\theta_k] - \theta^* = \prod_{j=0}^{k-1}\left(I - \alpha_j B^{-1}H^*\right)(\theta_0 - \theta^*)$$

Observing

$$H^{*1/2}(I - \alpha_i B^{-1}H^*) = (I - \alpha_i H^{*1/2}B^{-1}H^{*1/2})H^{*1/2}$$

it follows that

$$\begin{aligned} H^{*1/2}(\mathrm{E}[\theta_k] - \theta^*) &= H^{*1/2} \prod_{j=0}^{k-1} \left(I - \alpha_j B^{-1} H^*\right)(\theta_0 - \theta^*) \\ &= \prod_{j=0}^{k-1} \left(I - \alpha_j H^{*1/2} B^{-1} H^{*1/2}\right) H^{*1/2}(\theta_0 - \theta^*) \\ &= \psi_k \left(H^{*1/2} B^{-1} H^{*1/2}\right) H^{*1/2}(\theta_0 - \theta^*) \end{aligned}$$

where ψ_k is a polynomial defined by

$$\psi_k(x) = \prod_{j=0}^{k-1} (1 - \alpha_j x) = \prod_{j=0}^{k-1} \left(1 - \frac{x}{j+1}\right)$$

As argued by Murata (1998) (in the discussion after their Theorem 4), we have $\psi_k(x) = \mathcal{O}(1/k^x)$. Then recalling the fact that eigenvalues of $\psi_k(X)$ for any matrix X are given by $\{\psi_k(\lambda_i(X))\}_i$, it follows that

$$\lambda_1 \left(\psi_k \left(H^{*1/2} B^{-1} H^{*1/2}\right)\right) = \mathcal{O}\left(\frac{1}{k^{\lambda_n(B^{-1}H^*)}}\right)$$

Thus we have by Lemma 4 that

$$\begin{aligned} \frac{1}{2} \operatorname{tr}\left(H^*(\mathrm{E}[\theta_k] - \theta^*)(\mathrm{E}[\theta_k] - \theta^*)^\top\right) &= \frac{1}{2} \operatorname{tr}\left(H^{*1/2}(\mathrm{E}[\theta_k] - \theta^*)(\mathrm{E}[\theta_k] - \theta^*)^\top H^{*1/2}\right) \\ &= \frac{1}{2} \operatorname{tr}\left(\psi_k \left(H^{*1/2} B^{-1} H^{*1/2}\right)^2 \left(H^{*1/2}(\theta_0 - \theta^*)(\theta_0 - \theta^*)^\top H^{*1/2}\right)\right) \\ &\leq \lambda_1 \left(\psi_k \left(H^{*1/2} B^{-1} H^{*1/2}\right)\right)^2 \frac{1}{2} \operatorname{tr}\left(H^{*1/2}(\theta_0 - \theta^*)(\theta_0 - \theta^*)^\top H^{*1/2}\right) \\ &= \mathcal{O}\left(\frac{1}{k^{\lambda_n(B^{-1}H^*)}}\right)^2 h(\theta_0) = \mathcal{O}\left(\frac{h(\theta_0)}{k^{2\lambda_n(B^{-1}H^*)}}\right) \end{aligned}$$

Combining eqn. 5.32, eqn. 5.33, eqn. 5.37, eqn. 5.38 and the above asymptotic expression we thus have

$$\begin{aligned} \mathrm{E}[h(\theta_k)] - h(\theta^*) = \frac{1}{4k} \operatorname{tr}&\left(\left(I - \frac{1}{2} B^{1/2} H^{*-1} B^{1/2}\right)^{-1} B^{-1/2} \Sigma_g(\theta^*) B^{-1/2}\right) \\ - \frac{1}{8k^2} \operatorname{tr}&\left(\left(I - \frac{1}{4} B^{1/2} H^{*-1} B^{1/2}\right)^{-1} B^{-1/2} \Sigma_g(\theta^*) B^{-1/2}\right) + \mathcal{O}\left(\frac{h(\theta_0)}{k^{2\lambda_n(B^{-1}H^*)}}\right) + \mathcal{O}\left(\frac{1}{k^3}\right) \end{aligned}$$

The following theorem summarizes the results of this subsection:

Theorem 6. *Suppose that θ_k is generated by the stochastic iteration in eqn. 5.12 while optimizing a quadratic objective $h(\theta) = \frac{1}{2}(\theta - \theta^*)^\top H^*(\theta - \theta^*)$.*

If $\alpha_k = \alpha$ is constant and $2\alpha\lambda_1(B^{-1}H^*) < 1$, then we have

$$L(k) \leq \mathrm{E}[h(\theta_k)] - h(\theta^*) \leq U(k)$$

where

$$U(k) = \left[1 - (1 - 2\epsilon_1)^k\right] \frac{\alpha}{4} \operatorname{tr}\left(B^{-1}\Sigma_g(\theta^*)\right) + (1 - 2\epsilon_2)^k \, h(\theta_0) + (1 - \epsilon_2)^{2k} \, h(\theta_0)$$

and

$$L(k) = \left[1 - (1 - 2\epsilon_2)^k\right] \frac{\alpha}{4} \operatorname{tr}\left(B^{-1}\Sigma_g(\theta^*)\right) + (1 - 2\epsilon_1)^k \, h(\theta_0) + (1 - \epsilon_1)^{2k} \, h(\theta_0)$$

with $\epsilon_1 = \alpha\lambda_1\left(B^{-1}H^*\right)$ *and* $\epsilon_2 = \alpha\lambda_n\left(B^{-1}H^*\right)$.

If on the other hand $\alpha_k = 1/(k+1)$ *and* $\lambda_n\left(B^{-1}H^*\right) > 1/2$ *then we have*

$$\begin{aligned}\mathrm{E}[h(\theta_k)] - h(\theta^*) = &\frac{1}{4k} \operatorname{tr}\left(\left(I - \frac{1}{2}B^{1/2}H^{*-1}B^{1/2}\right)^{-1} B^{-1/2}\Sigma_g(\theta^*)B^{-1/2}\right) \\ &- \frac{1}{8k^2} \operatorname{tr}\left(\left(I - \frac{1}{4}B^{1/2}H^{*-1}B^{1/2}\right)^{-1} B^{-1/2}\Sigma_g(\theta^*)B^{-1/2}\right) + \mathcal{O}\left(\frac{h(\theta_0)}{k^{2\lambda_n(B^{-1}H^*)}}\right) + \mathcal{O}\left(\frac{1}{k^3}\right)\end{aligned}$$

Remark 7. *As with the theorem (Theorem 1) on which it is based, the above theorem can likely be extended to handle non-quadratic objectives (at least in the case where* $\alpha_k = 1/(k+1)$*).*

5.7.3 Consequences of Theorem 6

In the case of a fixed learning rate $\alpha_k = \alpha$, Theorem 6 shows that $\mathrm{E}[h(\theta_k)]$ will tend to $h(\theta^*) + \frac{\alpha}{4} \operatorname{tr}\left(B^{-1}\Sigma_g(\theta^*)\right)$. The size of this extra additive factor is correlated with the learning rate α and gradient noise covariance $\Sigma_g(\theta^*)$, and inversely correlated with the size of B. Thus, if the covariance is relatively small compared to the learning rate, this factor may not be very large in practice.

Moreover, one can use the fact that the θ_k's are (dependent) asymptotically unbiased estimators of θ^* to produce an asymptotically unbiased estimator with shrinking variance by averaging them together, as is done in the averaging method (e.g. Polyak and Juditsky, 1992), which we analyze in Section 5.7.6.

In the scenario where $\alpha_k = 1/(k+1)$, if one performs a stochastic 2nd-order optimization with $B = H^*$, Theorem 6 gives that

$$\mathrm{E}[h(\theta_k)] - h(\theta^*) = \left(\frac{1}{2k} - \frac{1}{6k^2}\right) \operatorname{tr}\left(H^{*-1}\Sigma_g(\theta^*)\right) + \mathcal{O}\left(\frac{h(\theta_0)}{k^2}\right)$$

And if one considers the scenario corresponding to 1st-order optimization where we take $B = \beta I$ for $\beta < 2\lambda_n(H^*)$ (so that the condition $\lambda_n\left(B^{-1}H^*\right) > 1/2$ holds), we get

$$\begin{aligned}\mathrm{E}[h(\theta_k)] - h(\theta^*) = &\frac{1}{4k\beta} \operatorname{tr}\left(\left(I - \frac{\beta}{2}H^{*-1}\right)^{-1} \Sigma_g(\theta^*)\right) \\ &- \frac{1}{8k^2\beta} \operatorname{tr}\left(\left(I - \frac{\beta}{4}H^{*-1}\right)^{-1} \Sigma_g(\theta^*)\right) + \mathcal{O}\left(\frac{h(\theta_0)}{k^{2\lambda_n(H^*)/\beta}}\right) + \mathcal{O}\left(\frac{1}{k^3}\right)\end{aligned}$$

For $\beta = \lambda_n(H^*)$, which is the lowest value we can choose while ensuring that the starting-point dependent term

$\mathcal{O}(h(\theta_0)/k^{2\lambda_n(H^*)/\beta})$ shrinks as $1/k^2$, we obtain the upper bound

$$\mathrm{E}[h(\theta_k)] - h(\theta^*) \leq \frac{1}{4k\lambda_n(H^*)} \operatorname{tr}\left(\left(I - \frac{\lambda_n(H^*)}{2} H^{*-1}\right)^{-1} \Sigma_g(\theta^*)\right) + \mathcal{O}\left(\frac{h(\theta_0)}{k^2}\right)$$

While the starting-point dependent terms (which are noise independent) are the same in either scenario (the hidden constant is the same too), the noise-dependent terms, which are the ones asymptotically dominant in k, differ. To compare the size of these terms we can apply Lemma 4 to obtain the following bounds (see Appendix A.2):

$$\frac{1}{2k\lambda_1\left(H^*\right)} \operatorname{tr}(\Sigma_g(\theta^*)) \leq \frac{1}{2k} \operatorname{tr}\left(H^{*-1}\Sigma_g(\theta^*)\right) \leq \frac{1}{2k\lambda_n\left(H^*\right)} \operatorname{tr}(\Sigma_g(\theta^*))$$

and

$$\frac{1}{4k\lambda_n(H^*)} \operatorname{tr}(\Sigma_g(\theta^*)) \leq \frac{1}{4k\lambda_n(H^*)} \operatorname{tr}\left(\left(I - \frac{\lambda_n(H^*)}{2} H^{*-1}\right)^{-1} \Sigma_g(\theta^*)\right) \leq \frac{1}{2k\lambda_n(H^*)} \operatorname{tr}(\Sigma_g(\theta^*))$$

So while in the worst case the noise-dependent terms are closely comparable between the two scenarios, in the $B = H^*$ scenario the term has the *potential* to be much smaller, due to the much smaller lower bound. A necessary condition for this to happen is that H^* is ill-conditioned (so that $\lambda_1\left(H^*\right) \gg \lambda_n\left(H^*\right)$), although this alone is not sufficient.

To actually provide an example where the noise-dependent term is smaller in the $B = H^*$ scenario we must make further assumptions about the nature of the gradient noise covariance matrix $\Sigma_g(\theta^*)$. As an important example, we consider the case where the stochastic gradients are computed using randomly sampled training cases from S (so that $H^* = \Sigma_g(\theta^*)$), and where we are in the realizable regime (so that $H^* = \Sigma_g(\theta^*)$, see Section 5.7.1). In this case we have that in the $B = H^*$ scenario the noise dependent term is

$$\frac{1}{2k} \operatorname{tr}\left(H^{*-1}\Sigma_g(\theta^*)\right) = \frac{1}{2k} \operatorname{tr}\left(H^{*-1}H^*\right) = \frac{n}{2k}$$

while in the $B = \lambda_n(H^*)I$ scenario it is

$$\begin{aligned}\frac{1}{4k\lambda_n(H^*)} \operatorname{tr}\left(\left(I - \frac{\lambda_n(H^*)}{2} H^{*-1}\right)^{-1} \Sigma_g(\theta^*)\right) &= \frac{1}{4k\lambda_n(H^*)} \operatorname{tr}\left(\left(I - \frac{\lambda_n(H^*)}{2} H^{*-1}\right)^{-1} H^*\right) \\ &= \frac{1}{4k\lambda_n(H^*)} \sum_{i=1}^{n} \frac{\lambda_i(H^*)}{1 - \frac{\lambda_n(H^*)}{2\lambda_i(H^*)}} = \frac{1}{4k} \sum_{i=1}^{n} \frac{r_i}{1 - \frac{1}{2r_i}}\end{aligned}$$

where we have defined $r_i = \lambda_i(H^*)/\lambda_n(H^*)$, and to go from the first to the second lines we have used the fact that all matrices involved have the same eigenvectors. Observing that $1 \leq r_i$ we have $r_i \leq r_i/(1 - 2/r_i) \leq 2r_i$, so that

$$\frac{1}{4k}\kappa(H^*) \leq \frac{1}{4k} \sum_{i=1}^{n} r_i \leq \frac{1}{4k} \sum_{i=1}^{n} \frac{r_i}{1 - \frac{1}{2r_i}} \leq \frac{1}{2k} \sum_{i=1}^{n} r_i \leq \frac{1}{2k}\kappa(H^*)$$

From these bounds we see that the noise dependent term may be much larger than $n/(2k)$ when $\kappa(H^*) \gg n$, or when the spectrum of H^* covers a large range. For example, if $\lambda_i(H^*) = n - i + 1$ then it will be $\Omega(n^2/k)$.

Moreover, in the non-realizable case, $1/(2k) \operatorname{tr}\left(H^{*-1}\Sigma_g(\theta^*)\right)$ turns out to be the same asymptotic rate as

that achieved by the "empirical risk minimizer" (i.e. the estimator of θ that minimizes the expected loss over the training cases processed thus far) and is thus "optimal" in a certain sense. See Frostig et al. (2014) for a good recent discussion of this.

Thus we see that, under the restriction that a $\mathcal{O}(1/k^2)$ rate is achieved for the starting-point dependent term (which is noise *independent*), the use of 2nd-order optimization, in the case where $\alpha_k = 1/(k+1)$, allows us to obtain a better trade-off between the noise dependent and independent terms, which may be very significant in practice.

This result may seem counter-intuitive since 2nd-order optimization is usually thought of as speeding up deterministic optimization, and to be less important in the stochastic case. However, formal results about the effectiveness of 2nd-order optimization tend to rely on the use of a fixed learning rate, and so we can identify the use of an annealed learning rate schedule $\alpha_k = 1/(k+1)$ as the source of this apparent paradox. Indeed, Theorem 6 shows that in the case of a fixed learning rate $\alpha_k = \alpha$ the noise-independent terms will shrink exponentially quickly, although at the cost of preventing the noise-dependent term from ever shrinking beyond a certain fixed size (and therefore preventing convergence).

5.7.4 Related results

The related result most directly comparable to Theorem 6 is Theorem 1 of Bordes et al. (2009), which provides upper and lower bounds for $\mathrm{E}[h(\theta_k)] - h(\theta^*)$ in the case where $\alpha_k = 1/(k+k_0)$ for some k_0 and $\lambda_n\left(B^{-1}H^*\right) > 1/2$. In particular, using a different technique from our own, Bordes et al. (2009) show that[12]

$$\frac{1}{k}\frac{\mathrm{tr}(H^*B^{-1}\Sigma_g(\theta^*)B^{-1})}{4\left(\lambda_1(B^{-1}H^*)-\frac{1}{2}\right)} + o\left(\frac{1}{k}\right) \leq \mathrm{E}[h(\theta_k)] - h(\theta^*) \leq \frac{1}{k}\frac{\mathrm{tr}(H^*B^{-1}\Sigma_g(\theta^*)B^{-1})}{4\left(\lambda_n(B^{-1}H^*)-\frac{1}{2}\right)} + o\left(\frac{1}{k}\right)$$

Apart from the minor assumption that $k_0 = 1$ which is inherited from Theorem 1, as well as more significant additional hypothesis that $h(\theta)$ is quadratic (which could possibly be removed as per Remark 2), Theorem 6 represents a strict improvement to the above result since it gives the exact asymptotic value of $\mathrm{E}[h(\theta_k)] - h(\theta^*)$ instead of a bound (and is therefore more precise even in just the $\Omega(1/k)$ term), where we note that $\mathcal{O}(h(\theta_0)/k^{2\lambda_n(B^{-1}H^*)}) = o(h(\theta_0)/k)$[13]. Interestingly, the above result can be obtained using our proof technique by bounding $\mathrm{tr}(H^*W)$ using eqn. 5.25 as applied to the CALE given in eqn. 5.28 (instead of computing it exactly via Lemma 5).

5.7.5 Are these kinds of results useful in practice?

Because Theorem 6 only considers the quadratic case, it will be applicable only in the asymptotic limit as the objective tends to be locally well approximated by its own 2nd-order Taylor series approximation. It is worth asking whether formal results about asymptotic convergence speed like Theorem 6 are relevant in practice to real optimization methods as applied to real problems.

After all, most of the time spent during practical optimization, where we can't afford to wait until "fine convergence" takes place, or wouldn't even want to due to overfitting (Bottou and Bousquet, 2011), likely occurs in the "non-asymptotic"/exploration regime (Darken et al., 1992). In this regime, $\mathrm{E}[\theta_k]$ may still be very far from

[12] Note that the notation 'B' as it is used by Bordes et al. (2009) means the *inverse* of the matrix B as it appears in this chapter. And while Bordes et al. (2009) presents their bounds with $\bar{F}$ in place of Σ_g, these are the same matrix when evaluated at $\theta = \theta^*$ as we have $\mathrm{E}[g(\theta^*)] = 0$ (since θ^* is a local optimum).

[13] Note that Bordes et al. (2009) treats $h(\theta_0)$ as a constant in their asymptotic expressions, which is why the term $o(1/k)$ appears in their bound instead of $o(h(\theta_0)/k)$.

any local optimum θ^*, so that in particular the objective may not be locally well-approximated by the 2nd-order Taylor series approximation of h about such a θ^*, and the noise in the stochastic gradient estimate may not yet be a significant factor impeding convergence, despite how these results show it to be the main limiting factor asymptotically. For example, in such a situation, the term $\frac{1}{2}\operatorname{tr}\left(H^*(\mathrm{E}[\theta_k]-\theta^*)(\mathrm{E}[\theta_k]-\theta^*)^\top\right)$ in eqn. 5.13 which measures the contribution to $\mathrm{E}[h(\theta_k)]$ of the convergence (or lack thereof) of the mean parameter estimate $\mathrm{E}[\theta_k]$, may actually be much more significant, suggesting that we use a larger/fixed learning rate α_k, despite the potential negative impact this can have on the variance of θ_k (which then contributes to $\mathrm{E}[h(\theta_k)]$ via the term $\frac{1}{2}\operatorname{tr}(H^*V_k)$).

Because the problem of following a winding path through a high-dimensional non-convex objective might be roughly analogous to solving a "connected sequence" of local optimization problems (although likely without the requirement of achieving "fine convergence" for any such problem in the sequence, save perhaps for the last one), the "asymptotic regime" can perhaps be viewed as a rough analogy for the more practically significant non-asymptotic regime, and so results about asymptotic convergence speed may still contain useful information, especially if one pays attention to the noise-independent terms in the error bounds.

Noise in the gradient, as well as more classical optimization concerns about curvature etc., are both factors which must be overcome to some extent in order to make progress optimizing $\mathrm{E}[h(\theta_k)]$ in many practical problems (especially ones where overfitting isn't the main concern), and these kinds of asymptotic results provide one of the few *concrete* mathematical tools we currently have for understanding how different choices made in designing an optimization algorithm can affect either of them. Moreover, achieving the best *asymptotic* convergence speed through the choice of B_k and α_k is a problem that we can potentially tackle in a completely rigorous way, and studying it might very well lead to ideas that will prove useful for accelerating optimizing in the non-asymptotic regime.

But this being said, basing our decisions about the design of an optimization method entirely on such asymptotic convergence speed results can be dangerous. For example, the learning rate schedule $\alpha_k = 1/k$ is often a very poor one to use in practice, despite it yielding asymptotically "optimal" bounds.

Another issue with most results about asymptotic convergence speed is that they either assume that B_k is constant, or view its possible non-constant nature as a mathematical inconvenience to be overcome (i.e. by establishing or assuming that B_k^{-1} converges sufficiently quickly to some B^{*-1} so that $B_k^{-1} - B^{*-1}$ becomes asymptotically negligible) rather than as something possibly good or useful. Bordes et al. (2009) even advocates choosing B_k to be constant and equal to H^* (assuming this choice were practically feasible) on the basis of their own asymptotic convergence theorems. But taking a step back and considering the behavior of the optimizer in the non-asymptotic regime we can see that choosing B_k in this way will often be a bad idea, especially for problems where the local properties of h can vary wildly. Indeed, if we adopt the classical view advocated for in Section 5.5, where B_k is viewed as the curvature matrix used in a local quadratic model/approximation of h which needs to be accurate in a neighborhood of the *current* θ_k, then we see that B_k should in fact be defined based on the local properties of h around θ_k, and not based on some hypothetical θ^*, where the curvature properties of h may be significantly different.

Asymptotic convergence results also have very little to say about the crucial role that regularization/damping techniques play in practice (see Section 5.5), and fail to provide a practical prescription for the properties of a good damping technique, beyond the requirement that its effect must eventually become negligible in order to achieve the optimal asymptotic rate (which is the case for well-designed methods such as the Levenberg-Marquardt method (Moré, 1978)).

5.7.6 An analysis of averaging

In this subsection we will extend the analysis from Subsection 5.7.2 of the case where $\alpha_k = \alpha$ and $2\alpha\lambda_1(B^{-1}H^*) < 1$ to incorporate basic iterate averaging of the standard type (e.g. Polyak and Juditsky, 1992). In particular, we will bound $\mathrm{E}[h(\bar{\theta}_k)]$ where

$$\bar{\theta}_k = \frac{1}{k+1}\sum_{i=0}^{k}\theta_i$$

Note that while this type of averaging leads to elegant bounds (as we will see), a different type of averaging, which can often perform better in practice, uses the iteration

$$\bar{\theta}_k = (1-\beta_k)\theta_k + \beta_{k-1}\bar{\theta}_k \qquad\qquad \bar{\theta}_0 = \theta_0$$

for $\beta_k = \min\{1 - 1/k, \beta_{\max}\}$, for $0 < \beta_{\max} < 1$ close to 1 (e.g. $\beta_{\max} = 0.99$). This type of averaging has the advantage that it more quickly "forgets" the very early θ_i's (since their "weight" in the average decays exponentially quickly), at the cost of also forgetting potentially valuable information about previously observed training cases.

The main result of this subsection is stated as follows:

Theorem 8. *Suppose that θ_k is generated by the stochastic iteration in eqn. 5.12 with constant learning rate $\alpha_k = \alpha$ while optimizing a quadratic objective $h(\theta) = \frac{1}{2}(\theta-\theta^*)^\top H^*(\theta-\theta^*)$. Further suppose that $2\alpha\lambda_1(B^{-1}H^*) < 1$, and define $\bar{\theta}_k = \frac{1}{k+1}\sum_{i=0}^{k}\theta_i$.*

Then we have the following bound:

$$\mathrm{E}[h(\bar{\theta}_k)] - h(\theta^*) \leq \min\left\{\frac{1}{k+1}\operatorname{tr}\left(H^{*-1}\Sigma_g(\theta^*)\right),\ \frac{\alpha}{2}\operatorname{tr}\left(B^{-1}\Sigma_g(\theta^*)\right)\right\}$$
$$+ \min\left\{\frac{1}{(k+1)^2\alpha^2}\left\|H^{*-1/2}B(\theta_0-\theta^*)\right\|^2,\ \frac{1}{(k+1)\alpha}\left\|B^{1/2}(\theta_0-\theta^*)\right\|^2,\ 3h(\theta_0)\right\}$$

The proof of Theorem 8 is located in Appendix A.3.

5.7.7 Consequences of Theorem 8

In the case of stochastic 2nd-order optimization where we take $B = H^*$ (which allows us to use an α close to $1/2$) this gives

$$\mathrm{E}[h(\bar{\theta}_k)] - h(\theta^*) \leq \frac{\operatorname{tr}\left(H^{*-1}\Sigma_g(\theta^*)\right)}{k+1} + \frac{2h(\theta_0)}{(k+1)^2\alpha^2}$$

Choosing the maximum allowable value of α gives

$$\mathrm{E}[h(\bar{\theta}_k)] - h(\theta^*) \leq \frac{\operatorname{tr}\left(H^{*-1}\Sigma_g(\theta^*)\right)}{k+1} + \frac{8h(\theta_0)}{(k+1)^2}$$

which is a similar bound to that obtained for stochastic 2nd-order optimization using an annealed learning rate $\alpha_k = 1/(k+1)$ (see Section 5.7.3).

For the sake of comparison, applying Theorem 8 with $B = I$ gives that

$$\mathrm{E}[h(\bar{\theta}_k)] - h(\theta^*) \leq \frac{\mathrm{tr}\left(H^{*-1}\Sigma_g(\theta^*)\right)}{k+1} + \frac{\left\|H^{*-1/2}(\theta_0 - \theta^*)\right\|^2}{(k+1)^2\alpha^2} \tag{5.39}$$

under the assumption that $2\alpha\lambda_1(H^*) < 1$. For the maximum allowable value of α this becomes

$$\mathrm{E}[h(\bar{\theta}_k)] - h(\theta^*) \leq \frac{\mathrm{tr}\left(H^{*-1}\Sigma_g(\theta^*)\right)}{k+1} + \frac{\lambda_1(H^*)^2\left\|H^{*-1/2}(\theta_0 - \theta^*)\right\|^2}{(k+1)^2}$$

To compare this bound to the $B = H^*$ case we note that $\lambda_1(H^*)^2\|H^{*-1/2}(\theta_0 - \theta^*)\|^2 \geq 2h(\theta_0)$.

An interesting observation we can make about these bounds is that they *do not* demonstrate any improvement through the use of 2nd-order optimization to the noise-dependent term when we use averaging, which is the term that is asymptotically dominant in terms of k. Moreover, in the case where the stochastic gradients (the $g_k(\theta_k)$'s) are sampled using random training cases in the usual way so that $\Sigma_g(\theta) = \bar{F}(\theta)$, and the realizability hypothesis is satisfied so that $H^* = F(\theta^*) = \bar{F}(\theta^*)$ (see Section 5.7.1), we can see that simple stochastic gradient descent with averaging achieves a similar *asymptotic* convergence speed (given by $n/(k+1) + o(1/k)$) to that possessed by Fisher efficient methods like stochastic natural gradient descent (c.f. eqn. 5.11), despite not involving the use of curvature matrices.

However, these bounds *do* demonstrate an improvement to the noise-independent term (which instead depends on the starting point θ_0) through the use of 2nd-order optimization, since when H^* is ill-conditioned and $\theta_0 - \theta^*$ has a large component in the direction of eigenvectors of H^* with small eigenvalues, we will have

$$\lambda_1(H^*)^2\|H^{*-1/2}(\theta_0 - \theta^*)\|^2 \gg h(\theta_0)$$

Crucially, this noise-independent term may often matter more in practice, as $\lambda_1(H^*)^2\|H^{*-1/2}(\theta_0 - \theta^*)\|^2$ may be very large compared to $\Sigma_g(\theta^*)$, and we may be interested in stopping the optimization long before the more slowly shrinking noise-dependent term begins to dominate asymptotically (e.g. if we have a fixed iteration budget). This is especially likely to be the case if the gradient noise is mitigated through the use of large mini-batches.

It is also worth pointing out that compared to standard stochastic 2nd-order optimization with a fixed learning rate (as considered by the first part of Theorem 6), the noise-independent starting point-dependent term shrinks much more slowly when we use averaging (quadratically vs exponentially), or for that matter when we use an annealed learning rate $\alpha_k = 1/(k+1)$ (see Section 5.7.3). This seems to be the price one has to pay in order to ensure that the noise-dependent term shrinks as $1/k$. However, in practice one can potentially obtain a more favorable dependence on the starting point by adopting the "forgetful" version of averaging discussed at the beginning of this subsection.

5.7.8 Related results

Under weaker assumptions about the nature of the stochastic gradient noise (strictly weaker than our own), Polyak and Juditsky (1992) showed that

$$\mathrm{E}\left[(\theta_k - \theta^*)(\theta_k - \theta^*)^\top\right] = \frac{1}{k+1}H^{*-1/2}\Sigma_g(\theta^*)H^{*-1/2} + o\left(\frac{1}{k}\right)$$

which using the first line of eqn. 5.13 yields,

$$\mathrm{E}[h(\theta_k)] - h(\theta^*) = \frac{\mathrm{tr}\left(H^{*-1}\Sigma_g(\theta^*)\right)}{k+1} + o\left(\frac{1}{k}\right)$$

While consistent with Theorem 8, this bound gives a less detailed picture of convergence, and in particular it fails to quantify the relative contribution of the noise-dependent and independent terms and thus fails to properly distinguish between the behavior of stochastic 1st or 2nd-order optimization (i.e. $B = I$ vs $B = H^*$).

Assuming a model for the gradient noise which is consistent with linear least-squares regression and $B = I$, Défossez and Bach (2014) showed that

$$\mathrm{E}[h(\theta_k)] - h(\theta^*) \approx \frac{\mathrm{tr}\left(H^{*-1}\Sigma_g(\theta^*)\right)}{k+1} + \frac{\left\|H^{*-1/2}(\theta_0 - \theta^*)\right\|^2}{(k+1)^2\alpha^2}$$

which holds in the asymptotic limit as $\alpha \to 0$ and $k \to \infty$.

This expression is similar to the one generated by Theorem 8 (see eqn. 5.39), although it only holds in the asymptotic limit of small α and large k, and assumes a particular kind of noise which is more narrowly specialized than our general formulation. Notably however, our formulation does not capture this kind of noise precisely either, since for least-squares linear regression the covariance of the noise $\Sigma_g(\theta)$ will depend on θ, which is contrary to our assumption that it remains constant (with value $\Sigma_g(\theta^*)$). An interesting question for future research is whether Theorem 1 could be extended in way that would allow $\Sigma_g(\theta)$ to vary with θ, and whether this would allow us to prove a more general version of Theorem 8 that would cover the case of linear least-squares.

A result which is more directly comparable to Theorem 8 is Theorem 3 of Flammarion and Bach (2015), which when applied to the same general case considered in Theorem 8 gives the following upper bound (assuming that $B = I$[14] and $\alpha\lambda_1(H^*) \leq 1$):

$$\mathrm{E}[h(\bar{\theta}_k)] - h(\theta^*) \leq 4\alpha\,\mathrm{tr}(\Sigma_g(\theta^*)) + \frac{\|\theta_0 - \theta^*\|^2}{(k+1)\alpha}$$

Unlike the bound proved in Theorem 8, this bound fails to establish that $\mathrm{E}[h(\bar{\theta}_k)]$ even converges, since the term $4\alpha\,\mathrm{tr}(\Sigma_g(\theta^*))$ is constant in k.

5.8 A critical analysis of parameterization invariance

One of the main selling points of the natural gradient is its invariance to reparameterizations. In particular, the smooth path through the space of distributions, generated by the idealized natural gradient method with infinitesimally small steps, will be invariant to any smooth invertible reparameterization of the model.

More precisely, it can be said that this path will be the same whether we use the default parameterization (given by $P_{y|x}(\theta)$), or parameterize our model as $P_{y|x}(\zeta(\gamma))$, where $\zeta : \mathbb{R}^n \to \mathbb{R}^n$ is a smooth invertible "reparameterization function" which relates θ to γ as $\theta = \zeta(\gamma)$.

In the remainder of this section we will examine this "smooth path parameterization invariance" property more closely in order to answer the following questions:

[14]Note that the assumption that $B = I$ doesn't actually limit this result since stochastic 2nd-order optimization of a quadratic using a *fixed* B can be understood as stochastic gradient descent applied to a transformed version of the original quadratic (with an appropriately transformed gradient noise matrix Σ_g).

- How can we characterize it using only basic properties of the curvature matrix?
- Does it have a more elementary proof that can be applied to more general settings?
- What other kinds of curvature matrices give rise to it, and is the Hessian included among these?
- Will this invariance property imply that *practical* optimization algorithms based on the natural gradient (i.e. those that use large discrete steps) will behave in a way that is invariant to the parameterization?

Let ζ be as above, and let d_θ and d_γ be updates given in θ-space and γ-space (resp.). Additively updating γ by d_γ and translating it back to θ-space via ζ gives $\zeta(\gamma + d_\gamma)$. Measured by some non-specific norm $\|\cdot\|$, this differs from $\theta + d_\theta$ by:

$$\|\zeta(\gamma + d_\gamma) - (\theta + d_\theta)\|$$

This can be rewritten and bounded as

$$\|(\zeta(\gamma + d_\gamma) - (\zeta(\gamma) + J_\zeta d_\gamma)) + (J_\zeta d_\gamma - d_\theta)\| \leq \|\zeta(\gamma + d_\gamma) - (\zeta(\gamma) + J_\zeta d_\gamma)\| + \|J_\zeta d_\gamma - d_\theta\| \tag{5.40}$$

where J_ζ is the Jacobian of ζ and we have used $\theta = \zeta(\gamma)$.

The first term on the RHS of eqn. 5.40 measures the extent to which $\zeta(\gamma + d_\gamma)$ fails to be predicted by the first-order Taylor series approximation of ζ centered at γ (i.e. the locally optimal affine approximation of ζ at γ). This quantity will depend on the size of d_γ and the degree of smoothness of γ, and in case where ζ is affine, it will be exactly 0. We can further bound it by applying Taylor's theorem, which gives

$$\|\zeta(\gamma + d_\gamma) - (\zeta(\gamma) + J_\zeta d_\gamma)\| \leq \frac{1}{2} \left\| \begin{bmatrix} d_\gamma^\top H_{[\zeta]_1}(\gamma + c_1 d_\gamma) d_\gamma \\ d_\gamma^\top H_{[\zeta]_2}(\gamma + c_2 d_\gamma) d_\gamma \\ \vdots \\ d_\gamma^\top H_{[\zeta]_n}(\gamma + c_n d_\gamma) d_\gamma \end{bmatrix} \right\| \tag{5.41}$$

for some $c_i \in (0, 1)$. If we assume that there is some $C > 0$ so that for all γ and i, $\|H_{[\zeta]_i}(\gamma)\|_2 \leq C$, then using the fact that $|d_\gamma^\top H_{[\zeta]_i}(\gamma + c_n d_\gamma) d_\gamma| \leq \frac{1}{2}\|H_{[\zeta]_i}(\gamma + c_i d_\gamma)\|_2 \|d_\gamma\|_2^2$ we can further upper this by $\frac{1}{2} C \|d_\gamma\|_2^2 \|\mathbf{1}_n\|$.

The second term on the RHS of eqn. 5.40 will be zero when

$$J_\zeta d_\gamma = d_\theta \tag{5.42}$$

which, as we will see, is a condition that is satisfied in certain natural situations.

A slightly weakened version of this condition is that $J_\zeta d_\gamma \propto d_\theta$. Because we have

$$\lim_{\epsilon \to 0} \frac{\zeta(\gamma + \epsilon d_\gamma) - \zeta(\gamma)}{\epsilon} = J_\zeta d_\gamma$$

this condition can thus be interpreted as saying that that d_γ, when translated appropriately via ζ, points in the same direction away from θ that d_θ does. In the smooth path case, where the optimizer only moves a infinitesimally small distance in the direction of d_γ (or d_θ) at each iteration before recomputing it at the new γ (or θ), this condition is sufficient to establish that the path in γ space, when mapped back to θ space via the ζ function, will be the same as the path which would have been taken if the optimizer had worked directly in θ space.

However, for a practical update scheme where we move the entire distance of d_γ or d_θ before recomputing the update vector, such as the one in eqn. 5.6, this kind of invariance will not strictly hold even when eqn. 5.42 holds. But in such a case, the per-iteration error will bounded by the first term on the RHS of eqn. 5.40, and will thus be small provided that d_γ is sufficiently small and ζ is sufficiently smooth (as shown above).

Now, suppose we generate the updates d_θ and d_γ from curvature matrices B_θ and B_γ according to $d_\theta = -\alpha B_\theta^{-1}\nabla h$ and $d_\gamma = -\alpha B_\gamma^{-1}\nabla_\gamma h$, where $\nabla_\gamma h$ is the gradient of $h(\zeta(\gamma))$ w.r.t. γ. Then noting that $\nabla_\gamma h = J_\zeta^\top \nabla h$, the condition in eqn. 5.42 becomes equivalent to

$$J_\zeta B_\gamma^{-1} J_\zeta^\top \nabla h = B_\theta^{-1}\nabla h$$

For this to hold, a *sufficient* condition is that $B_\theta^{-1} = J_\zeta B_\gamma^{-1} J_\zeta^\top$. As J_ζ is invertible (since ζ is), an equivalent condition is

$$J_\zeta^\top B_\theta J_\zeta = B_\gamma \tag{5.43}$$

The following theorem summarizes our results so far.

Theorem 9. *Suppose that B_θ and B_γ are invertible matrices satisfying*

$$J_\zeta^\top B_\theta J_\zeta = B_\gamma$$

*Then we have that additively updating θ by $d_\theta = -\alpha B_\theta^{-1}\nabla h$ is **approximately** equivalent to additively updating γ by $d_\gamma = -\alpha B_\gamma^{-1}\nabla_\gamma h$, in the sense that $\zeta(\gamma + d_\gamma) \approx \theta + d_\theta$, with error bounded as*

$$\|\zeta(\gamma + d_\gamma) - (\theta + d_\theta)\| \leq \|\zeta(\gamma + d_\gamma) - (\zeta(\gamma) + J_\zeta d_\gamma)\|$$

Moreover, this error can be further bounded as in eqn. 5.41, and will be exactly 0 if ζ is affine.

Extending Theorem 9 in the obvious way from the case of a single update to one of an entire optimization path/trajectory gives the following corollary:

Corollary 10. *Suppose either that ζ is affine, or that α goes to zero (so that the optimizer follows an idealized smooth path). Further suppose that B_θ and B_γ are invertible matrices satisfying*

$$J_\zeta^\top B_\theta J_\zeta = B_\gamma$$

for all values of θ. Then the path followed by an iterative optimizer working in θ space and using additive updates of the form $d_\theta = -\alpha B_\theta^{-1}\nabla h$ is the same as the path followed by an iterative optimizer working in γ space using additive updates of the form $d_\gamma = -\alpha B_\theta^{-1}\nabla_\gamma h$, provided that the optimizers use equivalent starting points (i.e. $\theta_0 = \zeta(\gamma_0)$).

So from these results we see that practical natural gradient-based methods will *not* be invariant to smooth invertible reparameterizations ζ, although they will be *approximately* invariant, and in a way that depends on the smoothness of ζ and the size α of the learning rate.

5.8.1 When is the condition $J_\zeta^\top B_\theta J_\zeta = B_\gamma$ satisfied?

Suppose the curvature matrix B_θ has the form

$$B_\theta = \mathrm{E}_{D_{x,y}}[J_f^\top A J_f]$$

where $D_{x,y}$ is some arbitrary distribution over x and y (such as the training distribution), and $A \in \mathbb{R}^{m \times m}$ is some arbitrary invertible matrix (which can depend on x, y and θ). Note that this type of curvature matrix includes as special cases the GGN (whether or not it is equivalent to the Fisher), the Fisher, and the empirical Fisher (discussed in Section 5.6).

To obtain the analogous curvature matrix B_γ for the γ parameterization we replace f by $f \circ \zeta$ which gives

$$B_\gamma = \mathrm{E}_{D_{x,y}}[(J_f J_\zeta)^\top A (J_f J_\zeta)]$$

Then noting that $J_{f \circ \zeta} = J_f J_\zeta$, where J_ζ is the Jacobian of ζ, we have

$$B_\gamma = \mathrm{E}_{D_{x,y}}[(J_f J_\zeta)^\top A (J_f J_\zeta)] = J_\zeta^\top \, \mathrm{E}_{D_{x,y}}[J_f^\top A J_f] J_\zeta = J_\zeta^\top B_\theta J_\zeta$$

where we have used the fact that the reparameterization function ζ is independent of x and y.

Thus, this type of curvature matrix satisfies the sufficient condition in eqn. 5.43.

The Hessian on the other hand does not satisfy this sufficient condition, except in certain narrow special cases. To see this, note that taking the curvature matrix to be the Hessian gives

$$B_\gamma = J_\zeta^\top H J_\zeta + \frac{1}{|S|} \sum_{(x,y) \in S} \sum_{j=1}^{n} [\nabla h]_j H_{[\zeta]_j}$$

where $H = B_\theta$ is the Hessian of h w.r.t. θ. Thus, when the curvature matrix is the Hessian, the sufficient condition $J_\zeta^\top B_\theta J_\zeta = J_\zeta^\top H J_\zeta \propto B_\gamma$ holds if and only if

$$\frac{1}{|S|} \sum_{(x,y) \in S} \sum_{j=1}^{n} [\nabla h]_j H_{[\zeta]_j} = J_\zeta^\top H J_\zeta$$

where ∇L is the gradient of $L(y, z)$ w.r.t. z (evaluated at $z = f(x, \theta)$), and we allow a proportionality constant of 0. Rearranging this gives

$$\frac{1}{|S|} \sum_{(x,y) \in S} \sum_{j=1}^{n} [\nabla h]_j J_\zeta^{-\top} H_{[\zeta]_j} J_\zeta^{-1} = H$$

This relation is unlikely to be satisfied unless the left hand side is equal to 0. One situation where this will occur is when $H_{[\zeta]_j} = 0$ for each j, which holds when $[\zeta]_j$ is an affine function of γ. Another situation is where we have $\nabla h = 0$ for each $(x, y) \in S$.

5.9 A new interpretation of the natural gradient

As discussed in Section 5.5, the natural gradient is given by the minimizer of a local quadratic approximation $M(\delta)$ to h whose curvature matrix is the Fisher F. And if we have that the gradient ∇h and F are computed on

the same dataset S, $M(\delta)$ can be written as

$$M(\delta) = \frac{1}{2}\delta^\top F\delta + \nabla h^\top \delta + h(\theta) = \frac{1}{|S|}\sum_{(x,y)\in S}\left[\frac{1}{2}\delta^\top J_f^\top F_R J_f \delta + (J_f^\top \nabla_z \log r(y|z))^\top \delta\right] + h(\theta)$$

$$= \frac{1}{|S|}\sum_{(x,y)\in S}\left[\frac{1}{2}(J_f\delta)^\top F_R(J_f\delta) + \nabla_z \log r(y|z)^\top {F_R}^{-1} F_R(J_f\delta) + \frac{1}{2}(\nabla_z \log r(y|z))^\top F_R^{-1} F_R F_R^{-1} \nabla_z \log r(y|z)\right.$$

$$\left. - \frac{1}{2}(\nabla_z \log r(y|z))^\top F_R^{-1} F_R F_R^{-1} \nabla_z \log r(y|z)\right] + h(\theta)$$

$$= \frac{1}{|S|}\sum_{(x,y)\in S}\frac{1}{2}(J_f\delta + F_R^{-1}\nabla_z \log r(y|z))^\top F_R (J_f\delta + F_R^{-1}\nabla_z \log r(y|z)) + c$$

$$= \frac{1}{|S|}\sum_{(x,y)\in S}\frac{1}{2}\|J_f\delta + F_R^{-1}\nabla_z \log r(y|z)\|^2_{F_R} + c$$

where $c = h(\theta) - \frac{1}{2}(\sum_{(x,y)\in S}\nabla_z \log r(y|z)^\top F_R^{-1}\nabla_z \log r(y|z))/|S|$ is a constant independent of δ.

Note that for a given $(x, y) \in S$, $F_R^{-1}\nabla_z \log r(y|z)$ can be interpreted as the natural gradient direction in z-space for the objective corresponding to the KL divergence between the predictive distribution $R_{y|z}$ and a point distribution around the given y. In other words, it points in the direction which moves $R_{y|z}$ most quickly towards this point distribution as measured by the KL divergence (see Section 5.3). And assuming that the GGN interpretation of F holds, we know that it also corresponds to the optimal change in z according to the 2nd-order Taylor series approximation of the loss function $L(y, z)$.

Thus, $M(\delta)$ can be interpreted as the sum of squared distances (as measured using the Fisher metric tensor) between these "optimal" changes in the z's, and the changes in the z's which result from adding δ to θ, as predicted using 1st-order Taylor-series approximations to f.

In addition to giving us a new interpretation for the natural gradient, this expression also gives us an easy-to-compute bound on the largest possible improvement to h (as predicted by $M(\delta)$). In particular, since the squared error terms are non-negative, we have

$$M(\theta) - h(\theta) \geq -\frac{1}{2|S|}\sum_{(x,y)\in S}\nabla_z \log r(y|z)^\top F_R^{-1}\nabla_z \log r(y|z)$$

Given $F_R = H_L$, this quantity has the simple interpretation of being the optimal improvement in h (as predicted by a 2nd-order order model of $L(y, z)$ for each case in S) achieved in the hypothetical scenario where we can change the z's independently for each case. This bound can also be used to argue that computing F and ∇h on the same data will tend to make the natural gradient more stable and less prone to "exploding", as was done in Section 4.10.1 for updates based on the GGN.

Chapter 6

Kronecker-factored Approximate Curvature (K-FAC)

6.1 Introduction

The problem of training neural networks is one of the most important and highly investigated ones in machine learning. Despite work on layer-wise pretraining schemes, and various sophisticated optimization methods which try to approximate Newton-Raphson updates or natural gradient updates, stochastic gradient descent (SGD), possibly augmented with momentum, remains the method of choice for large-scale neural network training (Sutskever, Martens, Dahl, and Hinton, 2013).

From the work on Hessian-free optimization (HF) (Martens, 2010) and related methods (e.g. Vinyals and Povey, 2012) we know that updates computed using local curvature information can make much more progress per iteration than the scaled gradient. The reason that HF sees fewer practical applications than SGD are twofold. Firstly, its updates are much more expensive to compute, as they involve running linear conjugate gradient (CG) for potentially hundreds of iterations, each of which requires a matrix-vector product with the curvature matrix (which are as expensive to compute as the stochastic gradient on the current mini-batch). Secondly, HF's estimate of the curvature matrix must remain fixed while CG iterates, and thus the method is able to go through much less data than SGD can in a comparable amount of time, making it less well suited to stochastic optimizations.

As discussed in Section 4.7 and Sutskever, Martens, Dahl, and Hinton (2013), CG has the potential to be much faster at local optimization than gradient descent, when applied to quadratic objective functions. Thus, insofar as the objective can be locally approximated by a quadratic, each step of CG could potentially be doing a lot more work than each iteration of SGD, which would result in HF being much faster overall than SGD. However, there are examples of quadratic functions (e.g. Li, 2005), characterized by curvature matrices with highly spread-out eigenvalue distributions, where CG will have no distinct advantage over well-tuned gradient descent with momentum. Thus, insofar as the quadratic functions being optimized by CG within HF are of this character, HF shouldn't in principle be faster than well-tuned SGD with momentum. The extent to which neural network objective functions give rise to such quadratics is unclear, although Sutskever, Martens, Dahl, and Hinton (2013) provides some preliminary evidence that they do.

CG falls victim to this worst-case analysis because it is a first-order method. This motivates us to consider methods which don't rely on first-order methods like CG as their primary engines of optimization. One such class of methods which have been widely studied are those which work by directly inverting a diagonal, block-

diagonal, or low-rank approximation to the curvature matrix (e.g. Becker and LeCun, 1989; Schaul et al., 2013; Zeiler, 2013; Le Roux et al., 2008; Ollivier, 2013). In fact, a diagonal approximation of the Fisher information matrix is used within HF as a preconditioner for CG. However, these methods provide only a limited performance improvement in practice, especially compared to SGD with momentum (see for example Schraudolph et al., 2007; Zeiler, 2013), and many practitioners tend to forgo them in favor of SGD or SGD with momentum.

We know that the curvature associated with neural network objective functions is highly non-diagonal, and that updates which properly respect and account for this non-diagonal curvature, such as those generated by HF, can make much more progress minimizing the objective than the plain gradient or updates computed from diagonal approximations of the curvature (usually $\sim 10^2$ HF updates are required to adequately minimize most objectives, compared to the $\sim 10^4 - 10^5$ required by methods that use diagonal approximations). Thus, if we had an efficient and direct way to compute the inverse of a high-quality non-diagonal approximation to the curvature matrix (i.e. without relying on first-order methods like CG) this could potentially yield an optimization method whose updates would be large and powerful like HF's, while being (almost) as cheap to compute as the stochastic gradient.

In this work we develop such a method, which we call Kronecker-factored Approximate Curvature (K-FAC). We show that our method can be much faster in practice than even highly tuned implementations of SGD with momentum on certain standard neural network optimization benchmarks.

The main ingredient in K-FAC is a sophisticated approximation to the Fisher information matrix, which despite being neither diagonal nor low-rank, nor even block-diagonal with small blocks, can be inverted very efficiently, and can be estimated in an online fashion using arbitrarily large subsets of the training data (without increasing the cost of inversion).

This approximation is built in two stages. In the first, the rows and columns of the Fisher are divided into groups, each of which corresponds to *all the weights in a given layer*, and this gives rise to a block-partitioning of the matrix (where the blocks are *much* larger than those used by Le Roux et al. (2008) or Ollivier (2013)). These blocks are then approximated as Kronecker products between much smaller matrices, which we show is equivalent to making certain approximating assumptions regarding the statistics of the network's gradients.

In the second stage, this matrix is further approximated as having an *inverse* which is either block-diagonal or block-tridiagonal. We justify this approximation through a careful examination of the relationships between inverse covariances, tree-structured graphical models, and linear regression. Notably, this justification doesn't apply to the Fisher itself, and our experiments confirm that while the inverse Fisher does indeed possess this structure (approximately), the Fisher itself does not.

The rest of this chapter is organized as follows. **Section 6.2** establishes the basic operating assumptions of our method and defines some new notation (to complement the definitions given in Chapter 2). **Section 6.3** describes our initial Kronecker product approximation to the Fisher. **Section 6.4** describes our further block-diagonal and block-tridiagonal approximations of the inverse Fisher, and how these can be used to derive an efficient inversion algorithm. **Section 6.5** describes how we compute online estimates of the quantities required by our inverse Fisher approximation over a large "window" of previously processed mini-batches (which makes K-FAC very different from methods like HF or KSD, which base their estimates of the curvature on a single mini-batch). **Section 6.6** describes how we use our approximate Fisher to obtain a practical and robust optimization algorithm which requires very little manual tuning, through the careful application of various theoretically well-founded "damping" techniques that are standard in the optimization literature. Note that damping techniques compensate both for the local quadratic approximation being implicitly made to the objective, and for our further approximation of the Fisher, and are non-optional for essentially any 2nd-order method like K-FAC to work properly, as is well established by both theory and practice within the optimization literature (Nocedal and Wright, 2006). **Section**

6.7 describes a simple and effective way of adding a type of "momentum" to K-FAC, which we have found works very well in practice. **Section 6.8** describes the computational costs associated with K-FAC, and various ways to reduce them to the point where each update is at most only several times more expensive to compute than the stochastic gradient. **Section 6.9** gives complete high-level pseudocode for K-FAC. **Section 6.10** characterizes a broad class of network transformations and reparameterizations to which K-FAC is essentially invariant. **Section 6.11** considers some related prior methods for neural network optimization. Proofs of formal results are located in the appendix.

6.2 Additional notation and assumptions

6.2.1 Neural Networks

The notation used in this chapter for neural networks (e.g. a_i, s_i, etc.), loss functions, and training objectives will be consistent with that which was defined in Chapter 2. In addition, we will make a couple new definitions, and adopt some of the operating assumptions of Chapter 5, as we will be developing a method which approximates the natural gradient.

As was done in Section 5.1, we will assume that the loss is given by the negative log probability associated with a simple predictive distribution $R_{y|z}$ for y parameterized by z, i.e. that we have

$$L(y, z) = -\log r(y|z)$$

where r is $R_{y|z}$'s density function. And we will let $P_{y|x}(\theta) = R_{y|f(x,\theta)}$ denote the conditional distribution defined by the neural network, as parameterized by θ, and $p(y|x, \theta) = r(y|f(x, \theta))$ its density function. Note that minimizing the objective function $h(\theta)$ can be seen as maximum likelihood learning of the model $P_{y|x}(\theta)$.

Note that under this assumption we have

$$\mathcal{D}v = \frac{\mathrm{d}L(y, f(x, \theta))}{\mathrm{d}v} = -\frac{\mathrm{d}\log p(y|x, \theta)}{\mathrm{d}v}$$

Finally, because $\mathcal{D}s_i$ will be frequently used throughout this chapter we will define the following special notation for it:

$$g_i = \mathcal{D}s_i$$

6.3 A block-wise Kronecker-factored Fisher approximation

The main computational challenge associated with using the natural gradient is computing the inverse F^{-1} (or its product with ∇h) of the Fisher information matrix (as defined in Section 5.2). For large networks, with potentially millions of parameters, computing this inverse naively is computationally impractical. In this section we develop an initial approximation of F which will be a key ingredient in deriving our efficiently computable approximation to F^{-1} and the natural gradient.

Note that $\mathcal{D}\theta = [\mathrm{vec}(\mathcal{D}W_1)^\top \ \mathrm{vec}(\mathcal{D}W_2)^\top \ \cdots \ \mathrm{vec}(\mathcal{D}W_\ell)^\top]^\top$ and so F can be expressed as

$$\begin{aligned}
F &= \mathrm{E}\left[\mathcal{D}\theta \mathcal{D}\theta^\top\right] \\
&= \mathrm{E}\left[[\mathrm{vec}(\mathcal{D}W_1)^\top \ \mathrm{vec}(\mathcal{D}W_2)^\top \ \cdots \ \mathrm{vec}(\mathcal{D}W_\ell)^\top]^\top [\mathrm{vec}(\mathcal{D}W_1)^\top \ \mathrm{vec}(\mathcal{D}W_2)^\top \ \cdots \ \mathrm{vec}(\mathcal{D}W_\ell)^\top]\right] \\
&= \begin{bmatrix}
\mathrm{E}\left[\mathrm{vec}(\mathcal{D}W_1)\,\mathrm{vec}(\mathcal{D}W_1)^\top\right] & \mathrm{E}\left[\mathrm{vec}(\mathcal{D}W_1)\,\mathrm{vec}(\mathcal{D}W_2)^\top\right] & \cdots & \mathrm{E}\left[\mathrm{vec}(\mathcal{D}W_1)\,\mathrm{vec}(\mathcal{D}W_\ell)^\top\right] \\
\mathrm{E}\left[\mathrm{vec}(\mathcal{D}W_2)\,\mathrm{vec}(\mathcal{D}W_1)^\top\right] & \mathrm{E}\left[\mathrm{vec}(\mathcal{D}W_2)\,\mathrm{vec}(\mathcal{D}W_2)^\top\right] & \cdots & \mathrm{E}\left[\mathrm{vec}(\mathcal{D}W_2)\,\mathrm{vec}(\mathcal{D}W_\ell)^\top\right] \\
\vdots & \vdots & \ddots & \vdots \\
\mathrm{E}\left[\mathrm{vec}(\mathcal{D}W_\ell)\,\mathrm{vec}(\mathcal{D}W_1)^\top\right] & \mathrm{E}\left[\mathrm{vec}(\mathcal{D}W_\ell)\,\mathrm{vec}(\mathcal{D}W_2)^\top\right] & \cdots & \mathrm{E}\left[\mathrm{vec}(\mathcal{D}W_\ell)\,\mathrm{vec}(\mathcal{D}W_\ell)^\top\right]
\end{bmatrix}
\end{aligned}$$

Thus, we see that F can be viewed as an ℓ by ℓ block matrix, with the (i,j)-th block $F_{i,j}$ given by $F_{i,j} = \mathrm{E}\left[\mathrm{vec}(\mathcal{D}W_i)\,\mathrm{vec}(\mathcal{D}W_j)^\top\right]$.

Noting that $\mathcal{D}W_i = g_i \bar{a}_{i-1}^\top$ and that $\mathrm{vec}(uv^\top) = v \otimes u$ we have $\mathrm{vec}(\mathcal{D}W_i) = \mathrm{vec}(g_i \bar{a}_{i-1}^\top) = \bar{a}_{i-1} \otimes g_i$, and thus we can rewrite $F_{i,j}$ as

$$\begin{aligned}
F_{i,j} = \mathrm{E}\left[\mathrm{vec}(\mathcal{D}W_i)\,\mathrm{vec}(\mathcal{D}W_j)^\top\right] = \mathrm{E}\left[(\bar{a}_{i-1} \otimes g_i)(\bar{a}_{j-1} \otimes g_j)^\top\right] &= \mathrm{E}\left[(\bar{a}_{i-1} \otimes g_i)(\bar{a}_{j-1}^\top \otimes g_j^\top)\right] \\
&= \mathrm{E}\left[\bar{a}_{i-1}\bar{a}_{j-1}^\top \otimes g_i g_j^\top\right]
\end{aligned}$$

where $A \otimes B$ denotes the Kronecker product between $A \in \mathbb{R}^{m \times n}$ and B, and is given by

$$A \otimes B \equiv \begin{bmatrix}
[A]_{1,1}B & \cdots & [A]_{1,n}B \\
\vdots & \ddots & \vdots \\
[A]_{m,1}B & \cdots & [A]_{m,n}B
\end{bmatrix}$$

Note that the Kronecker product satisfies many convenient properties that we will make use of in this chapter, especially the identity $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$. See Van Loan (2000) for a good discussion of the Kronecker product.

Our initial approximation $\tilde{F}$ to F will be defined by the following block-wise approximation:

$$F_{i,j} = \mathrm{E}\left[\bar{a}_{i-1}\bar{a}_{j-1}^\top \otimes g_i g_j^\top\right] \approx \mathrm{E}\left[\bar{a}_{i-1}\bar{a}_{j-1}^\top\right] \otimes \mathrm{E}\left[g_i g_j^\top\right] = \bar{A}_{i-1,j-1} \otimes G_{i,j} = \tilde{F}_{i,j} \tag{6.1}$$

where $\bar{A}_{i,j} = \mathrm{E}\left[\bar{a}_i \bar{a}_j^\top\right]$ and $G_{i,j} = \mathrm{E}\left[g_i g_j^\top\right]$.

This gives

$$\tilde{F} = \begin{bmatrix}
\bar{A}_{0,0} \otimes G_{1,1} & \bar{A}_{0,1} \otimes G_{1,2} & \cdots & \bar{A}_{0,\ell-1} \otimes G_{1,\ell} \\
\bar{A}_{1,0} \otimes G_{2,1} & \bar{A}_{1,1} \otimes G_{2,2} & \cdots & \bar{A}_{1,\ell-1} \otimes G_{2,\ell} \\
\vdots & \vdots & \ddots & \vdots \\
\bar{A}_{\ell-1,0} \otimes G_{\ell,1} & \bar{A}_{\ell-1,1} \otimes G_{\ell,2} & \cdots & \bar{A}_{\ell-1,\ell-1} \otimes G_{\ell,\ell}
\end{bmatrix}$$

which has the form of what is known as a Khatri-Rao product in multivariate statistics.

The expectation of a Kronecker product is, in general, not equal to the Kronecker product of expectations, and so this is indeed a major approximation to make, and one which likely won't become exact under any realistic set of assumptions, or as a limiting case in some kind of asymptotic analysis. Nevertheless, it seems to be fairly accurate in practice, and is able to successfully capture the "coarse structure" of the Fisher, as demonstrated in Figure 6.1 for an example network.

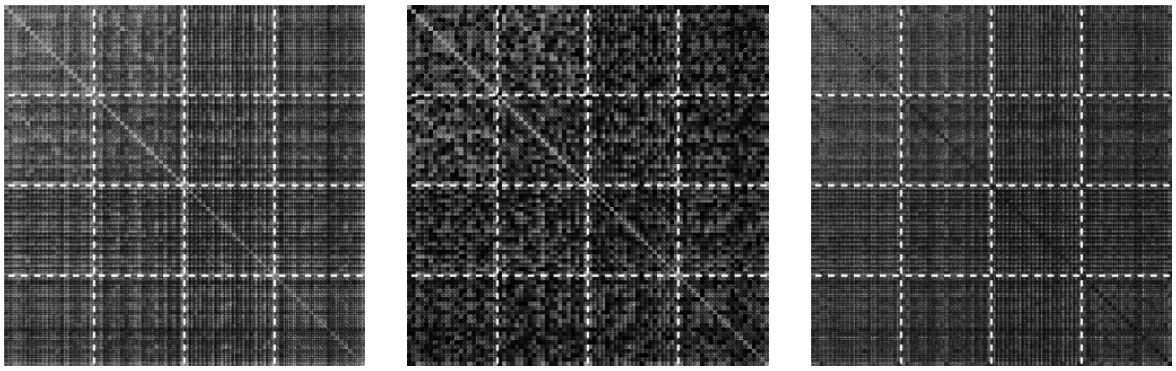

Figure 6.1: A comparison of the exact Fisher F and our block-wise Kronecker-factored approximation $\tilde{F}$, for the middle 4 layers of a standard deep neural network partially trained to classify a 16x16 down-scaled version of MNIST. The network was trained with 7 iterations of K-FAC in batch mode, achieving 5% error (the error reached 0% after 22 iterations) . The network architecture is 256-20-20-20-20-20-10 and uses standard tanh units. On the **left** is the exact Fisher F, in the **middle** is our approximation $\tilde{F}$, and on the **right** is the difference of these. The dashed lines delineate the blocks. Note that for the purposes of visibility we plot the absolute values of the entries, with the white level corresponding linearly to the size of these values (up to some maximum, which is the same in each image).

As we will see in later sections, this approximation leads to significant computational savings in terms of storage and inversion, which we will be able to leverage in order to design an efficient algorithm for computing an approximation to the natural gradient.

6.3.1 Interpretations of this approximation

Consider an arbitrary pair of weights $[W_i]_{k_1,k_2}$ and $[W_j]_{k_3,k_4}$ from the network, where $[\cdot]_{i,j}$ denotes the value of the (i,j)-th entry. We have that the corresponding derivatives of these weights are given by $\mathcal{D}[W_i]_{k_1,k_2} = \bar{a}^{(1)} g^{(1)}$ and $\mathcal{D}[W_j]_{k_3,k_4} = \bar{a}^{(2)} g^{(2)}$, where we denote for convenience $\bar{a}^{(1)} = [\bar{a}_{i-1}]_{k_1}$, $\bar{a}^{(2)} = [\bar{a}_{j-1}]_{k_3}$, $g^{(1)} = [g_i]_{k_2}$, and $g^{(2)} = [g_j]_{k_4}$.

The approximation given by eqn. 6.1 is equivalent to making the following approximation for each pair of weights:

$$\mathrm{E}\left[\mathcal{D}[W_i]_{k_1,k_2}\mathcal{D}[W_j]_{k_3,k_4}\right] = \mathrm{E}\left[(\bar{a}^{(1)}g^{(1)})(\bar{a}^{(2)}g^{(2)})\right] = \mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\, g^{(1)}g^{(2)}\right] \approx \mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\right]\mathrm{E}\left[g^{(1)}g^{(2)}\right] \tag{6.2}$$

And thus one way to interpret the approximation in eqn. 6.1 is that we are assuming statistical independence between products $\bar{a}^{(1)}\bar{a}^{(2)}$ of unit activities and products $g^{(1)}g^{(2)}$ of unit input derivatives.

Another more detailed interpretation of the approximation emerges by considering the following expression for the approximation error $\mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\, g^{(1)}g^{(2)}\right] - \mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\right]\mathrm{E}\left[g^{(1)}g^{(2)}\right]$ (which is derived in the appendix):

$$\kappa(\bar{a}^{(1)}, \bar{a}^{(2)}, g^{(1)}, g^{(2)}) + \mathrm{E}[\bar{a}^{(1)}]\kappa(\bar{a}^{(2)}, g^{(1)}, g^{(2)}) + \mathrm{E}[\bar{a}^{(2)}]\kappa(\bar{a}^{(1)}, g^{(1)}, g^{(2)}) \tag{6.3}$$

Here $\kappa(\cdot)$ denotes the cumulant of its arguments. Cumulants are a natural generalization of the concept of mean and variance to higher orders, and indeed 1st-order cumulants are means and 2nd-order cumulants are covariances. Intuitively, cumulants of order k measure the degree to which the interaction between variables is intrinsically of order k, as opposed to arising from many lower-order interactions.

A basic upper bound for the approximation error is

$$|\kappa(\bar{a}^{(1)}, \bar{a}^{(2)}, g^{(1)}, g^{(2)})| + |\,\mathrm{E}[\bar{a}^{(1)}]|\,|\kappa(\bar{a}^{(2)}, g^{(1)}, g^{(2)})| + |\,\mathrm{E}[\bar{a}^{(2)}]|\,|\kappa(\bar{a}^{(1)}, g^{(1)}, g^{(2)})| \tag{6.4}$$

which will be small if all of the higher-order cumulants are small (i.e. those of order 3 or higher). Note that in principle this upper bound may be loose due to possible cancellations between the terms in eqn. 6.3.

Because higher-order cumulants are zero for variables jointly distributed according to a multivariate Gaussian, it follows that this upper bound on the approximation error will be small insofar as the joint distribution over $\bar{a}^{(1)}$, $\bar{a}^{(2)}$, $g^{(1)}$, and $g^{(2)}$ is well approximated by a multivariate Gaussian. And while we are not aware of an argument for why this should be the case in practice, it does seem to be the case that for the example network from Figure 6.1, the size of the error is well predicted by the size of the higher-order cumulants. In particular, the total approximation error, summed over all pairs of weights in the middle 4 layers, is 2894.4, and is of roughly the same size as the corresponding upper bound (4134.6), whose size is tied to that of the higher order cumulants (due to the impossibility of cancellations in eqn. 6.4).

6.4 Additional approximations to $\tilde{F}$ and inverse computations

To the best of our knowledge there is no efficient general method for inverting a Khatri-Rao product like $\tilde{F}$. Thus, we must make further approximations if we hope to obtain an efficiently computable approximation of the inverse Fisher.

In the following subsections we argue that the inverse of $\tilde{F}$ can be reasonably approximated as having one of two special structures, either of which make it efficiently computable. The second of these will be slightly less restrictive than the first (and hence a better approximation) at the cost of some additional complexity. We will then show how matrix-vector products with these approximate inverses can be efficiently computed, which will thus give an efficient algorithm for computing an approximation to the natural gradient.

6.4.1 Structured inverses and the connection to linear regression

Suppose we are given a multivariate distribution whose associated covariance matrix is Σ.

Define the matrix B so that for $i \neq j$, $[B]_{i,j}$ is the coefficient on the j-th variable in the optimal linear predictor of the i-th variable from all the other variables, and for $i = j$, $[B]_{i,j} = 0$. Then define the matrix D to be the diagonal matrix where $[D]_{i,i}$ is the variance of the error associated with such a predictor of the i-th variable.

Pourahmadi (2011) showed that B and D can be obtained from the inverse covariance Σ^{-1} by the formulas

$$[B]_{i,j} = -\frac{[\Sigma^{-1}]_{i,j}}{[\Sigma^{-1}]_{i,i}} \quad \text{and} \quad [D]_{i,i} = \frac{1}{[\Sigma^{-1}]_{i,i}}$$

from which it follows that the inverse covariance matrix can be expressed as

$$\Sigma^{-1} = D^{-1}(I - B)$$

Intuitively, this result says that each row of the inverse covariance Σ^{-1} is given by the coefficients of the optimal linear predictor of the i-th variable from the others, up to a scaling factor. So if the j-th variable is much less "useful" than the other variables for predicting the i-th variable, we can expect that the (i, j)-th entry of the inverse covariance will be relatively small.

Note that "usefulness" is a subtle property as we have informally defined it. In particular, it is not equivalent to the degree of correlation between the j-th and i-th variables, or any such simple measure. As a simple example, consider the case where the j-th variable is equal to the k-th variable plus independent Gaussian noise. Since any linear predictor can achieve a lower variance simply by shifting weight from the j-th variable to the k-th variable, we have that the j-th variable is not useful (and its coefficient will thus be zero) in the task of predicting the i-th variable for any setting of i other than $i = j$ or $i = k$.

Noting that the Fisher F is a covariance matrix over $\mathcal{D}\theta$ w.r.t. the model's distribution (because $\mathrm{E}[\mathcal{D}\theta] = 0$ by Lemma 15), we can thus apply the above analysis to the distribution over $\mathcal{D}\theta$ to gain insight into the approximate structure of F^{-1}, and by extension its approximation $\tilde{F}^{-1}$.

Consider the derivative $\mathcal{D}W_i$ of the loss with respect to the weights W_i of layer i. Intuitively, if we are trying to predict one of the entries of $\mathcal{D}W_i$ from the other entries of $\mathcal{D}\theta$, those entries also in $\mathcal{D}W_i$ will likely be the most useful in this regard. Thus, it stands to reason that the largest entries of $\tilde{F}^{-1}$ will be those on the diagonal blocks, so that $\tilde{F}^{-1}$ will be well approximated as block-diagonal, with each block corresponding to a different $\mathcal{D}W_i$.

Beyond the other entries of $\mathcal{D}W_i$, it is the entries of $\mathcal{D}W_{i+1}$ and $\mathcal{D}W_{i-1}$ (i.e. those associated with adjacent layers) that will arguably be the most useful in predicting a given entry of $\mathcal{D}W_i$. This is because the true process for computing the loss gradient only uses information from the layer below (during the forward pass) and from the layer above (during the backwards pass). Thus, approximating $\tilde{F}^{-1}$ as block-tridiagonal seems like a reasonable and milder alternative than taking it to be block-diagonal. Indeed, this approximation would be exact if the distribution over $\mathcal{D}\theta$ were given by a directed graphical model which generated each of the $\mathcal{D}W_i$'s, one layer at a time, from either $\mathcal{D}W_{i+1}$ or $\mathcal{D}W_{i-1}$. Or equivalently, if $\mathcal{D}W_i$ were distributed according to an undirected Gaussian graphical model with binary potentials only between entries in the same or adjacent layers. Both of these models are depicted in Figure 6.3.

Now while in reality the $\mathcal{D}W_i$'s are generated using information from adjacent layers according to a process that is *neither linear nor Gaussian*, it nonetheless stands to reason that their joint statistics might be reasonably approximated by such a model. In fact, the idea of approximating the distribution over loss gradients with a directed graphical model forms the basis of the recent FANG method of Grosse and Salakhutdinov (2015).

Figure 6.2 examines the extent to which the inverse Fisher is well approximated as block-diagonal or block-tridiagonal for an example network.

In the following two subsections we show how both the block-diagonal and block-tridiagonal approximations to $\tilde{F}^{-1}$ give rise to computationally efficient methods for computing matrix-vector products with it. And at the end of Section 6.4 we present two figures (Figures 6.4 and 6.5) which examine the quality of these approximations for an example network.

6.4.2 Approximating $\tilde{F}^{-1}$ as block-diagonal

Approximating $\tilde{F}^{-1}$ as block-diagonal is equivalent to approximating $\tilde{F}$ as block-diagonal. A natural choice for such an approximation $\breve{F}$ of $\tilde{F}$, is to take the block-diagonal of $\breve{F}$ to be that of $\tilde{F}$. This gives the matrix

$$\breve{F} = \mathrm{diag}\left(\tilde{F}_{1,1}, \tilde{F}_{2,2}, \ \ldots, \ , \tilde{F}_{\ell,\ell}\right) = \mathrm{diag}\left(\bar{A}_{0,0} \otimes G_{1,1}, \ \bar{A}_{1,1} \otimes G_{2,2}, \ \ldots, \ \bar{A}_{\ell-1,\ell-1} \otimes G_{\ell,\ell}\right)$$

Using the identity $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$ we can easily compute the inverse of $\breve{F}$ as

$$\breve{F}^{-1} = \mathrm{diag}\left(\bar{A}_{0,0}^{-1} \otimes G_{1,1}^{-1}, \ \bar{A}_{1,1}^{-1} \otimes G_{2,2}^{-1}, \ \ldots, \ \bar{A}_{\ell-1,\ell-1}^{-1} \otimes G_{\ell,\ell}^{-1}\right)$$

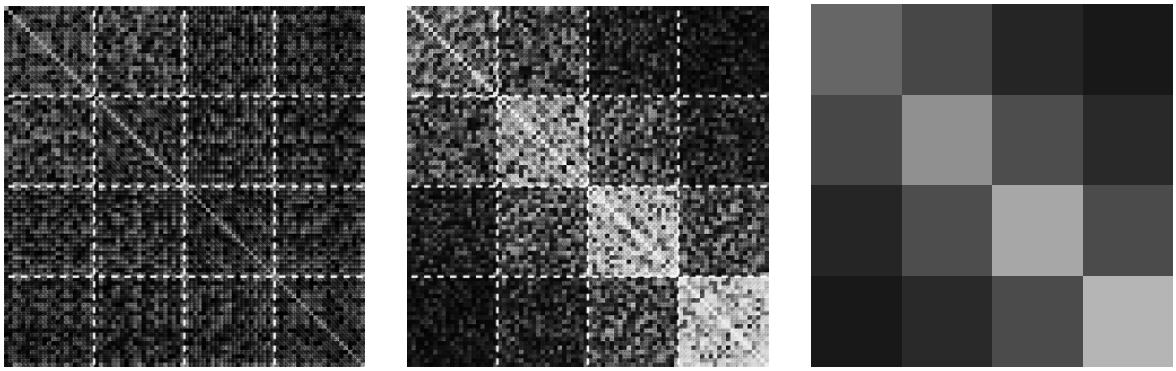

Figure 6.2: A comparison of our block-wise Kronecker-factored approximation $\tilde{F}$, and its inverse, using the example neural network from Figure 6.1. On the **left** is $\tilde{F}$, in the **middle** is its exact inverse, and on the **right** is a 4x4 matrix containing the averages of the absolute values of the entries in each block of the inverse. As predicted by our theory, the inverse exhibits an approximate block-tridiagonal structure, whereas $\tilde{F}$ itself does not. Note that the corresponding plots for the exact F and its inverse look similar. The very small blocks visible on the diagonal of the inverse each correspond to the weights on the outgoing connections of a particular unit. The inverse was computed subject to the factored Tikhonov damping technique described in Sections 6.6.3 and 6.6.6, using the same value of γ that was used by K-FAC at the iteration from which this example was taken (see Figure 6.1). Note that for the purposes of visibility we plot the absolute values of the entries, with the white level corresponding linearly to the size of these values (up to some maximum, which is chosen differently for the Fisher approximation and its inverse, due to the highly differing scales of these matrices).

Thus, computing $\breve{F}^{-1}$ amounts to computing the inverses of 2ℓ smaller matrices.

Then to compute $u = \breve{F}^{-1}v$, we can make use of the well-known identity $(A \otimes B)\operatorname{vec}(X) = \operatorname{vec}(BXA^\top)$ to get

$$U_i = G_{i,i}^{-1} V_i \bar{A}_{i-1,i-1}^{-1}$$

where v maps to $(V_1, V_2, \ldots, V_\ell)$ and u maps to $(U_1, U_2, \ldots, U_\ell)$ in an analogous way to how θ maps to $(W_1, W_2, \ldots, W_\ell)$.

Note that block-diagonal approximations to the Fisher information have been proposed before in TONGA (Le Roux et al., 2008), where each block corresponds to the weights associated with a particular unit. In our block-diagonal approximation, the blocks correspond to all the parameters in a given layer, and are thus *much* larger. In fact, they are so large that they would be impractical to invert as general matrices.

6.4.3 Approximating $\tilde{F}^{-1}$ as block-tridiagonal

Note that unlike in the above block-diagonal case, approximating $\tilde{F}^{-1}$ as block-tridiagonal is *not* equivalent to approximating $\tilde{F}$ as block-tridiagonal. Thus we require a more sophisticated approach to deal with such an approximation. We develop such an approach in this subsection.

To start, we will define $\hat{F}$ to be the matrix which agrees with $\tilde{F}$ on the tridiagonal blocks, and which satisfies the property that $\hat{F}^{-1}$ is block-tridiagonal. Note that this definition implies certain values for the off-tridiagonal blocks of $\hat{F}$ which will differ from those of $\tilde{F}$ insofar as $\tilde{F}^{-1}$ is not actually block-tridiagonal.

To establish that such a matrix $\hat{F}$ is well defined and can be inverted efficiently, we first observe that assuming that $\hat{F}^{-1}$ is block-tridiagonal is equivalent to assuming that it is the precision matrix of an undirected Gaussian graphical model (UGGM) over $\mathcal{D}\theta$ (as depicted in Figure 6.3), whose density function is proportional to $\exp(-\mathcal{D}\theta^\top \hat{F}^{-1} \mathcal{D}\theta)$. As this graphical model has a tree structure, there is an equivalent *directed* graphical

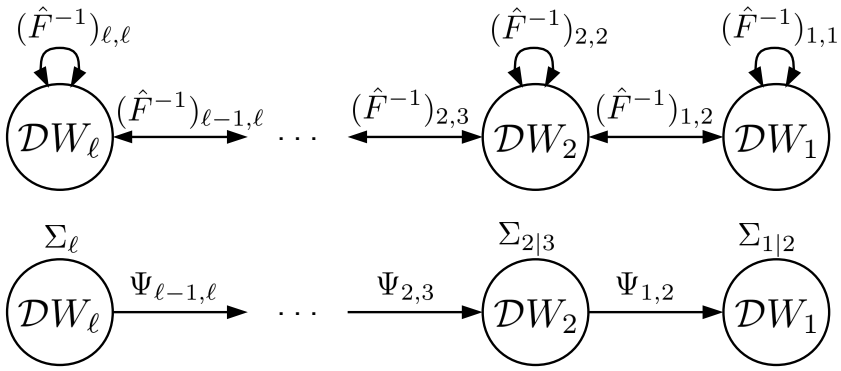

Figure 6.3: A diagram depicting the UGGM corresponding to $\hat{F}^{-1}$ and its equivalent DGGM. The UGGM's edges are labeled with the corresponding weights of the model (these are distinct from the network's weights). Here, $(\hat{F}^{-1})_{i,j}$ denotes the (i,j)-th block of $\hat{F}^{-1}$. The DGGM's edges are labeled with the matrices that specify the linear mapping from the source node to the conditional mean of the destination node (whose conditional covariance is given by its label).

model with the same distribution and the same (undirected) graphical structure (e.g. Bishop, 2006), where the directionality of the edges is given by a directed acyclic graph (DAG). Moreover, this equivalent directed model will also be linear/Gaussian, and hence a directed Gaussian Graphical model (DGGM).

Next we will show how the parameters of such a DGGM corresponding to $\hat{F}$ can be efficiently recovered from the tridiagonal blocks of $\hat{F}$, so that $\hat{F}$ is uniquely determined by these blocks (and hence well-defined). We will assume here that the direction of the edges is from the higher layers to the lower ones. Note that a different choice for these directions would yield a superficially different algorithm for computing the inverse of $\hat{F}$ that would nonetheless yield the same output.

For each i, we will denote the conditional covariance matrix of $\mathrm{vec}(\mathcal{D}W_i)$ on $\mathrm{vec}(\mathcal{D}W_{i+1})$ by $\Sigma_{i|i+1}$ and the linear coefficients from $\mathrm{vec}(\mathcal{D}W_{i+1})$ to $\mathrm{vec}(\mathcal{D}W_i)$ by the matrix $\Psi_{i,i+1}$, so that the conditional distributions defining the model are

$$\mathrm{vec}(\mathcal{D}W_i) \sim \mathcal{N}\left(\Psi_{i,i+1}\mathrm{vec}(\mathcal{D}W_{i+1}),\ \Sigma_{i|i+1}\right) \qquad \text{and} \qquad \mathrm{vec}(\mathcal{D}W_\ell) \sim \mathcal{N}\left(\vec{0},\ \Sigma_\ell\right)$$

Since Σ_ℓ is just the covariance of $\mathrm{vec}(\mathcal{D}W_\ell)$, it is given simply by $\hat{F}_{\ell,\ell} = \tilde{F}_{\ell,\ell}$. And for $i \leq \ell - 1$, we can see that $\Psi_{i,i+1}$ is given by

$$\Psi_{i,i+1} = \hat{F}_{i,i+1}\hat{F}^{-1}_{i+1,i+1} = \tilde{F}_{i,i+1}\tilde{F}^{-1}_{i+1,i+1} = \left(\bar{A}_{i-1,i} \otimes G_{i,i+1}\right)\left(\bar{A}_{i,i} \otimes G_{i+1,i+1}\right)^{-1} = \Psi^{\bar{A}}_{i-1,i} \otimes \Psi^{G}_{i,i+1}$$

where

$$\Psi^{\bar{A}}_{i-1,i} = \bar{A}_{i-1,i}\bar{A}^{-1}_{i,i} \qquad \text{and} \qquad \Psi^{G}_{i,i+1} = G_{i,i+1}G^{-1}_{i+1,i+1}$$

The conditional covariance $\Sigma_{i|i+1}$ is thus given by

$$\begin{aligned}\Sigma_{i|i+1} = \hat{F}_{i,i} - \Psi_{i,i+1}\hat{F}_{i+1,i+1}\Psi^\top_{i,i+1} &= \tilde{F}_{i,i} - \Psi_{i,i+1}\tilde{F}_{i+1,i+1}\Psi^\top_{i,i+1} \\ &= \bar{A}_{i-1,i-1} \otimes G_{i,i} - \Psi^{\bar{A}}_{i-1,i}\bar{A}_{i,i}\Psi^{\bar{A}\top}_{i-1,i} \otimes \Psi^{G}_{i,i+1}G_{i+1,i+1}\Psi^{G\top}_{i,i+1}\end{aligned}$$

Following the work of Grosse and Salakhutdinov (2015), we use the block generalization of well-known

"Cholesky" decomposition of the precision matrix of DGGMs (Pourahmadi, 1999), which gives

$$\hat{F}^{-1} = \Xi^\top \Lambda \Xi$$

where,

$$\Lambda = \operatorname{diag}\left(\Sigma_{1|2}^{-1}, \Sigma_{2|3}^{-1}, \ \ldots, \ \Sigma_{\ell-1|\ell}^{-1}, \Sigma_{\ell}^{-1}\right) \qquad \text{and} \qquad \Xi = \begin{bmatrix} I & -\Psi_{1,2} & & & \\ & I & -\Psi_{2,3} & & \\ & & I & \ddots & \\ & & & \ddots & -\Psi_{\ell-1,\ell} \\ & & & & I \end{bmatrix}$$

Thus, matrix-vector multiplication with $\hat{F}^{-1}$ amounts to performing matrix-vector multiplication by Ξ, followed by Λ, and then by $\Xi^\top$.

As in the block-diagonal case considered in the previous subsection, matrix-vector products with Ξ (and $\Xi^\top$) can be efficiently computed by using the well-known identity $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$. In particular, $u = \Xi^\top v$ can be computed as

$$U_i = V_i - \Psi_{i-1,i}^{G\top} V_{i-1} \Psi_{i-2,i-1}^{\bar{A}} \qquad \text{and} \qquad U_1 = V_1$$

and similarly $u = \Xi v$ can be computed as

$$U_i = V_i - \Psi_{i,i+1}^{G} V_{i+1} \Psi_{i-1,i}^{\bar{A}\top} \qquad \text{and} \qquad U_\ell = V_\ell$$

where the U_i's and V_i's are defined in terms of u and v as in the previous subsection.

Multiplying a vector v by Λ amounts to multiplying each $\operatorname{vec}(V_i)$ by the corresponding $\Sigma_{i|i+1}^{-1}$. This is slightly tricky because $\Sigma_{i|i+1}$ is the difference of Kronecker products, so we cannot use the straightforward identity $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$. Fortunately, there are efficient techniques for inverting such matrices which we discuss in detail in Appendix B.2.

6.4.4 Examining the approximation quality

Figures 6.4 and 6.5 examine the quality of the approximations $\breve{F}$ and $\hat{F}$ of $\tilde{F}$, which are derived by approximating $\tilde{F}^{-1}$ as block-diagonal and block-tridiagonal (resp.), for an example network.

From Figure 6.4, which compares $\breve{F}$ and $\hat{F}$ directly to $\tilde{F}$, we can see that while $\breve{F}$ and $\hat{F}$ exactly capture the diagonal and tridiagonal blocks (resp.) of $\tilde{F}$, as they must by definition, $\hat{F}$ ends up approximating the off-tridiagonal blocks of $\tilde{F}$ very well too. This is likely owed to the fact that the approximating assumption used to derive $\hat{F}$, that $\tilde{F}^{-1}$ is block-tridiagonal, is a very reasonable one in practice (judging by Figure 6.2).

Figure 6.5, which compares $\breve{F}^{-1}$ and $\hat{F}^{-1}$ to $\tilde{F}^{-1}$, paints an arguably more interesting and relevant picture, as the quality of the approximation of the natural gradient will be roughly proportional[1] to the quality of approximation of the *inverse* Fisher. We can see from this figure that due to the approximate block-diagonal structure of $\tilde{F}^{-1}$, $\breve{F}^{-1}$ is actually a reasonably good approximation of $\tilde{F}^{-1}$, despite $\breve{F}$ being a rather poor approximation of

[1] The error in any approximation $F_0^{-1}\nabla h$ of the natural gradient $F^{-1}\nabla h$ will be roughly proportional to the error in the approximation F_0^{-1} of the associated *inverse* Fisher F^{-1}, since $\|F^{-1}\nabla h - F_0^{-1}\nabla h\| \leq \|\nabla h\| \|F^{-1} - F_0^{-1}\|$.

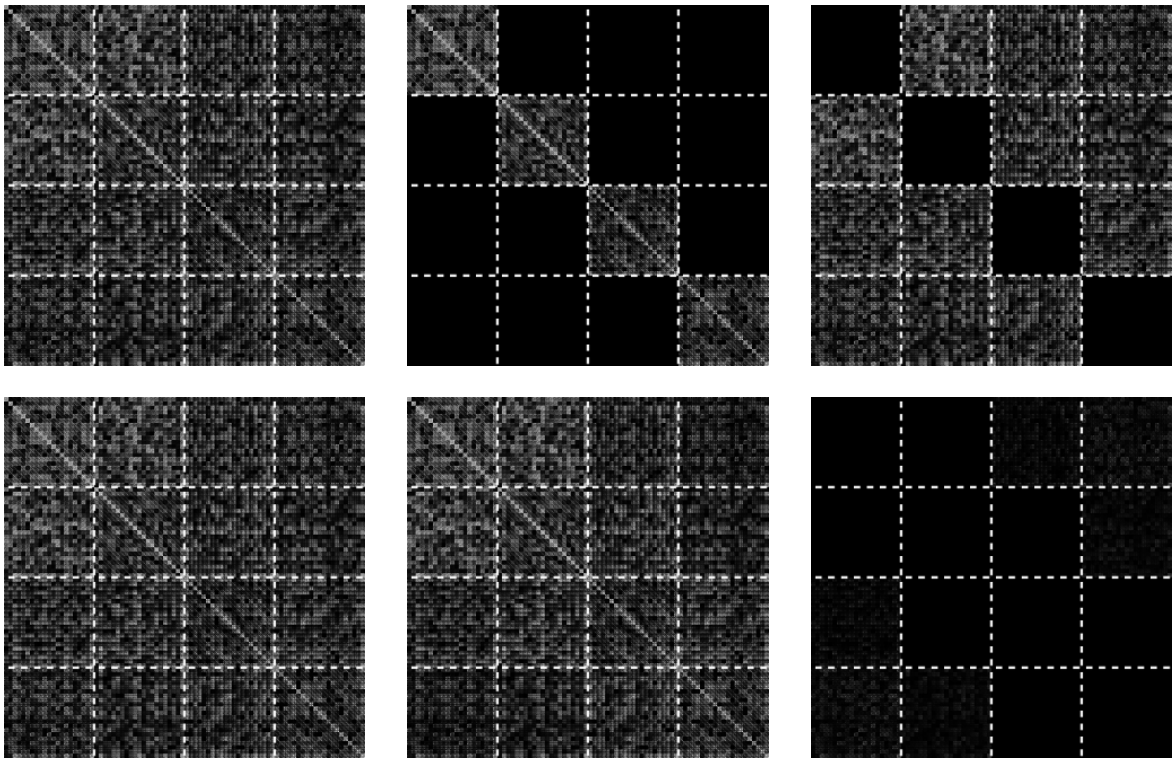

Figure 6.4: A comparison of our block-wise Kronecker-factored approximation $\tilde{F}$, and its approximations $\breve{F}$ and $\hat{F}$ (which are based on approximating the inverse $\tilde{F}^{-1}$ as either block-diagonal or block-tridiagonal, respectively), using the example neural network from Figure 6.1. On the **left** is $\tilde{F}$, in the **middle** its approximation, and on the **right** is the absolute difference of these. The **top row** compares to $\breve{F}$ and the **bottom row** compares to $\hat{F}$. While the diagonal blocks of the top right matrix, and the tridiagonal blocks of the bottom right matrix are exactly zero due to how $\breve{F}$ and $\hat{F}$ (resp.) are constructed, the off-tridiagonal blocks of the bottom right matrix, while being very close to zero, are actually non-zero (which is hard to see from the plot). Note that for the purposes of visibility we plot the absolute values of the entries, with the white level corresponding linearly to the size of these values (up to some maximum, which is the same in each image).

$\tilde{F}$ (based on Figure 6.4). Meanwhile, we can see that by accounting for the tri-diagonal blocks, $\hat{F}^{-1}$ is indeed a significantly better approximation of $\tilde{F}^{-1}$ than $\breve{F}^{-1}$ is, even on the *diagonal* blocks.

6.5 Estimating the required statistics

Recall that $\bar{A}_{i,j} = \mathrm{E}\left[\bar{a}_i \bar{a}_j^\top\right]$ and $G_{i,j} = \mathrm{E}\left[g_i g_j^\top\right]$. Both approximate Fisher inverses discussed in Section 6.4 require some subset of these. In particular, the block-diagonal approximation requires them for $i = j$, while the block-tridiagonal approximation requires them for $j \in \{i, i+1\}$ (noting that $\bar{A}_{i,j}^\top = \bar{A}_{j,i}$ and $G_{i,j}^\top = G_{j,i}$).

Since the $\bar{a}_i$'s don't depend on y, we can take the expectation $\mathrm{E}\left[\bar{a}_i \bar{a}_j^\top\right]$ with respect to just the training distribution $\hat{Q}_x$ over the inputs x. On the other hand, the g_i's do depend on y, and so the expectation[2] $\mathrm{E}\left[g_i g_j^\top\right]$ must be taken with respect to *both* $\hat{Q}_x$ and the network's predictive distribution $P_{y|x}$.

[2]It is important to note this expectation should *not* be taken with respect to the training/data distribution over y (i.e. $\hat{Q}_{y|x}$ or $Q_{y|x}$). Using the training/data distribution for y would perhaps give an approximation to a quantity known as the "empirical Fisher information matrix", which lacks the previously discussed equivalence to the Generalized Gauss-Newton matrix, and would not be compatible with the theoretical analysis performed in Section 6.3.1 (in particular, Lemma 15 would break down). Moreover, such a choice would not give rise to what is usually thought of as the natural gradient, and based on the findings of Martens (2010), would likely perform worse in practice as part of an optimization algorithm. See Section 5.6 for a more detailed discussion of the empirical Fisher and reasons why it may be a poor choice for a curvature matrix compared to the standard Fisher.

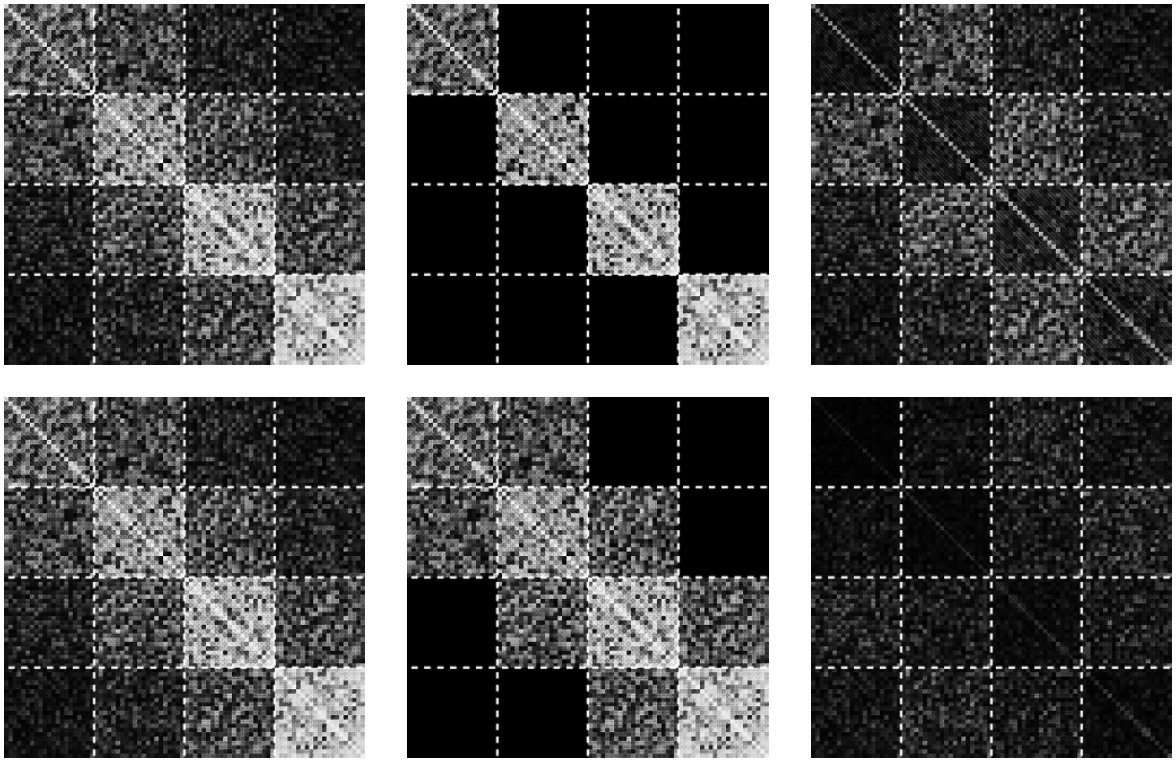

Figure 6.5: A comparison of the exact inverse $\tilde{F}^{-1}$ of our block-wise Kronecker-factored approximation $\tilde{F}$, and its block-diagonal and block-tridiagonal approximations $\breve{F}^{-1}$ and $\hat{F}^{-1}$ (resp.), using the example neural network from Figure 6.1. On the **left** is $\tilde{F}^{-1}$, in the **middle** its approximation, and on the **right** is the absolute difference of these. The **top row** compares to $\breve{F}^{-1}$ and the **bottom row** compares to $\hat{F}^{-1}$. The inverse was computed subject to the factored Tikhonov damping technique described in Sections 6.6.3 and 6.6.6, using the same value of γ that was used by K-FAC at the iteration from which this example was taken (see Figure 6.1). Note that for the purposes of visibility we plot the absolute values of the entries, with the white level corresponding linearly to the size of these values (up to some maximum, which is the same in each image).

While computing matrix-vector products with the $G_{i,j}$ could be done exactly and efficiently for a given input x (or small mini-batch of x's) by adapting the methods of Schraudolph (2002), there doesn't seem to be a sufficiently efficient method for computing the entire matrix itself. Indeed, the hardness results of Martens et al. (2012) suggest that this would require, for each example x in the mini-batch, work that is asymptotically equivalent to matrix-matrix multiplication involving matrices the same size as $G_{i,j}$. While a small constant number of such multiplications is arguably an acceptable cost (see Section 6.8), a number which grows with the size of the mini-batch would not be.

Instead, we will approximate the expectation over y by a standard Monte-Carlo estimate obtained by sampling y's from the network's predictive distribution and then rerunning the backwards phase of backpropagation (see Algorithm 1) as if these were the training targets.

Note that computing/estimating the required $\bar{A}_{i,j}$/$G_{i,j}$'s involves computing averages over outer products of various $\bar{a}_i$'s from network's usual forward pass, and g_i's from the modified backwards pass (with targets sampled as above). Thus we can compute/estimate these quantities on the same input data used to compute the gradient ∇h, at the cost of one or more additional backwards passes, and a few additional outer-product averages. Fortunately, this turns out to be quite inexpensive, as we have found that just one modified backwards pass is sufficient to obtain a good quality estimate in practice, and the required outer-product averages are similar to those already used to compute the gradient in the usual backpropagation algorithm.

In the case of online/stochastic optimization we have found that the best strategy is to maintain running estimates of the required $\bar{A}_{i,j}$'s and $G_{i,j}$'s using a simple exponentially decaying averaging scheme. In particular, we take the new running estimate to be the old one weighted by ϵ, plus the estimate on the new mini-batch weighted by $1 - \epsilon$, for some $0 \leq \epsilon < 1$. In our experiments we used $\epsilon = \min\{1 - 1/k, 0.95\}$, where k is the iteration number.

Note that the more naive averaging scheme where the estimates from each iteration are given equal weight would be inappropriate here. This is because the $\bar{A}_{i,j}$'s and $G_{i,j}$'s depend on the network's parameters θ, and these will slowly change over time as optimization proceeds, so that estimates computed many iterations ago will become stale.

This kind of exponentially decaying averaging scheme is commonly used in methods involving diagonal or block-diagonal approximations (with much smaller blocks than ours) to the curvature matrix (e.g. LeCun et al., 1998; Park et al., 2000; Schaul et al., 2013). Such schemes have the desirable property that they allow the curvature estimate to depend on much more data than can be reasonably processed in a single mini-batch.

Notably, for methods like HF which deal with the exact Fisher indirectly via matrix-vector products, such a scheme would be impossible to implement efficiently, as the exact Fisher matrix (or GGN) seemingly cannot be summarized using a compact data structure whose size is independent of the amount of data used to estimate it. Indeed, it seems that the only representation of the exact Fisher which would be independent of the amount of data used to estimate it would be an explicit $n \times n$ matrix (which is far too big to be practical). Because of this, HF and related methods must base their curvature estimates only on subsets of data that can be reasonably processed all at once, which limits their effectiveness in the stochastic optimization regime.

6.6 Update damping

6.6.1 Background and motivation

The idealized natural gradient approach is to follow the smooth path[3] in the Riemannian manifold (implied by the Fisher information matrix viewed as a metric tensor) that is generated by taking a series of infinitesimally small steps (in the original parameter space) in the direction of the natural gradient (which gets recomputed at each point). While this is clearly impractical as a real optimization method, one can take larger steps and still follow these paths approximately. But in our experience, to obtain an update which satisfies the minimal requirement of not worsening the objective function value, it is often the case that one must make the step size so small that the resulting optimization algorithm performs poorly in practice.

The reason that the natural gradient can only be reliably followed a short distance is that it is defined merely as an optimal *direction* (which trades off improvement in the objective versus change in the predictive distribution), and not a discrete *update*.

Fortunately, as observed in Section 5.4, the natural gradient can be understood using a more traditional optimization-theoretic perspective which implies how it can be used to generate updates that will be useful over larger distances. In particular, when $R_{y|z}$ is an exponential family model with z as its *natural* parameters (as it will be in our experiments), the Fisher becomes equivalent to the Generalized Gauss-Newton matrix (GGN), which is a positive semi-definite approximation of the Hessian of h. Additionally, there is the well-known fact that when $L(x, f(x, \theta))$ is the negative log-likelihood function associated with a given (x, y) pair (as we are assuming in this work), the Hessian H of h and the Fisher F are closely related in the sense H is the expected Hessian of L under the *training* distribution $\hat{Q}_{x,y}$, while F is the expected Hessian of L under the *model's* distribution $P_{x,y}$ (defined by the density $p(x, y) = p(y|x)q(x)$).

From these observations it follows that

$$M(\delta) = \frac{1}{2}\delta^\top F\delta + \nabla h(\theta)^\top \delta + h(\theta) \tag{6.5}$$

can be viewed as a convex approximation of the 2nd-order Taylor series of expansion of $h(\delta+\theta)$, whose minimizer δ^* is the (negative) natural gradient $-F^{-1}\nabla h(\theta)$. Note that if we add an ℓ_2 or "weight-decay" regularization term to h of the form $\frac{\eta}{2}\|\theta\|_2^2$, then similarly $F+\eta I$ can be viewed as an approximation of the Hessian of h, and replacing F with $F + \eta I$ in $M(\delta)$ yields an approximation of the 2nd-order Taylor series, whose minimizer is a kind of "regularized" (negative) natural gradient $-(F + \eta I)^{-1}\nabla h(\theta)$, which is what we end up using in practice.

From the interpretation of the natural gradient as the minimizer of $M(\delta)$, we can see that it fails to be useful as a local update only insofar as $M(\delta)$ fails to be a good local approximation to $h(\delta + \theta)$. And so as argued in Section 5.5, it is natural to make use of the various "damping" techniques that have been developed in the optimization literature for dealing with the breakdowns in local quadratic approximations that inevitably occur during optimization. Notably, this breakdown usually won't occur in the final "local convergence" stage of optimization where the function becomes well approximated as a convex quadratic within a sufficiently large neighborhood of the local optimum. This is the phase traditionally analyzed in most theoretical results, and while it is important that an optimizer be able to converge well in this final phase, it is arguably much more important from a practical standpoint that it behaves sensibly before this phase.

This initial "exploration phase" (Darken and Moody, 1990) is where damping techniques help in ways that

[3] Which has the interpretation of being a geodesic in the Riemannian manifold from the current predictive distribution towards the training distribution when using a likelihood or KL-divergence based objective function (see Martens (2014)).

are not apparent from the asymptotic convergence theorems alone, which is not to say there are not strong mathematical arguments that support their use (see Nocedal and Wright, 2006). In particular, in the exploration phase it will often still be true that $h(\theta + \delta)$ is accurately approximated by a convex quadratic *locally within some region* around $\delta = 0$, and that therefor optimization can be most efficiently performed by minimizing a sequence of such convex quadratic approximations within adaptively sized local regions.

Note that well designed damping techniques, such as the ones we will employ, automatically adapt to the local properties of the function, and effectively "turn themselves off" when the quadratic model becomes a sufficiently accurate local approximation of h, allowing the optimizer to achieve the desired asymptotic convergence behavior (Moré, 1978).

Successful and theoretically well-founded damping techniques include Tikhonov damping (aka Tikhonov regularization, which is closely connected to the trust-region method) with Levenberg-Marquardt style adaptation (Moré, 1978), line-searches, and trust regions, truncation, etc., all of which tend to be much more effective in practice than merely applying a learning rate to the update, or adding a *fixed* multiple of the identity to the curvature matrix. Indeed, a subset of these techniques was exploited in the work of Martens (2010), and primitive versions of them have appeared implicitly in older works such as Becker and LeCun (1989), and also in many recent diagonal methods like that of Zeiler (2013), although often without a good understanding of what they are doing and why they help.

Crucially, more powerful 2nd-order optimizers like HF and K-FAC, which have the capability of taking *much larger steps* than 1st-order methods (or methods which use diagonal curvature matrices), *require* more sophisticated damping solutions to work well, and will usually *completely fail* without them, which is consistent with predictions made in various theoretical analyses (e.g. Nocedal and Wright, 2006). As an analogy one can think of such powerful 2nd-order optimizers as extremely fast racing cars that need more sophisticated control systems than standard cars to prevent them from flying off the road. Arguably one of the reasons why high-powered 2nd-order optimization methods have historically tended to under-perform in machine learning applications, and in neural network training in particular, is that their designers did not understand or take seriously the issue of quadratic model approximation quality, and did not employ the more sophisticated and effective damping techniques that are available to deal with this issue.

For a detailed review and discussion of various damping techniques and their crucial role in practical 2nd-order optimization methods, see Section 4.6.

6.6.2 A highly effective damping scheme for K-FAC

Methods like HF which use the exact Fisher seem to work reasonably well with an adaptive Tikhonov regularization technique where λI is added to $F + \eta I$, and where λ is adapted according to Levenberg-Marquardt style adjustment rule. This common and well-studied method can be shown to be equivalent to imposing an adaptive spherical region (known as a "trust region") which constrains the optimization of the quadratic model (e.g Nocedal and Wright, 2006). However, we found that this simple technique is insufficient when used with our approximate natural gradient update proposals. In particular, we have found that there never seems to be a "good" choice for λ that gives rise to updates which are of a quality comparable to those produced by methods that use the exact Fisher, such as HF.

One possible explanation for this finding is that, unlike quadratic models based on the exact Fisher (or equivalently, the GGN), the one underlying K-FAC has no guarantee of being accurate up to 2nd-order. Thus, λ must remain large in order to compensate for this intrinsic 2nd-order inaccuracy of the model, which has the side effect

of "washing out" the small eigenvalues (which represent important low-curvature directions).

Fortunately, through trial and error, we were able to find a relatively simple and highly effective damping scheme, which combines several different techniques, and which works well within K-FAC. Our scheme works by computing an initial update proposal using a version of the above described adaptive Tikhonov damping/regularization method, and then re-scaling this according to quadratic model computed using the exact Fisher. This second step is made practical by the fact that it only requires a single matrix-vector product with the exact Fisher, and this can be computed efficiently using standard methods. We discuss the details of this scheme in the following subsections.

6.6.3 A factored Tikhonov regularization technique

In the first stage of our damping scheme we generate a candidate update proposal Δ by applying a slightly modified form of Tikhononv damping to our approximate Fisher, before multiplying $-\nabla h$ by its inverse.

In the usual Tikhonov regularization/damping technique, one adds $(\lambda + \eta)I$ to the curvature matrix (where η accounts for the ℓ_2 regularization), which is equivalent to adding a term of the form $\frac{\lambda+\eta}{2}\|\delta\|_2^2$ to the corresponding quadratic model (given by $M(\delta)$ with F replaced by our approximation). For the block-diagonal approximation $\breve{F}$ of $\tilde{F}$ (from Section 6.4.2) this amounts to adding $(\lambda + \eta)I$ (for a lower dimensional I) to each of the individual diagonal blocks, which gives modified diagonal blocks of the form

$$\bar{A}_{i-1,i-1} \otimes G_{i,i} \ + \ (\lambda + \eta)I \ = \ \bar{A}_{i-1,i-1} \otimes G_{i,i} \ + \ (\lambda + \eta)I \otimes I \tag{6.6}$$

Because this is the sum of two Kronecker products we cannot use the simple identity $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$ anymore. Fortunately however, there are efficient techniques for inverting such matrices, which we discuss in detail in Appendix B.2.

If we try to apply this same Tikhonov technique to our more sophisticated approximation $\hat{F}$ of $\tilde{F}$ (from Section 6.4.3) by adding $(\lambda + \eta)I$ to each of the diagonal blocks of $\hat{F}$, it is no longer clear how to efficiently invert $\hat{F}$. Instead, a solution which we have found works very well in practice (and which we also use for the block-diagonal approximation $\breve{F}$), is to add $\pi_i(\sqrt{\lambda + \eta})I$ and $\frac{1}{\pi_i}(\sqrt{\lambda + \eta})I$ for a scalar constant π_i to the individual Kronecker factors $\bar{A}_{i-1,i-1}$ and $G_{i,i}$ (resp.) of each diagonal block, giving

$$\left(\bar{A}_{i-1,i-1} + \pi_i(\sqrt{\lambda + \eta})I\right) \otimes \left(G_{i,i} + \frac{1}{\pi_i}(\sqrt{\lambda + \eta})I\right) \tag{6.7}$$

As this is a single Kronecker product, all of the computations described in Sections 6.4.2 and 6.4.3 can still be used here too, simply by replacing each $\bar{A}_{i-1,i-1}$ and $G_{i,i}$ with their modified versions $\bar{A}_{i-1,i-1} + \pi_i(\sqrt{\lambda + \eta})I$ and $G_{i,i} + \frac{1}{\pi_i}(\sqrt{\lambda + \eta})I$.

To see why the expression in eqn. 6.7 is a reasonable approximation to eqn. 6.6, note that expanding it gives

$$\bar{A}_{i-1,i-1} \otimes G_{i,i} + \pi_i(\sqrt{\lambda + \eta})I \otimes G_{i,i} + \frac{1}{\pi_i}(\sqrt{\lambda + \eta})\bar{A}_{i-1,i-1} \otimes I + (\lambda + \eta)I \otimes I$$

which differs from eqn. 6.6 by the residual error expression

$$\pi_i(\sqrt{\lambda + \eta})I \otimes G_{i,i} + \frac{1}{\pi_i}(\sqrt{\lambda + \eta})\bar{A}_{i-1,i-1} \otimes I$$

While the choice of $\pi_i = 1$ is simple and can sometimes work well in practice, a slightly more principled

choice can be found by minimizing the obvious upper bound (following from the triangle inequality) on the matrix norm of this residual expression, for some matrix norm $\|\cdot\|_v$. This gives

$$\pi_i = \sqrt{\frac{\|\bar{A}_{i-1,i-1} \otimes I\|_v}{\|I \otimes G_{i,i}\|_v}}$$

Evaluating this expression can be done efficiently for various common choices of the matrix norm $\|\cdot\|_v$. For example, for a general B we have $\|I \otimes B\|_F = \|B \otimes I\|_F = \sqrt{d}\|B\|_F$ where d is the height/dimension of I, and also $\|I \otimes B\|_2 = \|B \otimes I\|_2 = \|B\|_2$.

In our experience, one of the best and must robust choices for the norm $\|\cdot\|_v$ is the trace-norm, which for PSD matrices is given by the trace. With this choice, the formula for π_i has the following simple form:

$$\pi_i = \sqrt{\frac{\mathrm{tr}(\bar{A}_{i-1,i-1})/(d_{i-1}+1)}{\mathrm{tr}(G_{i,i})/d_i}}$$

where d_i is the dimension (number of units) in layer i. Intuitively, the inner fraction is just the average eigenvalue of $\bar{A}_{i-1,i-1}$ divided by the average eigenvalue of $G_{i,i}$.

Interestingly, we have found that this factored approximate Tikhonov approach, which was originally motivated by computational concerns, often works better than the exact version (eqn. 6.6) in practice. The reasons for this are still somewhat mysterious to us, but it may have to do with the fact that the inverse of the product of two quantities is often most robustly estimated as the inverse of the product of their individually regularized estimates.

6.6.4 Re-scaling according to the exact F

Given an update proposal Δ produced by multiplying the negative gradient $-\nabla h$ by our approximate Fisher inverse (subject to the Tikhonov technique described in the previous subsection), the second stage of our proposed damping scheme re-scales Δ according to the quadratic model M as computed with the exact F, to produce a final update $\delta = \alpha\Delta$.

More precisely, we optimize α according to the value of the quadratic model

$$M(\delta) = M(\alpha\Delta) = \frac{\alpha^2}{2}\Delta^\top(F + (\lambda+\eta)I)\Delta + \alpha\nabla h^\top\Delta + h(\theta)$$

as computed using an estimate of the exact Fisher F (to which we add the ℓ_2 regularization + Tikhonov term $(\lambda+\eta)I$). Because this is a 1-dimensional quadratic minimization problem, the formula for the optimal α can be computed very efficiently as

$$\alpha^* = \frac{-\nabla h^\top\Delta}{\Delta^\top(F + (\lambda+\eta)I)\Delta} = \frac{-\nabla h^\top\Delta}{\Delta^\top F\Delta + (\lambda+\eta)\|\Delta\|_2^2}$$

To evaluate this formula we use the current stochastic gradient ∇h (i.e. the same one used to produce Δ), and compute matrix-vector products with F using the input data from the same mini-batch. While using a mini-batch to compute F gets away from the idea of basing our estimate of the curvature on a long history of data (as we do with our *approximate* Fisher), it is made slightly less objectionable by the fact that we are only using it to estimate a single scalar quantity ($\Delta^\top F\Delta$). This is to be contrasted with methods like HF which perform a long and careful optimization of $M(\delta)$ using such an estimate of F.

Because the matrix-vector products with F are only used to compute scalar quantities in K-FAC, we can

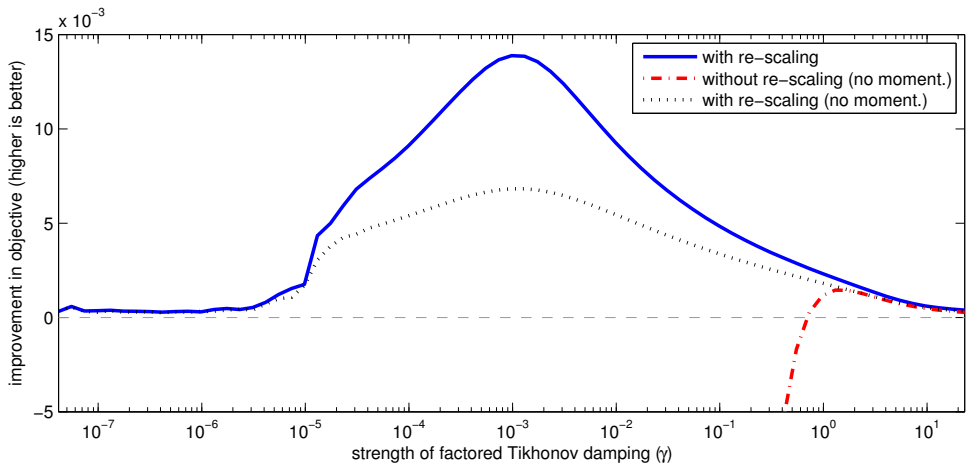

Figure 6.6: A comparison of the effectiveness of the proposed damping scheme, with and without the re-scaling techniques described in Section 6.6.4. The network used for this comparison is the one produced at iteration 500 by K-FAC (with the block-tridiagonal inverse approximation) on the MNIST autoencoder problem described in Section 6.13. The y-axis is the improvement in the objective function h (i.e. $h(\theta) - h(\theta + \delta)$) produced by the update δ, while the x-axis is the strength constant used in the factored Tikhonov damping technique (which is denoted by "γ" as described in Section 6.6.6). In the legend, "no moment." indicates that the momentum technique developed for K-FAC in Section 6.7 (which relies on the use of re-scaling) was not used.

reduce their computational cost by roughly one half (versus standard matrix-vector products with F) using a simple trick which is discussed in Appendix B.3.

Intuitively, this second stage of our damping scheme effectively compensates for the intrinsic inaccuracy of the approximate quadratic model (based on our approximate Fisher) used to generate the initial update proposal Δ, by essentially falling back on a more accurate quadratic model based on the exact Fisher.

Interestingly, by re-scaling Δ according to $M(\delta)$, K-FAC can be viewed as a version of HF which uses our approximate Fisher as a preconditioning matrix (instead of the traditional diagonal preconditioner), and runs CG for only 1 step, initializing it from 0. This observation suggests running CG for longer, thus obtaining an algorithm which is even closer to HF (although using a much better preconditioner for CG). Indeed, this approach works reasonably well in our experience, but suffers from some of the same problems that HF has in the stochastic setting, due its much stronger use of the mini-batch–estimated exact F.

Figure 6.6 demonstrates the effectiveness of this re-scaling technique versus the simpler method of just using the raw Δ as an update proposal. We can see that Δ, without being re-scaled, is a very poor update to θ, and won't even give *any* improvement in the objective function unless the strength of the factored Tikhonov damping terms is made very large. On the other hand, when the update is re-scaled, we can afford to compute Δ using a much smaller strength for the factored Tikhonov damping terms, and overall this yields a much larger and more effective update to θ.

6.6.5 Adapting λ

Tikhonov damping can be interpreted as implementing a trust-region constraint on the update δ, so that in particular the constraint $\|\delta\| \leq r$ is imposed for some r, where r depends on λ and the curvature matrix (e.g. Nocedal and Wright, 2006). While some approaches adjust r and then seek to find the matching λ, it is often simpler just to adjust λ directly, as the precise relationship between λ and r is complicated, and the curvature matrix is constantly evolving as optimization takes place.

The theoretically well-founded Levenberg-Marquardt style rule used by HF for doing this, which we will adopt for K-FAC, is given by

$$\text{if } \rho > 3/4 \text{ then } \lambda \leftarrow \omega_1 \lambda$$

$$\text{if } \rho < 1/4 \text{ then } \lambda \leftarrow \frac{1}{\omega_1} \lambda$$

where $\rho \equiv \dfrac{h(\theta + \delta) - h(\theta)}{M(\delta)}$ is the "reduction ratio" and $0 < \omega_1 < 1$ is some decay constant, and all quantities are computed on the current mini-batch (and M uses the exact F).

Intuitively, this rule tries to make λ as small as possible (and hence the implicit trust-region as large as possible) while maintaining the property that the quadratic model $M(\delta)$ remains a good *local* approximation to h (in the sense that it accurately predicts the value of $h(\theta + \delta)$ for the δ which gets chosen at each iteration). It has the desirable property that as the optimization enters the final convergence stage where M becomes an almost exact approximation in a sufficiently large neighborhood of the local minimum, the value of λ will go rapidly enough towards 0 that it doesn't interfere with the asymptotic local convergence theory enjoyed by 2nd-order methods (Moré, 1978).

In our experiments we applied this rule every T_1 iterations of K-FAC, with $\omega_1 = (19/20)^{T_1}$ and $T_1 = 5$, from a starting value of $\lambda = 150$. Note that the optimal value of ω_1 and the starting value of λ may be application dependent, and setting them inappropriately could significantly slow down K-FAC in practice.

Computing ρ can be done quite efficiently. Note that for the optimal δ, $M(\delta) = \frac{1}{2} \nabla h^\top \delta$, and $h(\theta)$ is available from the usual forward pass. The only remaining quantity which is needed to evaluate ρ is thus $h(\theta + \delta)$, which will require an additional forward pass. But fortunately, we only need to perform this once every T_1 iterations.

6.6.6 Maintaining a separate damping strength for the approximate Fisher

While the scheme described in the previous sections works reasonably well in most situations, we have found that in order to avoid certain failure cases and to be truly robust in a large variety of situations, the Tikhonov damping strength parameter for the factored Tikhonov technique described in Section 6.6.3 should be maintained and adjusted independently of λ. To this end we replace the expression $\sqrt{\lambda + \eta}$ in Section 6.6.3 with a separate constant γ, which we initialize to $\sqrt{\lambda + \eta}$ but which is then adjusted using a different rule, which is described at the end of this section.

The reasoning behind this modification is as follows. The role of λ, according to the Levenberg Marquardt theory (Moré, 1978), is to be as small as possible while maintaining the property that the quadratic model M remains a trust-worthy approximation of the true objective. Meanwhile, γ's role is to ensure that the initial update proposal Δ is as good an approximation as possible to the true optimum of M (as computed using a mini-batch estimate of the exact F), so that in particular the re-scaling performed in Section 6.6.4 is as benign as possible. While one might hope that adding the same multiple of the identity to our approximate Fisher as we do to the exact F (as it appears in M) would produce the best Δ in this regard, this isn't obviously the case. In particular, using a larger multiple may help compensate for the approximation we are making to the Fisher when computing Δ, and thus help produce a more "conservative" but ultimately more useful initial update proposal Δ, which is what we observe happens in practice.

A simple measure of the quality of our choice of γ is the (negative) value of the quadratic model $M(\delta) = M(\alpha\Delta)$ for the optimally chosen α. To adjust γ based on this measure (or others like it) we use a simple greedy adjustment rule. In particular, every T_2 iterations during the optimization we try 3 different values of γ (γ_0, $\omega_2\gamma_0$, and $(1/\omega_2)\gamma_0$, where γ_0 is the current value) and choose the new γ to be the best of these, as measured by our

quality metric. In our experiments we used $T_2 = 20$ (which must be a multiple of the constant T_3 as defined in Section 6.8), and $\omega_2 = (\sqrt{19/20})^{T_2}$.

We have found that $M(\delta)$ works well in practice as a measure of the quality of γ, and has the added bonus that it can be computed at essentially no additional cost from the incidental quantities already computed when solving for the optimal α. In our initial experiments we found that using it gave similar results to those obtained by using other obvious measures for the quality of γ, such as $h(\theta + \delta)$.

6.7 Momentum

Sutskever, Martens, Dahl, and Hinton (2013) found that momentum (Polyak, 1964; Plaut et al., 1986) was very helpful in the context of stochastic gradient descent optimization of deep neural networks. A version of momentum is also present in the original HF method, and it plays an arguably even more important role in more "stochastic" versions of HF (Martens and Sutskever, 2012; Kiros, 2013) (see Section 4.11).

A natural way of adding momentum to K-FAC, and one which we have found works well in practice, is to take the update to be $\delta = \alpha\Delta + \mu\delta_0$, where δ_0 is the final update computed at the previous iteration, and where α and μ are chosen to minimize $M(\delta)$. This allows K-FAC to effectively build up a better solution to the local quadratic optimization problem $\min_\delta M(\delta)$ (where M uses the *exact* F) over many iterations, somewhat similarly to how Matrix Momentum (Scarpetta et al., 1999) and HF do this (see Sutskever, Martens, Dahl, and Hinton, 2013).

The optimal solution for α and μ can be computed as

$$\begin{bmatrix} \alpha^* \\ \mu^* \end{bmatrix} = -\begin{bmatrix} \Delta^\top F\Delta + (\lambda+\eta)\|\Delta\|_2^2 & \Delta^\top F\delta_0 + (\lambda+\eta)\Delta^\top\delta_0 \\ \Delta^\top F\delta_0 + (\lambda+\eta)\Delta^\top\delta_0 & \delta_0^\top F\delta_0 + (\lambda+\eta)\|\delta_0\|_2^2 \end{bmatrix}^{-1} \begin{bmatrix} \nabla h^\top\Delta \\ \nabla h^\top\delta_0 \end{bmatrix}$$

The main cost in evaluating this formula is computing the two matrix-vector products $F\Delta$ and $F\delta_0$. Fortunately, the technique discussed in Appendix B.3 can be applied here to compute the 4 required scalars at the cost of only two forwards passes (equivalent to the cost of only one matrix-vector product with F).

Empirically we have found that this type of momentum provides substantial acceleration in regimes where the gradient signal has a low noise to signal ratio, which is usually the case in the early to mid stages of stochastic optimization, but can also be the case in later stages if the mini-batch size is made sufficiently large. These findings are consistent with predictions made by convex optimization theory, and with older empirical work done on neural network optimization (LeCun et al., 1998).

Notably, because the implicit "momentum decay constant" μ in our method is being computed on the fly, one doesn't have to worry about setting schedules for it, or adjusting it via heuristics, as one often does in the context of SGD.

Interestingly, if h is a quadratic function (so the definition of $M(\delta)$ remains fixed at each iteration) and all quantities are computed deterministically (i.e. without noise), then using this type of momentum makes K-FAC equivalent to performing preconditioned linear CG on $M(\delta)$, with the preconditioner given by our approximate Fisher. This follows from the fact that linear CG can be interpreted as a momentum method where the learning rate α and momentum decay coefficient μ are chosen to jointly minimize $M(\delta)$ at the current iteration.

6.8 Computational Costs and Efficiency Improvements

Let d be the typical number of units in each layer and m the mini-batch size. The significant computational tasks required to compute a single update/iteration of K-FAC, and rough estimates of their associated computational costs, are as follows:

1. standard forwards and backwards pass: $2C_1\ell d^2 m$
2. computation of the gradient ∇h on the current mini-batch using quantities computed in backwards pass: $C_2\ell d^2 m$
3. additional backwards pass with random targets (as described in Section 6.5): $C_1\ell d^2 m$
4. updating the estimates of the required $\bar{A}_{i,j}$'s and $G_{i,j}$'s from quantities computed in the forwards pass and the additional randomized backwards pass: $2C_2\ell d^2 m$
5. matrix inverses (or SVDs for the block-tridiagonal inverse, as described in Appendix B.2) required to compute the inverse of the approximate Fisher: $C_3\ell d^3$ for the block-diagonal inverse, $C_4\ell d^3$ for the block-tridiagonal inverse
6. various matrix-matrix products required to compute the matrix-vector product of the approximate inverse with the stochastic gradient: $C_5\ell d^3$ for the block-diagonal inverse, $C_6\ell d^3$ for the block-tridiagonal inverse
7. matrix-vector products with the exact F on the current mini-batch using the approach in Appendix B.3: $4C_1\ell d^2 m$ with momentum, $2C_1\ell d^2 m$ without momentum
8. additional forward pass required to evaluate the reduction ratio ρ needed to apply the λ adjustment rule described in Section 6.6.5: $C_1\ell d^2 m$ every T_1 iterations

Here the C_i are various constants that account for implementation details, and we are assuming the use of the naive cubic matrix-matrix multiplication and inversion algorithms when producing the cost estimates. Note that it it is hard to assign precise values to the constants, as they very much depend on how these various tasks are implemented.

Note that most of the computations required for these tasks will be sped up greatly by performing them in parallel across units, layers, training cases, or all of these. The above cost estimates however measure sequential operations, and thus may not accurately reflect the true computation times enjoyed by a parallel implementation. In our experiments we used a vectorized implementation that performed the computations in parallel over units and training cases, although not over layers (which is possible for computations that don't involve a sequential forwards or backwards "pass" over the layers).

Tasks 1 and 2 represent the standard stochastic gradient computation.

The costs of tasks 3 and 4 are similar and slightly smaller than those of tasks 1 and 2. One way to significantly reduce them is to use a random subset of the current mini-batch of size $\tau_1 m$ to update the estimates of the required $\bar{A}_{i,j}$'s and $G_{i,j}$'s. One can similarly reduce the cost of task 7 by computing the (factored) matrix-vector product with F using such a subset of size $\tau_2 m$, although we recommend proceeding with caution when doing this, as using inconsistent sets of data for the quadratic and linear terms in $M(\delta)$ can hypothetically cause instability problems which are avoided by using consistent data (see Section 4.10.1). In our experiments in Section 6.13 we used $\tau_1 = 1/8$ and $\tau_2 = 1/4$, which seemed to have a negligible effect on the quality of the resultant updates, while significantly reducing per-iteration computation time. In a separate set of unreported experiments we found

that in certain situations, such as when ℓ_2 regularization isn't used and the network starts heavily overfitting the data, or when smaller mini-batches were used, we had to revert to using $\tau_2 = 1$ to prevent significant deterioration in the quality of the updates.

The cost of task 8 can be made relatively insignificant by making the adjustment period T_1 for λ large enough. We used $T_1 = 5$ in our experiments.

The costs of tasks 5 and 6 are hard to compare directly with the costs associated with computing the gradient, as their relative sizes will depend on factors such as the architecture of the neural network being trained, as well as the particulars of the implementation. However, one quick observation we can make is that both tasks 5 and 6 involve computations that be performed in parallel across the different layers, which is to be contrasted with many of the other tasks which require *sequential* passes over the layers of the network.

Clearly, if $m \gg d$, then the cost of tasks 5 and 6 becomes negligible in comparison to the others. However, it is more often the case that m is comparable or perhaps smaller than d. Moreover, while algorithms for inverses and SVDs tend to have the same asymptotic cost as matrix-matrix multiplication, they are at least several times more expensive in practice, in addition to being harder to parallelize on modern GPU architectures (indeed, CPU implementations are often faster in our experience). Thus, C_3 and C_4 will typically be (much) larger than C_5 and C_6, and so in a basic/naive implementation of K-FAC, task 5 can dominate the overall per-iteration cost.

Fortunately, there are several possible ways to mitigate the cost of task 5. As mentioned above, one way is to perform the computations for each layer in parallel, and even simultaneously with the gradient computation and other tasks. In the case of our block-tridiagonal approximation to the inverse, one can avoid computing any SVDs or matrix square roots by using an iterative Stein-equation solver (see Appendix B.2). And there are also ways of reducing matrix-inversion (and even matrix square-root) to a short sequence of matrix-matrix multiplications using iterative methods (Pan and Schreiber, 1991). Furthermore, because the matrices in question only change slowly over time, one can consider hot-starting these iterative inversion methods from previous solutions. In the extreme case where d is very large, one can also consider using low-rank + diagonal approximations of the $\bar{A}_{i,j}$ and $G_{i,j}$ matrices maintained online (e.g. using a similar strategy as Le Roux et al. (2008)) from which inverses and/or SVDs can be more easily computed. Although based on our experience such approximations can, in some cases, lead to a substantial degradation in the quality of the updates.

While these ideas work reasonably well in practice, perhaps the simplest method, and the one we ended up settling on for our experiments, is to simply recompute the approximate Fisher inverse only every T_3 iterations (we used $T_3 = 20$ in our experiments). As it turns out, the curvature of the objective stays relatively stable during optimization, especially in the later stages, and so in our experience this strategy results in only a modest decrease in the quality of the updates.

If m is much smaller than d, the costs associated with task 6 can begin to dominate (provided T_3 is sufficiently large so that the cost of task 5 is relatively small). And unlike task 5, task 6 must be performed at every iteration. While the simplest solution is to increase m (while reaping the benefits of a less noisy gradient), in the case of the block-diagonal inverse it turns out that we can change the cost of task 6 from $C_5 \ell d^3$ to $C_5 \ell d^2 m$ by taking advantage of the low-rank structure of the stochastic gradient. The method for doing this is described below.

Let $\bar{\mathcal{A}}_i$ and $\mathcal{G}_i$ be matrices whose columns are the m $\bar{a}_i$'s and g_i's (resp.) associated with the current mini-batch. Let $\nabla_{W_i} h$ denote the gradient of h with respect to W_i, shaped as a matrix (instead of a vector). The estimate of $\nabla_{W_i} h$ over the mini-batch is given by $\frac{1}{m} \mathcal{G}_i \bar{\mathcal{A}}_{i-1}^\top$, which is of rank-$m$. From Section 6.4.2, computing the $\breve{F}^{-1} \nabla h$ amounts to computing $U_i = G_{i,i}^{-1} (\nabla_{W_i} h) \bar{A}_{i-1,i-1}^{-1}$. Substituting in our mini-batch estimate of $\nabla_{W_i} h$

gives

$$U_i = G_{i,i}^{-1} \left(\frac{1}{m} \mathcal{G}_i \bar{\mathcal{A}}_{i-1}^\top \right) \bar{A}_{i-1,i-1}^{-1} = \frac{1}{m} \left(G_{i,i}^{-1} \mathcal{G}_i \right) \left(\bar{\mathcal{A}}_{i-1}^\top \bar{A}_{i-1,i-1}^{-1} \right)$$

Direct evaluation of the expression on the right-hand side involves only matrix-matrix multiplications between matrices of size $m \times d$ and $d \times m$ (or between those of size $d \times d$ and $d \times m$), and thus we can reduce the cost of task 6 to $C_5 \ell d^2 m$.

Note that the use of standard ℓ_2 weight-decay is not compatible with this trick. This is because the contribution of the weight-decay term to each $\nabla_{W_i} h$ is νW_i, which will typically not be low-rank. Some possible ways around this issue include computing the weight-decay contribution $\nu \breve{F}^{-1} \theta$ separately and refreshing it only occasionally, or using a different regularization method, such as drop-out (Hinton et al., 2012) or weight-magnitude constraints.

Note that the adjustment technique for γ described in Section 6.6.6 requires that, at every T_2 iterations, we compute 3 different versions of the update for each of 3 candidate values of γ. In an ideal implementation these could be computed in parallel with each other, although in the summary analysis below we will assume they are computed serially.

Summarizing, we have that with all of the various efficiency improvements discussed in this section, the average per-iteration computational cost of K-FAC, in terms of *serial* arithmetic operations, is

$$\begin{aligned} &[(2 + \tau_1 + 2(1 + \chi_{mom})(1 + 2/T_2)\tau_2 + 1/T_1)C_1 + (1 + 2\tau_1)C_2]\ell d^2 m \\ &\quad + (1 + 2/T_2)[(C_4/T_3 + C_6)\chi_{tri} + C_3/T_3(1 - \chi_{tri})]\ell d^3 + (1 + 2/T_2)C_5(1 - \chi_{tri})\ell d^2 \min\{d, m\} \end{aligned}$$

where $\chi_{mom}, \chi_{tri} \in \{0, 1\}$ are flag variables indicating whether momentum and the block-tridiagonal inverse approximation (resp.) are used.

Plugging in the values of these various constants that we used in our experiments, for the block-diagonal inverse approximation ($\chi_{tri} = 0$) this becomes

$$(3.425C_1 + 1.25C_2)\ell d^2 m + 0.055 C_3 \ell d^3 + 1.1 C_5 \ell d^2 \min\{d, m\}$$

and for the block-tridiagonal inverse approximation ($\chi_{tri} = 1$)

$$(3.425C_1 + 1.25C_2)\ell d^2 m + (0.055C_4 + 1.1C_6)\ell d^3$$

which is to be compared to the per-iteration cost of SGD, as given by

$$(2C_1 + C_2)\ell d^2 m$$

6.9 Pseudocode for K-FAC

Algorithm 7 gives high-level pseudocode for the K-FAC method, with the details of how to perform the computations required for each major step left to their respective sections.

Algorithm 7 High-level pseudocode for K-FAC

- Initialize θ_1 (e.g. using a good method such as the ones described in Martens (2010) or Glorot and Bengio (2010))
- Choose initial values of λ (err on the side of making it too large)
- $\gamma \leftarrow \sqrt{\lambda + \eta}$
- $k \leftarrow 1$

while θ_k is not satisfactory **do**

- Choose a mini-batch size m (e.g. using a fixed value, an adaptive rule, or some predefined schedule)
- Select a random mini-batch $S' \subset S$ of training cases of size m
- Select a random subset $S_1 \subset S'$ of size $\tau_1|S'|$
- Select a random subset $S_2 \subset S'$ of size $\tau_2|S'|$
- Perform a forward and backward pass on S' to estimate the gradient $\nabla h(\theta_k)$ (see Algorithm 1)
- Perform an additional backwards pass on S_1 using random targets generated from the model's predictive distribution (as described in Section 6.5)
- Update the estimates of the required $\bar{A}_{i,j}$'s and $G_{i,j}$'s using the a_i's computed in forward pass for S_1, and the g_i's computed in the additional backwards pass for S_1 (as described Section 6.5)
- Choose a set Γ of new candidate γ's as described in Section 6.6.6 (setting $\Gamma = \{\gamma\}$ if not adjusting γ at this iteration, i.e. if $k \not\equiv 0 \pmod{T_2}$)

for each $\gamma \in \Gamma$ **do**

if recomputing the approximate Fisher inverse this iteration (i.e. if $k \equiv 0 \pmod{T_3}$ or $k \leq 3$) **then**

- Compute the approximate Fisher inverse (using the formulas derived in Section 6.4.2 or Section 6.4.3) from versions of the current $\bar{A}_{i,j}$'s and $G_{i,j}$'s which are modified as per the factored Tikhonov damping technique described in Section 6.6.3 (but using γ as described in Section 6.6.6)

end if

- Compute the update proposal Δ by multiplying current estimate of approximate Fisher inverse by the estimate of ∇h (using the formulas derived in Section 6.4.2 or Section 6.4.3). For layers with size $d < m$ consider using trick described at the end of Section 6.8 for increased efficiency.
- Compute the final update δ from Δ as described in Section 6.6.4 (or Section 6.7 if using momentum) where the matrix-vector products with F are estimated on S_2 using the a_i's computed in the forward pass

end for

- Select the δ and the new γ computing in the above loop that correspond to the lowest value of $M(\delta)$ (see Section 6.6.6)

if updating λ this iteration (i.e. if $k \equiv 0 \pmod{T_1}$) **then**

- Update λ with the Levenberg-Marquardt style rule described in Section 6.6.5

end if

- $\theta_{k+1} \leftarrow \theta_k + \delta$
- $k \leftarrow k + 1$

end while

6.10 Invariance Properties and the Relationship to Whitening and Centering

When computed with the exact Fisher, the natural gradient specifies a direction in the space of predictive distributions which is invariant to the specific way that the model is parameterized. This invariance means that the smooth path through distribution space produced by following the natural gradient with infinitesimally small steps will be similarly invariant.

For a practical natural gradient based optimization method which takes large discrete steps in the direction of the natural gradient, this invariance of the optimization path will only hold approximately. As shown by Martens (2014), the approximation error will go to zero as the effects of damping diminish and the reparameterizing function ζ tends to a locally linear function. Note that the latter will happen as ζ becomes smoother, or the local region containing the update shrinks to zero.

Because K-FAC uses an approximation of the natural gradient, these invariance results are not applicable in our case. Fortunately, as was shown by Martens (2014), one can establish invariance of an update direction with respect to a given reparameterization of the model by verifying certain simple properties of the curvature matrix C used to compute the update. We will use this result to show that, under the assumption that damping is absent (or negligible in its affect), K-FAC is invariant to a broad and natural class of transformations of the network.

This class of transformations is given by the following modified network definition (c.f. the definition in Section 2.1):

$$
\begin{aligned}
s_i^\dagger &= W_i^\dagger \bar{a}_{i-1}^\dagger \\
\bar{a}_i^\dagger &= \Omega_i \bar{\phi}_i(\Phi_i s_i^\dagger)
\end{aligned}
$$

where $\bar{\phi}_i$ is the function that computes ϕ_i and then appends a homogeneous coordinate (with value 1), Ω_i and Φ_i are arbitrary invertible matrices of the appropriate sizes (except that we assume $\Omega_\ell = I$), $\bar{a}_0^\dagger = \Omega_0 \bar{a}_0$, and where the network's output is given by $f^\dagger(x, \theta) = a_\ell^\dagger$. Note that because Ω_i multiplies $\bar{\phi}_i(\Phi_i s_i^\dagger)$, it can implement arbitrary translations of the unit activities $\phi_i(\Phi_i s_i^\dagger)$ in addition to arbitrary linear transformations. Figure 6.7 illustrates our modified network definition for $\ell = 2$ (c.f. Figure 2.1).

Here, and going forward, we will add a "$\dagger$" superscript to any network-dependent quantity in order to denote the analogous version of it computed by the transformed network. Note that under this identification, the loss derivative formulas for the transformed network are analogous to those of the original network, and so our various Fisher approximations are still well defined.

The following theorem describes the main technical result of this section.

Theorem 11. *There exists an invertible linear function $\theta = \zeta(\theta^\dagger)$ so that $f^\dagger(x, \theta^\dagger) = f(x, \theta) = f(x, \zeta(\theta^\dagger))$, and thus the transformed network can be viewed as a reparameterization of the original network by $\theta^\dagger$. Moreover, additively updating θ by $\delta = -\alpha \breve{F}^{-1} \nabla h$ or $\delta = -\alpha \hat{F}^{-1} \nabla h$ in the original network is equivalent to additively updating $\theta^\dagger$ by $\delta^\dagger = -\alpha \breve{F}^{\dagger -1} \nabla h^\dagger$ or $\delta^\dagger = -\alpha \hat{F}^{\dagger -1} \nabla h^\dagger$ (resp.) in the transformed network, in the sense that $\zeta(\theta^\dagger + \delta^\dagger) = \theta + \delta$.*

This immediately implies the following corollary which characterizes the invariance of a basic version of K-FAC to the given class of network transformations.

Corollary 12. *The optimization path taken by K-FAC (using either of our Fisher approximations $\breve{F}$ or $\hat{F}$) through the space of predictive distributions is the same for the default network as it is for the transformed network (where the Ω_i's and Φ_i's remain fixed). This assumes the use of an equivalent initialization ($\theta_0 = \zeta(\theta_0^\dagger)$), and a basic*

version of K-FAC where damping is absent or negligible in effect, momentum is not used, and where the learning rates are chosen in a way that is independent of the network's parameterization.

While this corollary assumes that the Ω_i's and Φ_i's are fixed, if we relax this assumption so that they are allowed to vary smoothly with θ, then ζ will be a smooth function of θ, and so as discussed in Martens (2014), invariance of the optimization path will hold approximately in a way that depends on the smoothness of ζ (which measures how quickly the Ω_i's and Φ_i's change) and the size of the update. Moreover, invariance will hold exactly in the limit as the learning rate goes to 0.

Note that the network transformations can be interpreted as replacing the network's nonlinearity $\bar{\phi}_i(s_i)$ at each layer i with a "transformed" version $\Omega_i \bar{\phi}_i(\Phi_i s_i)$. So since the well-known logistic sigmoid and tanh functions are related to each other by such a transformation, an immediate consequence of Corollary 12 is that K-FAC is invariant to the choice of logistic sigmoid vs. tanh activation functions (provided that equivalent initializations are used and that the effect of damping is negligible, etc.).

Also note that because the network inputs are also transformed by Ω_0, K-FAC is thus invariant to arbitrary affine transformations of the input, which includes many popular training data preprocessing techniques.

Many other natural network transformations, such as ones which "center" and normalize unit activities so that they have mean 0 and variance 1 can be described using diagonal choices for the Ω_i's and Φ_i's which vary smoothly with θ. In addition to being approximately invariant to such transformations (or exactly, in the limit as the step size goes to 0), K-FAC is similarly invariant to a more general class of such transformations, such as those which transform the units so that they have a mean of 0, so they are "centered", and a *covariance matrix* of I, so they are "whitened", which is a much stronger condition than the variances of the individual units each being 1.

In the case where we use the block-diagonal approximation $\breve{F}$ and compute updates without damping, Theorem 11 affords us an additional elegant interpretation of what K-FAC is doing. In particular, the updates produced by K-FAC end up being equivalent to those produced by *standard gradient descent* using a network which is transformed so that the unit activities and the unit-gradients are both centered and whitened (with respect to the model's distribution). This is stated formally in the following corollary.

Corollary 13. *Additively updating θ by $-\alpha \breve{F}^{-1} \nabla h$ in the original network is equivalent to additively updating $\theta^\dagger$ by the gradient descent update $-\alpha \nabla h^\dagger$ (where $\theta = \zeta(\theta^\dagger)$ as in Theorem 11) in a transformed version of the network where the unit activities $a_i^\dagger$ and the unit-gradients $g_i^\dagger$ are both centered and whitened with respect to the model's distribution.*

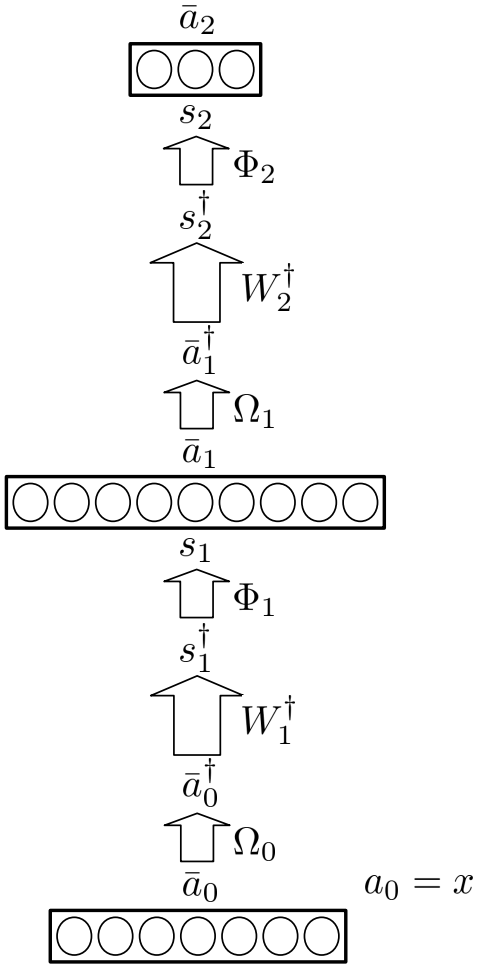

Figure 6.7: A depiction of a transformed network for $\ell = 2$. Note that the quantities labeled with $\bar{a}_i$ and s_i (without "†") will be equal to the analogous quantities from the default network, provided that $\theta = \zeta(\theta^\dagger)$ as in Theorem 11.

6.11 Related Work

The Hessian-free optimization method of Martens (2010) uses linear conjugate gradient (CG) to optimize local quadratic models of the form of eqn. 6.5 (subject to an adaptive Tikhonov damping technique) in lieu of directly solving it using matrix inverses. As discussed in the introduction, the main advantages of K-FAC over HF are twofold. Firstly, K-FAC uses an efficiently computable direct solution for the inverse of the curvature matrix and thus avoids the costly matrix-vector products associated with running CG within HF. Secondly, it can estimate the curvature matrix from a lot of data by using an online exponentially-decayed average, as opposed to relatively small-sized fixed mini-batches used by HF. The cost of doing this is of course the use of an inexact approximation to the curvature matrix.

Le Roux et al. (2008) proposed a neural network optimization method known as TONGA based on a block-diagonal approximation of the *empirical* Fisher where each block corresponds to the weights associated with a particular unit. By contrast, K-FAC uses *much* larger blocks, each of which corresponds to all the weights within a particular layer. The matrices which are inverted in K-FAC are roughly the same size as those which are inverted in TONGA, but rather than there being one per unit as in TONGA, there are only two per layer. Therefore, K-FAC is significantly less computationally intensive than TONGA, despite using what is arguably a much more accurate approximation to the Fisher. Note that to help mitigate the cost of the many matrix inversions it requires, TONGA approximates the blocks as being low-rank plus a diagonal term, although this introduces further approximation error.

Centering methods work by either modifying the gradient (Schraudolph, 1998) or dynamically reparameterizing the network itself (Raiko et al., 2012; Vatanen et al., 2013; Wiesler et al., 2014), so that various unit-wise scalar quantities like the activities (the a_i's) and local derivatives (the $\phi_i'(s_i)$'s) are 0 on average (i.e. "centered"), as they appear in the formula for the gradient. Typically, these methods require the introduction of additional "skip" connections (which bypass the nonlinearities of a given layer) in order to preserve the expressive power/efficiency of the network after these transformations are applied.

It is argued by Raiko et al. (2012) that the application of the centering transformation makes the Fisher of the resulting network closer to a diagonal matrix, and thus makes its gradient more closely resemble its natural gradient. However, this argument uses the strong approximating assumption that the correlations between various network-dependent quantities, such as the activities of different units within a given layer, are zero. In our notation, this would be like assuming that the $G_{i,i}$'s are diagonal, and that the $\bar{A}_{i,i}$'s are rank-1 plus a diagonal term. Indeed, using such an approximation within the block-diagonal version of K-FAC would yield an algorithm similar to standard centering, although without the need for skip connections (and hence similar to the version of centering proposed by Wiesler et al. (2014)).

As shown in Corollary 13, K-FAC can also be interpreted as using the gradient of a transformed network as its update direction, although one in which the g_i's and a_i's are both centered and *whitened* (with respect to the model's distribution). Intuitively, it is this whitening which accounts for the correlations between activities (or back-propagated gradients) within a given layer.

Ollivier (2013) proposed a neural network optimization method which uses a block-diagonal approximation of the Fisher, with the blocks corresponding to the incoming weights (and bias) of each unit. This method is similar to TONGA, except that it approximates the Fisher instead of the empirical Fisher (see Martens (2014) for a discussion of the difference between these). Because computing blocks of the Fisher is expensive (it requires k backpropagations, where k is the number of output units), this method uses a biased deterministic approximation which can be computed more efficiently, and is similar in spirit to the deterministic approximation used by LeCun et al. (1998). Note that while such an approximation could hypothetically be used within K-FAC to compute the

$G_{i,j}$'s, we have found that our basic unbiased stochastic approximation works nearly as well as the exact values in practice.

The work most closely related to ours is that of Heskes (2000), who proposed an approximation of the Fisher of feed-forward neural networks similar to our Kronecker-factored block-diagonal approximation $\breve{F}$ from Section 6.4.2, and used it to derive an efficient approximate natural-gradient based optimization method by exploiting the identity $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$. K-FAC differs from Heskes' method in several important ways which turn out to be crucial to it working well in practice.

In Heskes' method, update damping is accomplished using a basic factored Tikhonov technique where γI is added to each $G_{i,i}$ and $\bar{A}_{i,i}$ for a fixed parameter $\gamma > 0$ which is set by hand. By contrast, K-FAC uses a factored Tikhonov technique where γ adapted dynamically as described in Section 6.6.6, combined with a re-scaling technique based on a local quadratic model computed using the exact Fisher (see Section 6.6.4). Note that the adaptation of γ is important since what constitutes a good or even merely acceptable value of γ will change significantly over the course of optimization. And the use of our re-scaling technique, or something similar to it, is also crucial as we have observed empirically that basic Tikhonov damping is incapable of producing high quality updates by itself, even when γ is chosen optimally at each iteration (see Figure 6.6 of Section 6.6.4).

Also, while Heskes' method computes the $G_{i,i}$'s exactly, K-FAC uses a stochastic approximation which scales efficiently to neural networks with much higher-dimensional outputs (see Section 6.5).

Other advances we have introduced include the more accurate block-tridiagonal approximation to the inverse Fisher, a parameter-free type of momentum (see Section 6.7), online estimation of the $G_{i,i}$ and $\bar{A}_{i,i}$ matrices, and various improvements in computational efficiency (see Section 6.8). We have found that each of these additional elements is important in order for K-FAC to work as well as it does in various settings.

Concurrently with this work Povey et al. (2015) has developed a neural network optimization method which uses a block-diagonal Kronecker-factored approximation similar to the one from Heskes (2000). This approach differs from K-FAC in numerous ways, including its use of the empirical Fisher (which doesn't work as well as the standard Fisher in our experience – see Section 6.5), and its use of only a basic factored Tikhonov damping technique without adaptive re-scaling or any form of momentum. One interesting idea introduced by Povey et al. (2015) is a particular method for maintaining an online low-rank plus diagonal approximation of the factor matrices for each block, which allows their inverses to be computed more efficiently (although subject to an approximation). While our experiments with similar kinds of methods for maintaining such online estimates found that they performed poorly in practice compared to the solution of refreshing the inverses only occasionally (see Section 6.8), the particular one developed by Povey et al. (2015) could potentially still work well, and may be especially useful for networks with very wide layers.

6.12 Heskes' interpretation of the block-diagonal approximation

Heskes (2000) discussed an alternative interpretation of the block-diagonal approximation which yields some useful insight to complement our own theoretical analysis. In particular, he observed that the block-diagonal Fisher approximation $\breve{F}$ is the curvature matrix corresponding to the following quadratic function which measures the difference between the new parameter value θ' and the current value θ:

$$D(\theta', \theta) = \frac{1}{2} \sum_{i=1}^{\ell} \mathrm{E}\left[(s_i - s_i')^\top G_{i,i} (s_i - s_i')\right]$$

Here, $s'_i = W'_i \bar{a}_{i-1}$, and the s_i's and $\bar{a}_i$'s are determined by θ and are independent of θ' (which determines the W'_i's).

$D(\theta', \theta)$ can be interpreted as a reweighted sum of squared changes of each of the s_i's. The reweighing matrix $G_{i,i}$ is given by

$$G_{i,i} = \mathrm{E}\left[g_i g_i^\top\right] = \mathrm{E}\left[F_{P^{(i)}_{y|s_i}}\right]$$

where $P^{(i)}_{y|s_i}$ is the network's predictive distribution as parameterized by s_i, and $F_{P^{(i)}_{y|s_i}}$ is its Fisher information matrix, and where the expectation is taken w.r.t. the distribution on s_i (as induced by the distribution on the network's input x). Thus, the effect of reweighing by the $G_{i,i}$'s is to (approximately) translate changes in s_i into changes in the predictive distribution over y, although using the expected/average Fisher $G_{i,i} = \mathrm{E}[F_{P^{(i)}_{y|s_i}}]$ instead of the more specific Fisher $F_{P^{(i)}_{y|s_i}}$.

Interestingly, if one used $F_{P^{(i)}_{y|s_i}}$ instead of $G_{i,i}$ in the expression for $D(\theta', \theta)$, then $D(\theta', \theta)$ would correspond to a basic layer-wise block-diagonal approximation of F where the blocks are computed exactly (i.e. without the Kronecker-factorizing approximation introduced in Section 6.3). Such an approximate Fisher would have the interpretation of being the Hessian w.r.t. θ' of either of the measures

$$\sum_{i=1}^{\ell} \mathrm{E}\left[\mathrm{KL}\left(P^{(i)}_{y|s_i} \,\|\, P^{(i)}_{y|s'_i}\right)\right] \qquad \text{or} \qquad \sum_{i=1}^{\ell} \mathrm{E}\left[\mathrm{KL}\left(P^{(i)}_{y|s'_i} \,\|\, P^{(i)}_{y|s_i}\right)\right]$$

Note that each term in either of these sums is a function measuring an intrinsic quantity (i.e. changes in the output distribution), and so overall these are intrinsic measures except insofar as they assume that θ is divided into ℓ independent groups that each parameterize one of the ℓ different predictive distributions (which are each conditioned on their respective a_{i-1}'s).

It is not clear whether $\breve{F}$, with its Kronecker-factorizing structure can similarly be interpreted as the Hessian of such a self-evidently intrinsic measure. If it could be, then this would considerably simplify the proof of our Theorem 11 (e.g. using the techniques of Arnold et al. (2011)). Note that $D(\theta', \theta)$ itself doesn't work, as it isn't obviously intrinsic. Despite this, as shown in Section 6.10, both $\breve{F}$ and our more advanced approximation $\hat{F}$ produce updates which have strong invariance properties.

6.13 Experiments

To investigate the practical performance of K-FAC we applied it to the 3 deep autoencoder optimization problems considered in Section 4.12, which use the "MNIST", "CURVES", and "FACES" datasets respectively (see Hinton and Salakhutdinov (2006) for a complete description of these datasets). Due to their high difficulty, performance on these problems has become a standard benchmark for neural network optimization methods (e.g. Martens, 2010; Vinyals and Povey, 2012; Sutskever et al., 2013). We included ℓ_2 regularization with a coefficient of $\nu = 10^{-5}$ in each of these three optimization problems (i.e. so that $\frac{\eta}{2}\|\theta\|_2^2$ was added to the objective), which is lower than what was used by Martens (2010) (and by extension Section 4.12), but higher than what was used by Sutskever, Martens, Dahl, and Hinton (2013).

As our baseline we used the version of SGD with momentum based on Nesterov's Accelerated Gradient (Nesterov, 1983b) described in Sutskever, Martens, Dahl, and Hinton (2013), which was calibrated to work well on these particular deep autoencoder problems. For each problem we followed the prescription given by

Sutskever, Martens, Dahl, and Hinton (2013) for determining the learning rate, and the increasing schedule for the decay parameter μ. We did not compare to methods based on diagonal approximations of the curvature matrix, as in our experience such methods tend not perform as well on these kinds of optimization problems as the baseline does (an observation which is consistent with the findings of Schraudolph (2002); Zeiler (2013)).

Our implementation of K-FAC used most of the efficiency improvements described in Section 6.8, except that all "tasks" were computed serially (and thus with better engineering and more hardware, a faster implementation could likely be obtained). Because the mini-batch size m tended to be comparable to or larger than the typical/average layer size d, we did not use the technique described at the end of Section 6.8 for accelerating the computation of the approximate inverse, as this only improves efficiency in the case where $m < d$, and will otherwise decrease efficiency.

Both K-FAC and the baseline were implemented using vectorized MATLAB code accelerated with the GPU package Jacket. All tests were performed on a single computer with a 4.4 Ghz 6 core Intel CPU and an NVidia GTX 580 GPU with 3GB of memory. Each method used the same initial parameter setting, which was generated using the "sparse initialization" technique from Martens (2010) (which was also used by Sutskever, Martens, Dahl, and Hinton (2013), and is briefly described in Section 4.13).

To help mitigate the detrimental effect that the noise in the stochastic gradient has on the convergence of the baseline (and to a lesser extent K-FAC as well) we used a exponentially decayed iterate averaging approach based loosely on Polyak averaging (e.g. Swersky et al., 2010). In particular, at each iteration we took the "averaged" parameter estimate to be the previous such estimate, multiplied by ξ, plus the new iterate produced by the optimizer, multiplied by $1 - \xi$, for $\xi = 0.99$. Since the training error associated with the optimizer's current iterate may sometimes be lower than the training error associated with the averaged estimate (which will often be the case when the mini-batch size m is very large), we report the minimum of these two quantities.

To be consistent with the numbers given in previous papers we report the reconstruction error instead of the actual objective function value (although these are almost perfectly correlated in our experience). And we report the error on the training set as opposed to the test set, as we are chiefly interested in optimization speed and not the generalization capabilities of the networks themselves.

In our first experiment we examined the relationship between the mini-batch size m and the per-iteration rate of progress made by K-FAC and the baseline on the MNIST problem. The results from this experiment are plotted in Figure 6.8. They strongly suggest that the per-iteration rate of progress of K-FAC tends to a superlinear function of m (which can be most clearly seen by examining the plots of training error vs training cases processed), which is to be contrasted with the baseline, where increasing m has a much smaller effect on the per-iteration rate of progress, and with K-FAC without momentum, where the per-iteration rate of progress seems to be a linear or slightly sublinear function of m. It thus appears that the main limiting factor in the convergence of K-FAC (with momentum applied) is the noise in the gradient, at least in later stages of optimization, and that this is not true of the baseline to nearly the same extent. This would seem to suggest that K-FAC, much more than SGD, would benefit from a massively parallel distributed implementation which makes use of more computational resources than a single GPU.

But even in the single CPU/GPU setting, the fact that the per-iteration rate of progress tends to a *superlinear* function of m, while the per-iteration computational cost of K-FAC is a roughly linear function of m, suggests that in order to obtain the best per-second rate of progress with K-FAC, we should use a rapidly increasing schedule for m. To this end we designed an exponentially increasing schedule for m, given by $m_i = \min(m_1 \exp((i - 1)/b), |S|)$, where i is the current iteration, $m_1 = 1000$, and where b is chosen so that $m_{500} = |S|$. The approach of increasing the mini-batch size in this way is analyzed by Friedlander and Schmidt (2012). Note that for other

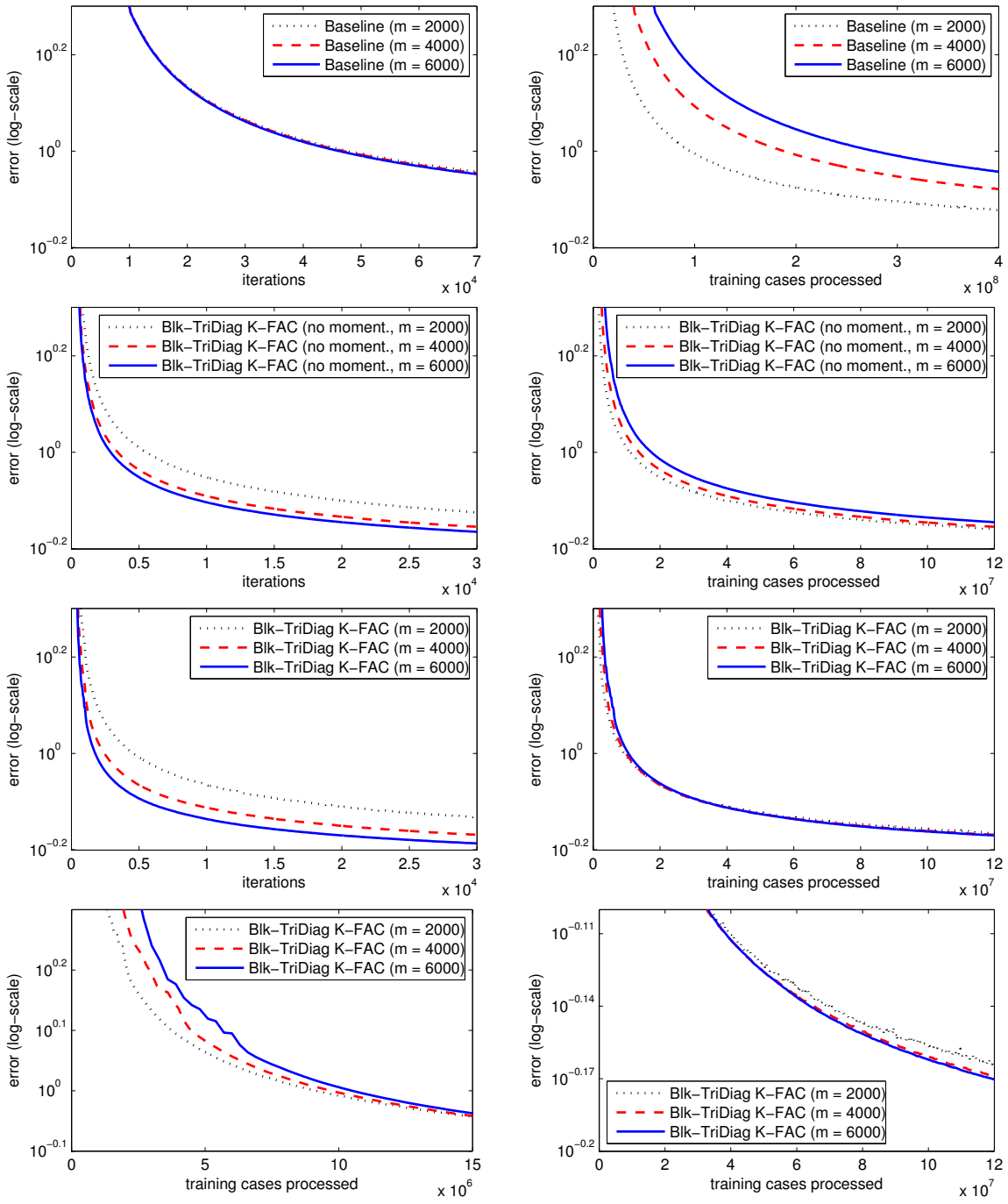

Figure 6.8: Results from our first experiment examining the relationship between the mini-batch size m and the per-iteration progress (**left column**) or per-training case progress (**right column**) progress made by K-FAC on the MNIST deep autoencoder problem. Here, "Blk-TriDiag K-FAC" is the block-tridiagonal version of K-FAC, and "Blk-Diag K-FAC" is the block-diagonal version, and "no moment." indicates that momentum was not used. The **bottom row** consists of zoomed-in versions of the right plot from the row above it, with the left plot concentrating on the beginning stage of optimization, and the right plot concentrating on the later stage. Note that the x-axes of these two last plots are at significantly different scales (10^6 vs 10^7).

neural network optimization problems, such as ones involving larger training datasets than these autoencoder problems, a more slowly increasing schedule, or one that stops increasing well before m reaches $|S|$, may be more appropriate. One may also consider using an approach similar to that of Byrd et al. (2012) for adaptively determining a suitable mini-batch size.

In our second experiment we evaluated the performance of our implementation of K-FAC versus the baseline on all 3 deep autoencoder problems, where we used the above described exponentially increasing schedule for m for K-FAC, and a fixed setting of m for the baseline and momentum-less K-FAC (which was chosen from a small range of candidates to give the best overall per-second rate of progress). The relatively high values of m chosen for the baseline ($m = 250$ for CURVES, and $m = 500$ for MNIST and FACES, compared to the $m = 200$ which was used by Sutskever, Martens, Dahl, and Hinton (2013)) reflect the fact that our implementation of the baseline uses a high-performance GPU and a highly optimized linear algebra package, which allows for many training cases to be efficiently processed in parallel. Indeed, after a certain point, making the mini-batches much smaller didn't result in a significant reduction in the baseline's per-iteration computation time.

Note that in order to process the very large mini-batches required for the exponentially increasing schedule without overwhelming the memory of the GPU, we partitioned the mini-batches into smaller "chunks" and performed all computations involving the mini-batches, or subsets thereof, one chunk at a time.

The results from this second experiment are plotted in Figures 6.9 and 6.10. For each problem K-FAC had a *per-iteration* rate of progress which was orders of magnitude higher than that of the baseline's (Figure 6.10), provided that momentum was used, which translated into an overall much higher *per-second* rate of progress (Figure 6.9), despite the higher cost of K-FAC's iterations (due mostly to the much larger mini-batch sizes used). Note that Polyak averaging didn't produce a significant increase in convergence rate of K-FAC in this second experiment (actually, it hurt a bit) as the increasing schedule for m provided a much more effective (although expensive) solution to the problem of noise in the gradient.

The importance of using some form of momentum on these problems is emphasized in these experiments by the fact that without the momentum technique developed in Section 6.7, K-FAC wasn't significantly faster than the baseline (which itself used a strong form of momentum). These results echo those of Sutskever, Martens, Dahl, and Hinton (2013), who found that without momentum, SGD was orders of magnitude slower on these particular problems. Indeed, if we had included results for the baseline without momentum they wouldn't even have appeared in the axes boundaries of the plots in Figure 6.9.

Recall that the type of momentum used by K-FAC compensates for the inexactness of our approximation to the Fisher by allowing K-FAC to build up a better solution to the exact quadratic model minimization problem (defined using the exact Fisher) across many iterations. Thus, if we were to use a much stronger approximation to the Fisher when computing our update proposals Δ, the benefit of using this type of momentum would have likely been much smaller than what we observed. One might hypothesize that it is the particular type of momentum used by K-FAC that is mostly responsible for its advantages over the SGD baseline. However in our testing we found that for SGD the more conventional type of momentum used by Sutskever, Martens, Dahl, and Hinton (2013) performs significantly better.

From Figure 6.10 we can see that the block-tridiagonal version of K-FAC has a per-iteration rate of progress which is typically 25% to 40% larger than the simpler block-diagonal version. This observation provides empirical support for the idea that the block-tridiagonal approximate inverse Fisher $\hat{F}^{-1}$ is a more accurate approximation of F^{-1} than the block-diagonal approximation $\breve{F}^{-1}$. However, due to the higher cost of the iterations in the block-tridiagonal version, its overall per-second rate of progress seems to be only moderately higher than the block-diagonal version's, depending on the problem.

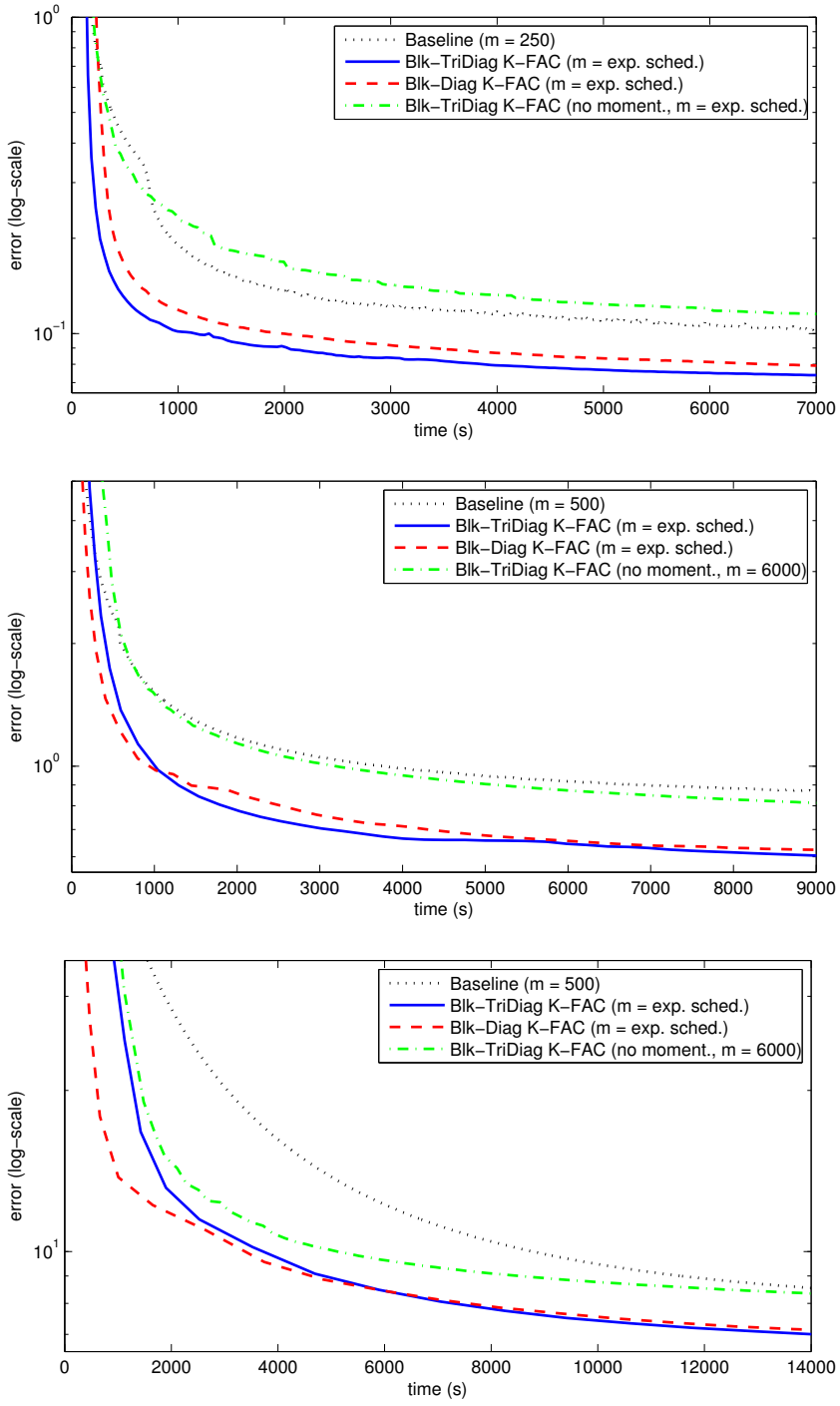

Figure 6.9: Results from our second experiment showing training error versus computation time on the CURVES (**top**), MNIST (**middle**), and FACES (**bottom**) deep autoencoder problems.

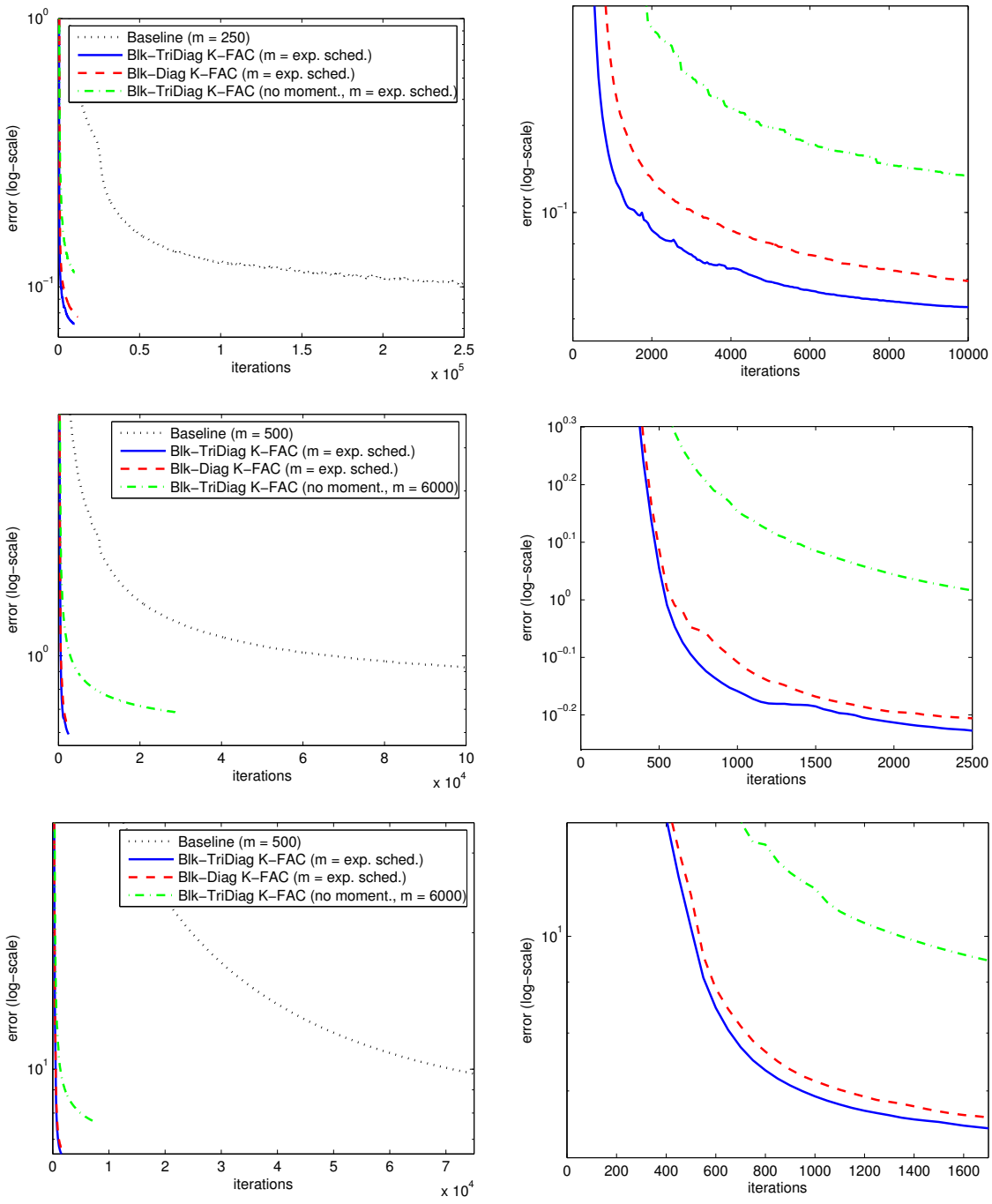

Figure 6.10: More results from our second experiment showing training error versus iteration on the CURVES (**top** row), MNIST (**middle** row), and FACES (**bottom** row) deep autoencoder problems. The plots on the right are zoomed in versions of those on the left which highlight the difference in per-iteration progress made by the different versions of K-FAC.

Note that while matrix-matrix multiplication, matrix inverse, and SVD computation all have the same computational complexity, in practice their costs differ significantly (in increasing order as listed). Computation of the approximate Fisher inverse, which is performed in our experiments once every 20 iterations (and for the first 3 iterations), requires matrix inverses for the block-diagonal version, and SVDs for the block-tridiagonal version. For the FACES problem, where the layers can have as many as 2000 units, this accounted for a significant portion of the difference in the average per-iteration computational cost of the two versions (as these operations must be performed on 2000×2000 sized matrices).

While our results suggest that the block-diagonal version is probably the better option overall due to its greater simplicity (and comparable per-second progress rate), the situation may be different given a more efficient implementation of K-FAC where the more expensive SVDs required by the tri-diagonal version are computed approximately and/or in parallel with the other tasks, or perhaps even while the network is being optimized.

Our results also suggest that K-FAC may be much better suited than the SGD baseline for a massively distributed implementation, since it would require far fewer synchronization steps (by virtue of the fact that it requires far fewer iterations).

Chapter 7

Conclusions

7.1 Summary of contributions

In this thesis we reviewed and discussed approaches for optimizing neural networks, with a focus on approximate-Newton/2nd-order approaches, and contributed some novel theoretical analyses and methods. This work culminated with the development of the K-FAC method in the final chapter, which we showed has significant advantages over SGD with momentum.

A more detailed summary of our contributions is as follows.

Chapter 3 discussed the theoretical and practical difficulties associated with optimizing deep and recurrent neural networks, such as vanishing/exploding gradients, and presented arguments for why 2nd-order methods may help alleviate these. In particular, it was shown that 2nd-order optimization can, at least in principle, boost gradient signals corresponding to long-range information in RNNs (in the case of vanishing gradients), thus making it easier for the network to learn to make predictions based on such information. Chapter 3 also presented a detailed review of some of the 2nd-order methods which have been previously proposed for neural network optimization, identifying many of their potential strengths and weaknesses.

Chapter 4 developed a powerful 2nd-order optimization method called the Hessian-free method (HF) by combining classical optimization techniques, such as the truncated Newton method, with some existing ideas developed in the context of neural network optimization (and reviewed in Chapter 3), such as the Gauss-Newton matrix, as well as some novel ideas particular to HF, such as structural damping. A detailed discussion and analysis was given of the various elements of this approach, such as damping, preconditioning, and the use of linear conjugate gradient (CG). Experiments demonstrated that the method can effectively train recurrent neural networks on long-range dependency problems, as well as deep autoencoders, which were both tasks believed to be impossible for purely local optimization methods when these results were originally published. While subsequent research suggested that SGD with momentum, provided it is carefully initialized and tuned, is a faster and more practical solution than HF for optimizing deep neural networks, these results nevertheless demonstrated that updates computed using high-quality models of the local curvature can make much more progress per iteration than the standard gradient.

Chapter 5 discussed the natural gradient approach and established important links to 2nd-order methods. In particular, it was shown that in many important cases that arise in neural network optimization, natural gradient descent is equivalent to 2nd-order optimization using the generalized Gauss-Newton matrix (GGN), so that HF and related methods can be seen as natural gradient methods, albeit much more practical and robust than basic natural

gradient descent. It was also observed that this perspective conversely lets us view natural gradient methods as 2nd-order methods, justifying the application of various damping techniques when designing them, such as those discussed and developed in Chapter 4. Various properties of natural gradient descent, such as its invariance to parameterization and its asymptotic convergence speed were critically analyzed, and new results were presented that consider these properties for more general 2nd-order methods.

Chapter 6 developed a powerful 2nd-order optimization method based on a high-quality approximation to the Fisher that can be inverted efficiently, despite being neither sparse nor low-rank. This approximation was derived by approximating each layer-wise block of the Fisher as the Kronecker product between two much smaller matrices. Utilizing the connections between natural gradient descent and 2nd-order methods made in Chapter 5, this approximation was developed into a robust optimization method using many of the ideas present in HF (Chapter 4). It was then demonstrated that this method can be much faster in practice than SGD with momentum at optimizing deep neural networks.

7.2 Future directions

Because K-FAC utilizes the special structure of neural networks (unlike SGD or HF), it is not directly applicable to other neural architectures like RNNs or convolutional neural networks (CNNs). Thus a potentially very productive line of research would be to extend K-FAC to these (and other) architectures.

The problem of stochasticity in the gradient estimates used by K-FAC was handled in the experiments via the method of increasing the mini-batch size m throughout optimization, in combination with a standard iterate averaging approach. Forcing the learning rate to decrease according to a schedule is a solution that can also work in practice. However, it may be worthwhile to develop a version of K-FAC that can handle stochasticity in a better or more principled way, or according to some kind of adaptive rule that considers the empirically observed amount/shape of noise in the gradient.

Finally, it would also be interesting to develop an MCMC method from K-FAC that can realize a more Bayesian approach to neural network learning, as in the work of Neal (1996), but while benefiting from the ability to generate proposals according to a good estimate of the local curvature. Potential advantages of such a method over standard K-FAC could include resilience to overfitting, along with the ability to avoid some of the global optimization problems associated with standard gradient methods, such as the inability to break symmetries, or to salvage dead or saturated units.

Bibliography

Almeida, L. B., T. Langlois, J. D. Amaral, and A. Plakhov (1998). "Parameter adaptation in stochastic optimization". In: *On-line learning in neural networks*. Ed. by D. Saad. Cambridge University Press, pp. 111–134.

Amari, S. (1967). "Theory of adaptive pattern classifiers". In: *IEEE Transactions on Electronic Computers* 16.3, pp. 299–307.

Amari, S. and H. Nagaoka (2000). *Methods of Information Geometry*. Vol. 191. Translations of Mathematical monographs. Oxford University Press.

Amari, S.-I. (1998). "Natural gradient works efficiently in learning". In: *Neural Computation* 10.2, pp. 251–276.

Arnold, L., A. Auger, N. Hansen, and Y. Ollivier (2011). "Information-Geometric Optimization Algorithms: A Unifying Picture via Invariance Principles". In: arXiv: `1106.3708 [math.OC]`.

Ba, J. and D. Kingma (2015). "Adam: A Method for Stochastic Optimization". In: *ICLR*.

Bartels, R. H. and G. Stewart (1972). "Solution of the matrix equation $AX + XB = C$". In: *Communications of the ACM* 15.9, pp. 820–826.

Becker, S. and Y. LeCun (1989). "Improving the Convergence of Back-Propagation Learning with Second Order Methods". In: *Proceedings of the 1988 Connectionist Models Summer School*, pp. 29–37.

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

Bengio, Y., N. Boulanger-Lewandowski, and R. Pascanu (2013). "Advances in Optimizing Recurrent Networks". In: *Proc. ICASSP 38*.

Bergstra, J., O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio (2010). "Theano: a CPU and GPU math expression compiler". In: *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Vol. 4.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer.

Bishop, C. (1992). "Exact calculation of the Hessian matrix for the multilayer perceptron". In: *Neural Computation* 4.4, pp. 494–501.

Blum, A. and R. L. Rivest (1988). "Training a 3-node neural network is NP-complete". In: *Proceedings of the first annual workshop on Computational learning theory*. COLT '88, pp. 9–18.

Bordes, A., L. Bottou, and P. Gallinari (2009). "SGD-QN: Careful Quasi-Newton Stochastic Gradient Descent". In: *Journal of Machine Learning Research* 10, pp. 1737–1754.

Bordes, A., L. Bottou, P. Gallinari, J. Chang, and S. A. Smith (2010). "Erratum: SGDQN is Less Careful than Expected". In: *Journal of Machine Learning Research* 11, pp. 2229–2240.

Bottou, L. and O. Bousquet (2011). "The Tradeoffs of Large Scale Learning". In: *Optimization for Machine Learning*. Ed. by S. Sra, S. Nowozin, and S. J. Wright. MIT Press, pp. 351–368.

Bottou, L. and Y. LeCun (2005). "On-line learning for very large data sets". In: *Appl. Stoch. Model. Bus. Ind.* 21.2, pp. 137–151. ISSN: 1524-1904.

Boulanger-Lewandowski, N., Y. Bengio, and P. Vincent (2012). "Modeling Temporal Dependencies in High-Dimensional Sequences: Application to Polyphonic Music Generation and Transcription". In: *Proceedings of the 29th International Conference on Machine Learning (ICML)*.

Byrd, R., G. Chin, W. Neveitt, and J. Nocedal (2011). "On the use of stochastic Hessian information in optimization methods for machine learning". In: *SIAM Journal on Optimization* 21, p. 977.

Byrd, R. H., G. M. Chin, J. Nocedal, and Y. Wu (2012). "Sample size selection in optimization methods for machine learning". In: *Mathematical programming* 134.1, pp. 127–155.

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

Chu, K.-w. E. (1987). "The solution of the matrix equations $AXB - CXD = E$ and $(YA - DZ, YC - BZ) = (E, F)$". In: *Linear Algebra and its Applications* 93.0, pp. 93 –105.

Chung, I.-H., T. N. Sainath, B. Ramabhadran, M. Picheny, J. Gunnels, V. Austel, U. Chauhari, and B. Kingsbury (2014). "Parallel Deep Neural Network Training for Big Data on Blue Gene/Q". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, pp. 745–753.

Darken, C. and J. E. Moody (1990). "Note on Learning Rate Schedules for Stochastic Optimization". In: *Advances in Neural Information Processing Systems*, pp. 832–838.

Darken, C., J. Chang, J. C. Z, and J. Moody (1992). "Learning Rate Schedules For Faster Stochastic Gradient Search". In: *Neural networks for signal processing*. Vol. 2. IEEE Press.

Défossez, A. and F. Bach (2014). "Constant step size least-mean-square: Bias-variance trade-offs and optimal sampling distributions". In: arXiv: `1412.0156 [cs.LG]`.

Desjardins, G., R. Pascanu, A. Courville, and Y. Bengio (2013). "Metric-Free Natural Gradient for Joint-Training of Boltzmann Machines". In: *International Conference on Learning Representations (ICLR'2013)*.

Duane, S., A. D. Kennedy, B. J. Pendleton, and D. Roweth (1987). "Hybrid Monte Carlo". In: *Physics Letters B* 195.2, pp. 216 –222.

Duchi, J. C., E. Hazan, and Y. Singer (2011). "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." In: *Journal of Machine Learning Research* 12, pp. 2121–2159.

Erhan, D., Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio (2010). "Why does unsupervised pre-training help deep learning?" In: *The Journal of Machine Learning Research* 11, pp. 625–660.

Flammarion, N. and F. Bach (2015). "From Averaging to Acceleration, There is Only a Step-size". In: arXiv: `1504.01577 [stat.ML]`.

Forster, J. (2002). "A linear lower bound on the unbounded error probabilistic communication complexity". In: *J. Comput. Syst. Sci.* 65.4, pp. 612–625.

Friedlander, M. P. and M. W. Schmidt (2012). "Hybrid Deterministic-Stochastic Methods for Data Fitting". In: *SIAM J. Scientific Computing* 34.3.

Frostig, R., R. Ge, S. M. Kakade, and A. Sidford (2014). "Competing with the Empirical Risk Minimizer in a Single Pass". In: arXiv: `1412.6606 [stat.ML]`.

Gardiner, J. D., A. J. Laub, J. J. Amato, and C. B. Moler (1992). "Solution of the Sylvester Matrix Equation $AXB^T + CXD^T = E$". In: *ACM Trans. Math. Softw.* 18.2, pp. 223–231.

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

Gould, N., S. Lucidi, M. Roma, and P. Toint (1999). "Solving the trust-region subproblem using the Lanczos method". In: *SIAM Journal on Optimization* 9.2, pp. 504–525.

Graff, P., F. Feroz, M. P. Hobson, and A. Lasenby (2012). "BAMBI: blind accelerated multimodal Bayesian inference". In: *Monthly Notices of the Royal Astronomical Society* 421, pp. 169–180.

Graff, P., F. Feroz, M. P. Hobson, and A. N. Lasenby (2014). "SKYNET: an efficient and robust neural network training tool for machine learning in astronomy." In: *Monthly Notices of the Royal Astronomical Society* 441, pp. 1741–1759.

Grosse, R. and R. Salakhutdinov (2015). "Scaling up natural gradient by factorizing Fisher information". In: *Proceedings of the 32nd International Conference on Machine Learning (ICML)*.

Hajnal, A., W. Maass, P. Pudlák, M. Szegedy, and G. Turán (1993). "Threshold circuits of bounded depth". In: *J. Comput. System. Sci.* 46, pp. 129–154.

Hansen, P. and D. O'Leary (1993). "The use of the L-curve in the regularization of discrete ill-posed problems". In: *SIAM Journal on Scientific Computing* 14, p. 1487.

Hazan, E., A. Agarwal, and S. Kale (2007). "Logarithmic Regret Algorithms for Online Convex Optimization". In: *Maching Learning* 69.2-3, pp. 169–192.

Heskes, T. (2000). "On "Natural" Learning and Pruning in Multilayered Perceptrons." In: *Neural Computation* 12.4, pp. 881–901.

Hestenes, M. and E. Stiefel (1952). *Methods of conjugate gradients for solving linear systems*.

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

Hinton, G. E., N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2012). "Improving neural networks by preventing co-adaptation of feature detectors". In: *CoRR* abs/1207.0580.

Hochreiter, S. and J. Schmidhuber (1997). "Long short-term memory". In: *Neural Computation* 9.8, pp. 1735–1780.

Hochreiter, S. (1991). "Untersuchungen zu dynamischen neuronalen Netzen". In: *Diploma, Technische Universität München*.

Hochreiter, S., F. F. Informatik, Y. Bengio, P. Frasconi, and J. Schmidhuber (2000). "Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies". In: *Field Guide to Dynamical Recurrent Networks*. Ed. by J. Kolen and S. Kremer. IEEE Press.

Hornik, K., M. Stinchcombe, and H. White (1989). "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5, pp. 359–366.

Jaeger, H. and H. Haas (2004). "Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication". In: *Science* 304, pp. 78–80.

Johnson, R. and T. Zhang (2013). "Accelerating Stochastic Gradient Descent using Predictive Variance Reduction". In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 315–323.

Kingsbury, B., T. N. Sainath, and H. Soltau (2012). "Scalable Minimum Bayes Risk Training of Deep Neural Network Acoustic Models Using Distributed Hessian-free Optimization." In: *INTERSPEECH*. ISCA.

Kiros, R. (2013). "Training Neural Networks with Stochastic Hessian-Free Optimization". In: *International Conference on Learning Representations (ICLR)*.

Komaroff, N (1988). "Simultaneous eigenvalue lower bounds for the Lyapunov matrix equation". In: *Automatic Control, IEEE Transactions on* 33.1, pp. 126–128.

— (1992). "Upper summation and product bounds for solution eigenvalues of the Lyapunov matrix equation". In: *IEEE transactions on automatic control* 37.7, pp. 1040–1042.

Kwon, W. H., Y. S. Moon, and S. C. Ahn (1996). "Bounds in algebraic Riccati and Lyapunov equations: a survey and some new results". In: *International Journal of Control* 64.3, pp. 377–389.

Le Roux, N. and A. Fitzgibbon (2010). "A fast natural Newton method". In: *Proceedings of the 27th International Conference on Machine Learning (ICML)*.

Le Roux, N., P.-a. Manzagol, and Y. Bengio (2008). "Topmoumoute online natural gradient algorithm". In: *Advances in Neural Information Processing Systems 20*. MIT Press, pp. 849–856.

Le Roux, N., M. W. Schmidt, and F. Bach (2012). "A Stochastic Gradient Method with an Exponential Convergence Rate for Finite Training Sets". In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 2672–2680.

LeCun, Y., L. Bottou, G. Orr, and K. Müller (1998). "Efficient backprop". In: *Neural networks: Tricks of the trade*, pp. 546–546.

Li, R.-C. (2005). *Sharpness in rates of convergence for CG and symmetric Lanczos methods*. Tech. rep. 05-01. Department of Mathematics, University of Kentucky.

Liu, D. C. and J. Nocedal (1989). "On the limited memory BFGS method for large scale optimization". In: *Mathematical programming* 45.1-3, pp. 503–528.

Martens, J. (2010). "Deep learning via Hessian-free optimization". In: *Proceedings of the 27th International Conference on Machine Learning (ICML)*.

— (2014). "New insights and perspectives on the natural gradient method". In: arXiv: `1412.1193 [cs.LG]`.

Martens, J. and I. Sutskever (2011). "Learning Recurrent Neural Networks with Hessian-Free Optimization". In: *Proceedings of the 28th International Conference on Machine Learning (ICML)*, pp. 1033–1040.

— (2012). "Training deep and recurrent networks with Hessian-free optimization". In: *Neural Networks: Tricks of the Trade*. Springer, pp. 479–535.

Martens, J., I. Sutskever, and K. Swersky (2012). "Estimating the Hessian by Backpropagating Curvature". In: *Proceedings of the 29th International Conference on Machine Learning (ICML)*.

Martens, J. (2016). "Beyond Universality: On the Expressive Efficiency of Deep Models". In: *In preparation*.

Martens, J. and R. Grosse (2015). "Optimizing Neural Networks with Kronecker-factored Approximate Curvature". In: *Proceedings of the 32nd International Conference on Machine Learning (ICML)*.

Martens, J. and V. Medabalimi (2014). "On the Expressive Efficiency of Sum Product Networks". In: arXiv: `1411.7717 [cs.LG]`.

Martens, J., A. Chattopadhyay, T. Pitassi, and R. Zemel (2013). "On the representational efficiency of Restricted Boltzmann Machines". In: *Advances in Neural Information Processing Systems (NIPS)*.

Moré, J. (1978). "The Levenberg-Marquardt algorithm: implementation and theory". In: *Numerical analysis*, pp. 105–116.

Moré, J. and D. Sorensen (1983). "Computing a trust region step". In: *SIAM Journal on Scientific and Statistical Computing* 4, p. 553.

Murata, N. (1998). "A Statistical Study of On-line Learning". In: *On-line Learning in Neural Networks*. Ed. by D. Saad. Cambridge University Press, pp. 63–92.

Nash, S. (1984). "Newton-type minimization via the Lanczos method". In: *SIAM Journal on Numerical Analysis*, pp. 770–788.

— (2000). "A survey of truncated-Newton methods". In: *Journal of Computational and Applied Mathematics* 124.1, pp. 45–59.

Neal, R. M. (1996). *Bayesian Learning for Neural Networks*. Lecture Notes in Statistics. Springer-Verlag.

Nesterov, Y. (1983a). "A method for unconstrained convex minimization problem with the rate of convergence $O(1/k^2)$". In: *Doklady AN SSSR*. Vol. 269. 3, pp. 543–547.

Nesterov, Y. (1983b). "A method of solving a convex programming problem with convergence rate $\mathcal{O}(1/\sqrt{k})$". In: *Soviet Mathematics Doklady* 27, pp. 372–376.

Nocedal, J. and S. J. Wright (2006). *Numerical optimization*. 2. ed. Springer.

Ollivier, Y. (2013). "Riemannian metrics for neural networks". In: arXiv: `1303.0818 [cs.NE]`.

Orr, G. B. and T. K. Leen (1996). "Using curvature information for fast stochastic search". In: *In Advances in Neural Information Processing Systems 9*. MIT Press, pp. 606–612.

Orr, G. B. (1995). "Dynamics and algorithms for stochastic search". UMI Order No. GAX96-08998. PhD thesis. Beaverton, OR, USA.

Pan, V. and R. Schreiber (1991). "An Improved Newton Iteration for the Generalized Inverse of a Matrix, with Applications". In: *SIAM Journal on Scientific and Statistical Computing* 12.5, pp. 1109–1130.

Park, H., S.-I. Amari, and K. Fukumizu (2000). "Adaptive natural gradient learning algorithms for various stochastic models". In: *Neural Networks* 13.7, pp. 755–764.

Pascanu, R. and Y. Bengio (2014). "Revisiting Natural Gradient for Deep Networks". In: *International Conference on Learning Representations*.

Pearlmutter, B. (1994). "Fast exact multiplication by the Hessian". In: *Neural Computation* 6.1, pp. 147–160.

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

Polyak, B. T. and A. B. Juditsky (1992). "Acceleration of Stochastic Approximation by Averaging". In: *SIAM J. Control Optim.* 30.4.

Polyak, B. (1964). "Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics* 4.5, pp. 1 –17. ISSN: 0041-5553.

Pourahmadi, M. (1999). "Joint mean-covariance models with applications to longitudinal data: unconstrained parameterisation". In: *Biometrika* 86.3, pp. 677–690.

— (2011). "Covariance Estimation: The GLM and Regularization Perspectives". In: *Statistical Science* 26.3, pp. 369–387.

Povey, D., X. Zhang, and S. Khudanpur (2015). "Parallel training of DNNs with Natural Gradient and Parameter Averaging". In: *International Conference on Learning Representations: Workshop track*.

Raiko, T., H. Valpola, and Y. LeCun (2012). "Deep Learning Made Easier by Linear Transformations in Perceptrons". In: *AISTATS*. Vol. 22. JMLR Proceedings, pp. 924–932.

Rattray, M. and D. Saad (1998). "Incorporating curvature information into on-line learning". In: *On-line learning in neural networks*. Ed. by D. Saad. New York, NY, USA: Cambridge University Press, pp. 183–207.

Riedmiller, M. and H. Braun (1993). "A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm". In: *IEEE International Conference on Neural Networks*, pp. 586–591.

Rumelhart, D., G. Hinton, and R. Williams (1986). "Learning representations by back-propagating errors". In: *Nature* 323.6088, pp. 533–536.

Sainath, T. N., L. Horesh, B. Kingsbury, A. Y. Aravkin, and B. Ramabhadran (2013a). "Accelerating Hessian-free optimization for Deep Neural Networks by implicit preconditioning and sampling". In: *IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU), 2013*, pp. 303–308.

Sainath, T. N., B. Kingsbury, A.-r. Mohamed, G. E. Dahl, G. Saon, H. Soltau, T. Beran, A. Y. Aravkin, and B. Ramabhadran (2013b). "Improvements to Deep Convolutional Neural Networks for LVCSR". In: *IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU), 2013*, pp. 315–320.

Sainath, T. N., B. Kingsbury, H. Soltau, and B. Ramabhadran (2013c). "Optimization Techniques to Improve Training Speed of Deep Neural Networks for Large Speech Tasks". In: *IEEE Transactions on Audio, Speech & Language Processing* 21.11, pp. 2267–2276.

Scarpetta, S., M. Rattray, and D. Saad (1999). "Matrix momentum for practical natural gradient learning". In: *Journal of Physics A: Mathematical and General* 32.22, p. 4047.

Schaul, T., S. Zhang, and Y. LeCun (2013). "No More Pesky Learning Rates". In: *Proceedings of the 30th International Conference on Machine Learning (ICML)*.

Schraudolph, N. N. (1998). "Centering Neural Network Gradient Factors". In: *Neural Networks: Tricks of the Trade*. Ed. by G. B. Orr and K.-R. Müller. Vol. 1524. Lecture Notes in Computer Science. Berlin: Springer Verlag, pp. 207–226.

— (1999). "Local Gain Adaptation in Stochastic Gradient Descent". In: *In Proc. Intl. Conf. Artificial Neural Networks*, pp. 569–574.

— (2002). "Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent". In: *Neural Computation* 14.

Schraudolph, N. N. and T. Graepel (2002). "Towards Stochastic Conjugate Gradient Methods". In: *Proc. 9^{th} Intl. Conf. Neural Information Processing (ICONIP)*, pp. 853–856.

Schraudolph, N. N., J. Yu, and S. Günter (2007). "A stochastic Quasi-Newton Method for online convex optimization". In: *In Proceedings of 11th International Conference on Artificial Intelligence and Statistics*.

Shewchuk, J. R. (1994). *An introduction to the conjugate gradient method without the agonizing pain*.

Silva, F. M. and L. B. Almeida (1990). "Acceleration Techniques for the Backpropagation Algorithm". In: *Proceedings of the EURASIP Workshop 1990 on Neural Networks*, pp. 110–119. ISBN: 3-540-52255-7.

Simoncini, V. (2014). "Computational methods for linear matrix equations".

Smith, R. (1968). "Matrix Equation $XA + BX = C$". In: *SIAM J. Appl. Math.* 16.1, pp. 198 –201.

Steihaug, T. (1983). "The conjugate gradient method and trust regions in large scale optimization". In: *SIAM Journal on Numerical Analysis* 20.3, pp. 626–637.

Sutskever, I., J. Martens, and G. Hinton (2011). "Generating Text with Recurrent Neural Networks". In: *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. Bellevue, Washington, USA, pp. 1017–1024. ISBN: 978-1-4503-0619-5.

Sutskever, I., J. Martens, G. Dahl, and G. Hinton (2013). "On the Importance of Initialization and Momentum in Deep Learning". In: *Proceedings of the 30th International Conference on Machine Learning (ICML)*.

Swersky, K., B. Chen, B. Marlin, and N. de Freitas (2010). "A tutorial on stochastic approximation algorithms for training Restricted Boltzmann Machines and Deep Belief Nets". In: *Information Theory and Applications Workshop (ITA), 2010*, pp. 1–10.

Tieleman, T. and G. Hinton (2012). *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning.

Toint, P. (1981). "Towards an efficient sparsity exploiting Newton method for minimization". In: *Sparse matrices and their uses*, p. 1981.

Van Loan, C. F. (2000). "The ubiquitous Kronecker product". In: *Journal of computational and applied mathematics* 123.1, pp. 85–100.

Vatanen, T., T. Raiko, H. Valpola, and Y. LeCun (2013). "Pushing Stochastic Gradient towards Second-Order Methods – Backpropagation Learning with Transformations in Nonlinearities". In: arXiv: `1301.3476 [cs.LG]`.

Vinyals, O. and D. Povey (2012). “Krylov Subspace Descent for Deep Learning”. In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*.

Wang, S.-D., T.-S. Kuo, and C.-F. Hsu (1986). “Trace bounds on the solution of the algebraic matrix Riccati and Lyapunov equation”. In: *Automatic Control, IEEE Transactions on* 31.7, pp. 654–656.

Wengert, R. (1964). “A simple automatic derivative evaluation program”. In: *Communications of the ACM* 7.8, pp. 463–464.

Wiegerinck, W., A. Komoda, and T. Heskes (1994). “Stochastic dynamics of learning with momentum in neural networks”. In: *Journal of Physics A* 27, pp. 4425–4437.

Wiesler, S., A. Richard, R. Schlüter, and H. Ney (2014). “Mean-normalized stochastic gradient for large-scale deep learning”. In: *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 180–184.

Yuan, Y. (2000). “On the truncated conjugate gradient method”. In: *Mathematical Programming* 87.3, pp. 561–573.

Zeiler, M. D. (2013). “ADADELTA: An Adaptive Learning Rate Method”. In: arXiv: 1212.5701 [cs.LG].

Appendix A

Supplemental appendices for Chapter 5

A.1 Extra derivations for Theorem 1

Given $\alpha_k = 1/(k+1)$ and $\lambda_n\left(B^{-1}H^*\right) > \frac{1}{2}$ Murata (1998) shows that

$$V_k = \frac{1}{k}\left(\Xi_{B^{-1}H^*} - I\right)^{-1}\left(B^{-1}\Sigma_g(\theta^*)B^{-1}\right) + o\left(\frac{1}{k}\right)$$

In this section we will derive the following more detailed asymptotic expression for V_k which is accurate up to terms of order $1/k^2$:

$$V_k = \frac{1}{k}\left(\Xi_{B^{-1}H^*} - I\right)^{-1}\left(B^{-1}\Sigma_g(\theta^*)B^{-1}\right) - \frac{1}{k^2}\left(\Xi_{B^{-1}H^*} - 2I\right)^{-1}\left(B^{-1}\Sigma_g(\theta^*)B^{-1}\right) + \mathcal{O}\left(\frac{1}{k^3}\right)$$

To derive this expression we will make use of the following recursive expression for V_k (which holds under the hypotheses of Theorem 1 concerning h):

$$\begin{aligned} V_{k+1} &= V_k - \alpha_k(B^{-1}H^*V_k + V_kH^*B^{-1}) + \alpha_k^2 B^{-1}\Sigma_g(\theta^*)B^{-1} \\ &= \left(I - \Xi_{\alpha_k B^{-1}H^*}\right)(V_k) + \alpha_k^2 B^{-1}\Sigma_g(\theta^*)B^{-1} \end{aligned} \tag{A.1}$$

A similar expression to this one is derived by Murata (1998) in their Lemmas 2 & 3, but included several extra terms on the RHS. The presence of the first set of terms is due to the approximation of h by its own 2nd-order Taylor series expansion, and we can drop these since we are assuming that h is *exactly* quadratic. Note that Murata (1998) also drops these terms when giving their Lemma 3, although they only justify this in a non-rigorous way by claiming that these terms can be ignored in the subsequent analysis. While this is likely true for some of the subsequent results like their Theorem 4 (covering the $\alpha_k = 1/(k+1)$ case), it is likely false for their Theorem 1 (covering the constant α_k case).

The second extra term is of the form $-\alpha_k^2 B^{-1}H^*(\mathrm{E}[\theta_t] - \theta^*)(\mathrm{E}[\theta_t] - \theta^*)^\top H^* B^{-1}$, and appears in Lemma 3 of Murata (1998). The presence of this extra term is due to our use of a slightly different assumption regarding the gradient noise made than is made by Murata (1998). In particular, Murata (1998) assumes that the second order statistics of the gradient, i.e. $\mathrm{E}[g(\theta)g(\theta)^\top] = \Sigma_g(\theta) + \mathrm{E}[g(\theta)]\,\mathrm{E}[g(\theta)]^\top$, is constant, whereas we assume that the covariance $\Sigma_g(\theta)$ is constant. Our assumption is arguably more realistic and also happens to simplify the analysis. By adopting it and making minor modifications to the proof of Lemma 3 of Murata (1998) as appropriate, one

can easily derive our eqn. A.1.

In the remainder of this subsection we will compute the asymptotic value of V_k and also compute the constant on the $\Omega(1/k^2)$ term, which will be useful in the proof of Theorem 6 (note that an expression for this term doesn't appear in Murata (1998)).

We start by expanding

$$V_k = \frac{1}{k}C_1 + \frac{1}{k^2}C_2 + \frac{1}{k^3}C_3 + \mathcal{O}\left(\frac{1}{k^4}\right)$$

Plugging this into eqn. A.1 and using $\alpha_k = 1/(k+1)$ we have

$$\begin{aligned}
&\frac{1}{k+1}C_1 + \frac{1}{(k+1)^2}C_2 + \frac{1}{(k+1)^3}C_3 + \mathcal{O}\left(\frac{1}{k^4}\right) \\
&\quad = \left(I - \frac{1}{k+1}\Xi_{B^{-1}H^*}\right)\left(\frac{1}{k}C_1 + \frac{1}{k^2}C_2 + \frac{1}{k^3}C_3 + \mathcal{O}\left(\frac{1}{k^4}\right)\right) + \frac{1}{(k+1)^2}B^{-1}\Sigma_g(\theta^*)B^{-1}
\end{aligned}$$

Rearrangement gives

$$\begin{aligned}
\left(\frac{1}{k(k+1)}\Xi_{B^{-1}H^*} + \left(\frac{1}{k+1} - \frac{1}{k}\right)I\right)(C_1) + \left(\frac{1}{k^2(k+1)}\Xi_{B^{-1}H^*} + \left(\frac{1}{(k+1)^2} - \frac{1}{k^2}\right)I\right)(C_2) \\
+ \left(\frac{1}{k^3(k+1)}\Xi_{B^{-1}H^*} + \left(\frac{1}{(k+1)^3} - \frac{1}{k^3}\right)I\right)(C_3) \\
= \frac{1}{(k+1)^2}B^{-1}\Sigma_g(\theta^*)B^{-1} + \mathcal{O}\left(\frac{1}{k^4}\right)
\end{aligned}$$

Because $1/(k+1)^3 - 1/k^3 = \mathcal{O}(1/k^4)$ we note that the term depending on C_3 can thus be absorbed into the $\Omega(1/k^4)$ term. Then using $1/(k+1) - 1/k = -1/(k(k+1))$ and $1/(k+1)^2 - 1/k^2 = -(2k+1)/(k^2(k+1))$, the above equation becomes

$$\begin{aligned}
\left(\frac{1}{k(k+1)}\Xi_{B^{-1}H^*} - \frac{1}{k(k+1)}I\right)(C_1) + \left(\frac{1}{k^2(k+1)}\Xi_{B^{-1}H^*} - \frac{2k+1}{k^2(k+1)^2}I\right)(C_2) \\
= \frac{1}{(k+1)^2}B^{-1}\Sigma_g(\theta^*)B^{-1} + \mathcal{O}\left(\frac{1}{k^4}\right)
\end{aligned}$$

Multiplying both sides of this by $k(k+1)$ yields

$$\begin{aligned}
&\left(\Xi_{B^{-1}H^*} - I\right)(C_1) + \left(\frac{1}{k}\Xi_{B^{-1}H^*} - \frac{2k+1}{k(k+1)}I\right)(C_2) \\
&\quad = \frac{k}{k+1}B^{-1}\Sigma_g(\theta^*)B^{-1} + \mathcal{O}\left(\frac{1}{k^2}\right) = B^{-1}\Sigma_g(\theta^*)B^{-1} - \frac{1}{k+1}B^{-1}\Sigma_g(\theta^*)B^{-1} + \mathcal{O}\left(\frac{1}{k^2}\right)
\end{aligned}$$

Noting that $1/k - 1/(k+1) = \mathcal{O}(1/k^2)$ and $(2k+1)/(k(k+1)) - 2/(k+1) = \mathcal{O}(1/k^2)$ we have that

$$\begin{aligned}
\left(\frac{1}{k}\Xi_{B^{-1}H^*} - \frac{2k+1}{k(k+1)}I\right)(C_2) &= \left(\frac{1}{k+1}\Xi_{B^{-1}H^*} - \frac{2}{k+1}I\right)(C_2) + \mathcal{O}\left(\frac{1}{k^2}\right) \\
&= \frac{1}{k+1}\left(\Xi_{B^{-1}H^*} - 2I\right)(C_2) + \mathcal{O}\left(\frac{1}{k^2}\right)
\end{aligned}$$

which combined with the previous equation yields

$$\left(\Xi_{B^{-1}H^*} - I\right)(C_1) + \frac{1}{k+1}\left(\Xi_{B^{-1}H^*} - 2I\right)(C_2) = B^{-1}\Sigma_g(\theta^*)B^{-1} - \frac{1}{k+1}B^{-1}\Sigma_g(\theta^*)B^{-1} + \mathcal{O}\left(\frac{1}{k^2}\right)$$

Comparing coefficients gives

$$\begin{aligned}\left(\Xi_{B^{-1}H^*} - I\right)(C_1) &= B^{-1}\Sigma_g(\theta^*)B^{-1} \\ \left(\Xi_{B^{-1}H^*} - 2I\right)(C_2) &= -B^{-1}\Sigma_g(\theta^*)B^{-1}\end{aligned}$$

or in other words

$$\begin{aligned}C_1 &= \left(\Xi_{B^{-1}H^*} - I\right)^{-1}\left(B^{-1}\Sigma_g(\theta^*)B^{-1}\right) \\ C_2 &= -\left(\Xi_{B^{-1}H^*} - 2I\right)^{-1}\left(B^{-1}\Sigma_g(\theta^*)B^{-1}\right)\end{aligned}$$

Thus we may conclude that

$$\begin{aligned}V_k &= \frac{1}{k}C_1 + \frac{1}{k^2}C_2 + \mathcal{O}\left(\frac{1}{k^3}\right) \\ &= \frac{1}{k}\left(\Xi_{B^{-1}H^*} - I\right)^{-1}\left(B^{-1}\Sigma_g(\theta^*)B^{-1}\right) - \frac{1}{k^2}\left(\Xi_{B^{-1}H^*} - 2I\right)^{-1}\left(B^{-1}\Sigma_g(\theta^*)B^{-1}\right) + \mathcal{O}\left(\frac{1}{k^3}\right)\end{aligned}$$

A.2 Bound derivations for Section 5.7.3

By Lemma 4

$$\operatorname{tr}\left(H^{*-1}\Sigma_g(\theta^*)\right) \geq \lambda_n\left(H^{*-1}\right)\operatorname{tr}(\Sigma_g(\theta^*)) = \frac{1}{\lambda_1\left(H^*\right)}\operatorname{tr}(\Sigma_g(\theta^*))$$

and

$$\operatorname{tr}\left(H^{*-1}\Sigma_g(\theta^*)\right) \leq \lambda_1\left(H^{*-1}\right)\operatorname{tr}(\Sigma_g(\theta^*)) = \frac{1}{\lambda_n\left(H^*\right)}\operatorname{tr}(\Sigma_g(\theta^*))$$

and so we have

$$\frac{1}{2k\lambda_1\left(H^*\right)}\operatorname{tr}(\Sigma_g(\theta^*)) \leq \frac{1}{2k}\operatorname{tr}\left(H^{*-1}\Sigma_g(\theta^*)\right) \leq \frac{1}{2k\lambda_n\left(H^*\right)}\operatorname{tr}(\Sigma_g(\theta^*))$$

Meanwhile, by Lemma 4

$$\begin{aligned}\operatorname{tr}\left(\left(I - \frac{\lambda_n(H^*)}{2}H^{*-1}\right)^{-1}\Sigma_g(\theta^*)\right) &\geq \lambda_n\left(\left(I - \frac{\lambda_n(H^*)}{2}H^{*-1}\right)^{-1}\right)\operatorname{tr}(\Sigma_g(\theta^*)) \\ &= \frac{1}{\lambda_1\left(I - \frac{\lambda_n(H^*)}{2}H^{*-1}\right)}\operatorname{tr}(\Sigma_g(\theta^*)) \\ &= \frac{1}{1 - \frac{\lambda_n(H^*)}{2}\lambda_n(H^{*-1})}\operatorname{tr}(\Sigma_g(\theta^*)) \\ &= \frac{1}{1 - \frac{1}{2\kappa(H^*)}}\operatorname{tr}(\Sigma_g(\theta^*)) \geq \operatorname{tr}(\Sigma_g(\theta^*))\end{aligned}$$

where $\kappa(H^*) = \lambda_1(H^*)/\lambda_n(H^*)$ is the condition number of H^*. Similarly, by Lemma 4 we have

$$\begin{aligned}
\operatorname{tr}\left(\left(I - \frac{\lambda_n(H^*)}{2}H^{*-1}\right)^{-1}\Sigma_g(\theta^*)\right) &\leq \lambda_1\left(\left(I - \frac{\lambda_n(H^*)}{2}H^{*-1}\right)^{-1}\right)\operatorname{tr}(\Sigma_g(\theta^*)) \\
&= \frac{1}{\lambda_n\left(I - \frac{\lambda_n(H^*)}{2}H^{*-1}\right)}\operatorname{tr}(\Sigma_g(\theta^*)) \\
&= \frac{1}{1 - \frac{\lambda_n(H^*)}{2}\lambda_1(H^{*-1})}\operatorname{tr}(\Sigma_g(\theta^*)) \\
&= \frac{1}{1 - \frac{\lambda_n(H^*)}{2\lambda_n(H^*)}}\operatorname{tr}(\Sigma_g(\theta^*)) = 2\operatorname{tr}(\Sigma_g(\theta^*))
\end{aligned}$$

and thus

$$\frac{1}{4k\lambda_n(H^*)}\operatorname{tr}(\Sigma_g(\theta^*)) \leq \frac{1}{4k\lambda_n(H^*)}\operatorname{tr}\left(\left(I - \frac{\lambda_n(H^*)}{2}H^{*-1}\right)^{-1}\Sigma_g(\theta^*)\right) \leq \frac{1}{2k\lambda_n(H^*)}\operatorname{tr}(\Sigma_g(\theta^*))$$

A.3 Proof of Theorem 8

To begin, we observe that analogously to eqn. 5.13 we have

$$\mathrm{E}[h(\bar{\theta}_k)] - h(\theta^*) = \frac{1}{2}\operatorname{tr}\left(H^*\bar{V}_k\right) + \frac{1}{2}\operatorname{tr}\left(H^*\left(\mathrm{E}[\bar{\theta}_k] - \theta^*\right)\left(\mathrm{E}[\bar{\theta}_k] - \theta^*\right)^\top\right) \tag{A.2}$$

where

$$\bar{V}_k = \operatorname{var}(\bar{\theta}_k) = \operatorname{cov}(\bar{\theta}_k, \bar{\theta}_k) = \mathrm{E}\left[\left(\bar{\theta}_k - \mathrm{E}[\bar{\theta}_k]\right)\left(\bar{\theta}_k - \mathrm{E}[\bar{\theta}_k]\right)^\top\right]$$

Our first major task is to find an expression for $\bar{V}_k$ in order to bound the term $\frac{1}{2}\operatorname{tr}\left(H^*\bar{V}_k\right)$. To this end we observe that

$$\bar{V}_k = \frac{1}{(k+1)^2}\sum_{i=0}^{k}\sum_{j=0}^{k}\operatorname{cov}(\theta_i, \theta_j)$$

For $j > i$ we have

$$\operatorname{cov}\left(\theta_i, \theta_j\right) = \operatorname{cov}\left(\theta_i, \theta_{j-1} - \alpha B^{-1}g_{j-1}(\theta_{j-1})\right) = \operatorname{cov}\left(\theta_i, \theta_{j-1}\right) - \alpha\operatorname{cov}\left(\theta_i, g_{j-1}(\theta_{j-1})\right)B^{-1}$$

where

$$\begin{aligned}
\operatorname{cov}\left(\theta_i, g_{j-1}(\theta_{j-1})\right) &= \mathrm{E}\left[(\theta_i - \mathrm{E}[\theta_i])(g_{j-1}(\theta_{j-1}) - \mathrm{E}[g_{j-1}(\theta_{j-1})])^\top\right] \\
&= \mathrm{E}_{\theta_i,\theta_{j-1}}\left[\mathrm{E}_{g_{j-1}(\theta_{j-1})|\theta_{j-1}}\left[(\theta_i - \mathrm{E}[\theta_i])(g_{j-1}(\theta_{j-1}) - \mathrm{E}[g_{j-1}(\theta_{j-1})])^\top\right]\right] \\
&= \mathrm{E}_{\theta_i,\theta_{j-1}}\left[(\theta_i - \mathrm{E}[\theta_i])(\nabla h(\theta_{j-1}) - \mathrm{E}[g_{j-1}(\theta_{j-1})])^\top\right] \\
&= \mathrm{E}\left[(\theta_i - \mathrm{E}[\theta_i])(\nabla h(\theta_{j-1}) - \mathrm{E}[g_{j-1}(\theta_{j-1})])^\top\right]
\end{aligned}$$

where we have used the fact that $g_{j-1}(\theta_{j-1})$ is conditionally independent of θ_i given θ_{j-1} for $j - 1 \geq i$ (which allows us to take the conditional expectation over $g_{j-1}(\theta_{j-1})$ inside), and is an unbiased estimator of $\nabla_h(\theta_{j-1})$.

Then noting that $\mathrm{E}[g_{j-1}(\theta_{j-1})] = \mathrm{E}[\nabla h(\theta_{j-1})] = \mathrm{E}[H^*(\theta_{j-1} - \theta^*)] = H^*(\mathrm{E}[\theta_{j-1}] - \theta^*)$ we have

$$\begin{aligned}\nabla h(\theta_{j-1}) - \mathrm{E}[g_{j-1}(\theta_{j-1})] &= H^*(\theta_{j-1} - \theta^*) - H^*(\mathrm{E}[\theta_{j-1}] - \theta^*) \\ &= H^*(\theta_{j-1} - \mathrm{E}[\theta_{j-1}])\end{aligned}$$

so that

$$\begin{aligned}\mathrm{cov}\left(\theta_i, g_{j-1}(\theta_{j-1})\right) &= \mathrm{E}\left[(\theta_i - \mathrm{E}[\theta_i])(\nabla h(\theta_{j-1}) - \mathrm{E}[g_{j-1}(\theta_{j-1})])^\top\right] \\ &= \mathrm{E}\left[(\theta_i - \mathrm{E}[\theta_i])(H^*(\theta_{j-1} - \mathrm{E}[\theta_{j-1}]))^\top\right] \\ &= \mathrm{E}\left[(\theta_i - \mathrm{E}[\theta_i])(\theta_{j-1} - \mathrm{E}[\theta_{j-1}])^\top\right] H^* = \mathrm{cov}(\theta_i, \theta_{j-1})H^*\end{aligned}$$

From this we conclude that

$$\begin{aligned}\mathrm{cov}\left(\theta_i, \theta_j\right) &= \mathrm{cov}\left(\theta_i, \theta_{j-1}\right) - \alpha\,\mathrm{cov}\left(\theta_i, g_{j-1}(\theta_{j-1})\right) B^{-1} \\ &= \mathrm{cov}\left(\theta_i, \theta_{j-1}\right) - \alpha\,\mathrm{cov}(\theta_i, \theta_{j-1})H^* B^{-1} \\ &= \mathrm{cov}\left(\theta_i, \theta_{j-1}\right)\left(I - \alpha B^{-1}H^*\right)^\top\end{aligned}$$

Applying this recursively we have that for $j \geq i$

$$\mathrm{cov}\left(\theta_i, \theta_j\right) = V_i\big(I - \alpha B^{-1}H^*\big)^{j-i^\top}$$

and taking transposes and switching the roles of i and j we similarly have for $i \geq j$

$$\mathrm{cov}\left(\theta_i, \theta_j\right) = \left(I - \alpha B^{-1}H^*\right)^{i-j} V_j$$

Thus we have the following expression for the variance $\bar{V}_k$ of the averaged parameter $\bar{\theta}_k$:

$$\begin{aligned}\bar{V}_k &= \frac{1}{(k+1)^2}\sum_{i=0}^{k}\sum_{j=0}^{k}\mathrm{cov}(\theta_i, \theta_j) \\ &= \sum_{i=0}^{k}\left(\sum_{j=0}^{i}\left(I - \alpha B^{-1}H^*\right)^{i-j} V_j + \sum_{j=i+1}^{k} V_i\big(I - \alpha B^{-1}H^*\big)^{j-i^\top}\right)\end{aligned}$$

which by reordering the sums and re-indexing can be written as

$$\bar{V}_k = \sum_{i=0}^{k}\left(\sum_{j=0}^{k-i}\left(I - \alpha B^{-1}H^*\right)^{j} V_i + \sum_{j=1}^{k-i} V_i\big(I - \alpha B^{-1}H^*\big)^{j^\top}\right)$$

Having computed $\bar{V}_k$ we now deal with the term $\frac{1}{2}\,\mathrm{tr}\left(H^*\bar{V}_k\right)$. Observing that

$$H^{*1/2}(I - \alpha B^{-1}H^*) = \left(I - \alpha H^{*1/2}B^{-1}H^{*1/2}\right)H^{*1/2} = (I - C)\,H^{*1/2}$$

where $C = \alpha {H^*}^{1/2} B^{-1} {H^*}^{1/2}$ (as it is defined in Subsection 5.7), we have

$$H^{*1/2}\bar{V}_k H^{*1/2} = \frac{1}{(k+1)^2}\sum_{i=0}^{k}\left(\sum_{j=0}^{k-i}(I-C)^j\left(H^{*1/2}V_iH^{*1/2}\right) + \sum_{j=1}^{k-i}\left(H^{*1/2}V_iH^{*1/2}\right)(I-C)^j\right)$$

It thus follows that

$$\frac{1}{2}\operatorname{tr}\left(H^*\bar{V}_k\right) = \frac{1}{2}\operatorname{tr}\left(H^{*1/2}\bar{V}_kH^{*1/2}\right) = \frac{1}{2(k+1)^2}\sum_{i=0}^{k}\operatorname{tr}\left(\left(I + 2\sum_{j=1}^{k-i}(I-C)^j\right)H^{*1/2}V_iH^{*1/2}\right)$$

Recall that from eqn. 5.14 we have

$$H^{*1/2}V_iH^{*1/2} = \left(I-(I-\Xi_C)^i\right)\left(H^{*1/2}V_\infty H^{*1/2}\right) + (I-\Xi_C)^i\left(H^{*1/2}(\theta_0-\theta^*)(\theta_0-\theta^*)^\top H^{*1/2}\right)$$

Plugging this into the previous equation and exploiting linearity gives

$$\begin{aligned}\frac{1}{2}\operatorname{tr}\left(H^*\bar{V}_k\right) = {}& \frac{1}{2(k+1)^2}\operatorname{tr}\left(\sum_{i=0}^{k}\left(I+2\sum_{j=1}^{k-i}(I-C)^j\right)\left(I-(I-\Xi_C)^i\right)\left(H^{*1/2}V_\infty H^{*1/2}\right)\right)\\ &+ \frac{1}{2(k+1)^2}\operatorname{tr}\left(\sum_{i=0}^{k}\left(I+2\sum_{j=1}^{k-i}(I-C)^j\right)(I-\Xi_C)^i\left(H^{*1/2}(\theta_0-\theta^*)(\theta_0-\theta^*)^\top H^{*1/2}\right)\right)\end{aligned}$$

Then applying eqn. 5.16 to this we obtain

$$\begin{aligned}\frac{1}{2}\operatorname{tr}\left(H^*\bar{V}_k\right) = {}& \frac{1}{2(k+1)^2}\operatorname{tr}\left(\sum_{i=0}^{k}\left(I+2\sum_{j=1}^{k-i}(I-C)^j\right)\left(I-(I-2C)^i\right)H^{*1/2}V_\infty H^{*1/2}\right)\\ &+ \frac{1}{2(k+1)^2}\operatorname{tr}\left(\sum_{i=0}^{k}\left(I+2\sum_{j=1}^{k-i}(I-C)^j\right)(I-2C)^i\ H^{*1/2}(\theta_0-\theta^*)(\theta_0-\theta^*)^\top H^{*1/2}\right)\end{aligned} \tag{A.3}$$

Because C and $I-2C$ are PSD (which follows from the hypothesis $2\alpha\lambda_1(B^{-1}H^*) < 1$ as shown in the previous section), as are $I-C$, $I-(I-2C) = 2C$ and $I-(I-C) = C$ by consequence, we have the following

basic matrix inequalities

$$I + 2\sum_{j=1}^{k-i}(I-C)^j \leq 2\sum_{j=0}^{\infty}(I-C)^j = 2C^{-1} \tag{A.4}$$

$$I + 2\sum_{j=1}^{k-i}(I-C)^j \leq 2(k+1)I \tag{A.5}$$

$$\sum_{i=0}^{k}(I-2C)^i \leq \sum_{i=0}^{\infty}(I-2C)^i = \frac{1}{2}C^{-1} \tag{A.6}$$

$$\sum_{i=0}^{k}(I-2C)^i \leq (k+1)I \tag{A.7}$$

$$\sum_{i=0}^{k}\left(I-(I-2C)^i\right) \leq (k+1)I \tag{A.8}$$

where $X \leq Y$ means that $Y - X$ is PSD.

To exploit these inequalities we will make use of the following lemma

Lemma 14. *If A, S, T, and X are matrices such that A, S and T commute with each other, $S \leq T$, and A and X are PSD, then we have*

$$\operatorname{tr}(ASX) \leq \operatorname{tr}(ATX)$$

Proof. Since A, S and T are commuting PSD matrices they have the same eigenvectors, as does $A^{1/2}$ (which thus also commutes).

Meanwhile, $S \leq T$ means that $T - S$ is PSD, and thus so is $A^{1/2}(T-S)A^{1/2}$. Because the trace of the product of two PSD matrices is non-negative (e.g. by Lemma 4), it follows that $\operatorname{tr}((A^{1/2}(T-S)A^{1/2})X) \geq 0$. Adding $\operatorname{tr}(A^{1/2}SA^{1/2}X)$ to both sides of this we get $\operatorname{tr}(A^{1/2}TA^{1/2}X) \geq \operatorname{tr}(A^{1/2}SA^{1/2}X)$. Because $A^{1/2}$ commutes with T and S we have $\operatorname{tr}(A^{1/2}TA^{1/2}X) = \operatorname{tr}(ATX)$ and $\operatorname{tr}(A^{1/2}SA^{1/2}X) = \operatorname{tr}(ASX)$, and so the result follows. □

As the right and left side of all the previously stated matrix inequalities are commuting matrices (because they all share their eigenvectors with C), we can apply this lemma to eqn. A.3 to obtain various simplifying upper bounds on $\frac{1}{2}\operatorname{tr}\left(H^*\bar{V}_k\right)$.

For the first term on the RHS of eqn. A.3 we can apply Lemma 14 using eqn. A.4 and then eqn. A.8, which gives an upper bound on this term of

$$\frac{1}{2(k+1)^2}\operatorname{tr}\left(2C^{-1}\,(k+1)I\,H^{*1/2}V_\infty H^{*1/2}\right) = \frac{1}{k+1}\operatorname{tr}\left(C^{-1}\,H^{*1/2}V_\infty H^{*1/2}\right) = \frac{1}{(k+1)\alpha}\operatorname{tr}\left(BV_\infty\right)$$

Or we can apply the lemma using eqn. A.5 and then eqn. A.8, which gives a different upper bound of

$$\frac{1}{2(k+1)^2}\operatorname{tr}\left(2(k+1)I\,(k+1)I\,H^{*1/2}V_\infty H^{*1/2}\right) = \operatorname{tr}\left(H^*V_\infty\right)$$

For the second term we can apply Lemma 14 using eqn. A.4 and then eqn. A.6, which gives an upper bound

on this term of

$$\frac{1}{2(k+1)^2}\operatorname{tr}\left(2C^{-1}\,\frac{1}{2}C^{-1}\,H^{*1/2}(\theta_0-\theta^*)(\theta_0-\theta^*)^\top H^{*1/2}\right) \quad = \frac{1}{2(k+1)^2\alpha^2}\left\|H^{*-1/2}B(\theta_0-\theta^*)\right\|^2$$

Or we can apply the lemma using eqn. A.5 and then eqn. A.6, which gives a different upper bound of

$$\frac{1}{2(k+1)^2}\operatorname{tr}\left(2(k+1)I\,\frac{1}{2}C^{-1}\,H^{*1/2}(\theta_0-\theta^*)(\theta_0-\theta^*)^\top H^{*1/2}\right) \quad = \frac{1}{2(k+1)\alpha}\left\|B^{1/2}(\theta_0-\theta^*)\right\|^2$$

Or finally, we can apply the lemma using eqn. A.5 and then eqn. A.7, which gives a third upper bound of

$$\frac{1}{2(k+1)^2}\operatorname{tr}\left(2(k+1)I\,(k+1)I\,H^{*1/2}(\theta_0-\theta^*)(\theta_0-\theta^*)^\top H^{*1/2}\right) = 2h(\theta_0)$$

Applying these bounds to eqn. A.3 yields

$$\begin{aligned}\frac{1}{2}\operatorname{tr}\left(H^*\bar{V}_k\right) \le \min&\left\{\frac{1}{(k+1)\alpha}\operatorname{tr}\left(BV_\infty\right),\ \operatorname{tr}\left(H^*V_\infty\right)\right\}\\ &+\min\left\{\frac{1}{2(k+1)^2\alpha^2}\left\|H^{*-1/2}B(\theta_0-\theta^*)\right\|^2,\ \frac{1}{2(k+1)\alpha}\left\|B^{1/2}(\theta_0-\theta^*)\right\|^2,\ 2h(\theta_0)\right\}\end{aligned} \tag{A.9}$$

To compute $\operatorname{tr}\left(BV_\infty\right)$, we have from eqn. 5.23 that

$$B^{-1}H^*V_\infty + V_\infty H^*B^{-1} = \alpha B^{-1}\Sigma_g(\theta^*)B^{-1}$$

Left multiplying both sides by $H^{*-1}B$ and right multiplying both sides by BH^{*-1} gives

$$V_\infty BH^{*-1} + H^{*-1}BV_\infty = \alpha H^{*-1}\Sigma_g(\theta^*)H^{*-1}$$

which after simple rearrangement can be written as a pseudo-CALE $AP + P^\top A + Q = 0$ (i.e. the form of eqn. 5.29), where

$$\begin{aligned}A &= -H^{*-1}\\ P &= BV_\infty\\ Q &= \alpha H^{*-1}\Sigma_g(\theta^*)H^{*-1}\end{aligned} \tag{A.10}$$

Thus applying Lemma 5 we get that

$$\operatorname{tr}(BV_\infty) = \operatorname{tr}(P) = -\frac{1}{2}\operatorname{tr}(A^{-1}Q) = -\frac{1}{2}\operatorname{tr}\left((-H^{*-1})^{-1}\alpha H^{*-1}\Sigma_g(\theta^*)H^{*-1}\right) = \alpha\operatorname{tr}\left(H^{*-1}\Sigma_g(\theta^*)\right) \tag{A.11}$$

It remains to bound the term $\frac{1}{2}\operatorname{tr}\left(H^*(\mathrm{E}[\bar{\theta}_k]-\theta^*)(\mathrm{E}[\bar{\theta}_k]-\theta^*)^\top\right)$. First we observe that by Theorem 1 we

have

$$\mathrm{E}[\bar{\theta}_k] - \theta^* = \frac{1}{k+1}\sum_{i=0}^{k}(\mathrm{E}[\theta_i] - \theta^*) = \frac{1}{k+1}\sum_{i=0}^{k}\left(I - \alpha B^{-1}H^*\right)^i(\theta_0 - \theta^*)$$

Then applying eqn. 5.19 gives

$$H^{*1/2}\left(\mathrm{E}[\bar{\theta}_k] - \theta^*\right) = \frac{1}{k+1}\sum_{i=0}^{k}(I-C)^i H^{*1/2}(\theta_0 - \theta^*)$$

And thus we have

$$\begin{aligned}\frac{1}{2}\operatorname{tr}\left(H^*\left(\mathrm{E}[\bar{\theta}_k] - \theta^*\right)\left(\mathrm{E}[\bar{\theta}_k] - \theta^*\right)^\top\right) &= \frac{1}{2}\operatorname{tr}\left(H^{*1/2}\left(\mathrm{E}[\bar{\theta}_k] - \theta^*\right)\left(\mathrm{E}[\bar{\theta}_k] - \theta^*\right)^\top H^{*1/2}\right)\\ &= \frac{1}{2(k+1)^2}\operatorname{tr}\left(\left(\sum_{i=0}^{k}(I-C)^i\right)H^{*1/2}(\theta_0-\theta^*)(\theta_0-\theta^*)^\top H^{*1/2}\left(\sum_{i=0}^{k}(I-C)^i\right)\right) \qquad \text{(A.12)}\end{aligned}$$

Similarly to eqn. A.4–A.8 we have the following matrix inequalities

$$\sum_{i=0}^{k}(I-C)^i \le \sum_{i=0}^{\infty}(I-C)^i = C^{-1} \qquad \text{(A.13)}$$

$$\sum_{i=0}^{k}(I-C)^i \le (k+1)I \qquad \text{(A.14)}$$

Applying Lemma 14 using eqn. A.13 twice we obtain an upper bound on the RHS of eqn. A.12 of

$$\frac{1}{2(k+1)^2}\operatorname{tr}\left(C^{-1}\,H^{*1/2}(\theta_0-\theta^*)(\theta_0-\theta^*)^\top H^{*1/2}\,C^{-1}\right) = \frac{1}{2(k+1)^2\alpha^2}\left\|H^{*-1/2}B(\theta_0-\theta^*)\right\|^2$$

Applying the lemma using eqn. A.13 and eqn. A.14 gives a different upper bound of

$$\frac{1}{2(k+1)^2}\operatorname{tr}\left(C^{-1}\,H^{*1/2}(\theta_0-\theta^*)(\theta_0-\theta^*)^\top H^{*1/2}\,(k+1)I\right) \qquad = \frac{1}{2(k+1)\alpha}\left\|B^{1/2}(\theta_0-\theta^*)\right\|^2$$

And finally, applying the lemma using eqn. A.14 twice gives an upper bound of

$$\frac{1}{2(k+1)^2}\operatorname{tr}\left((k+1)I\,H^{*1/2}(\theta_0-\theta^*)(\theta_0-\theta^*)^\top H^{*1/2}\,(k+1)I\right) = h(\theta_0)$$

Combining these various upper bounds gives us

$$\begin{aligned}&\frac{1}{2}\operatorname{tr}\left(H^*(\mathrm{E}[\bar{\theta}_k]-\theta^*)(\mathrm{E}[\bar{\theta}_k]-\theta^*)^\top\right)\\ &\quad\le \min\left\{\frac{1}{2(k+1)^2\alpha^2}\left\|H^{*-1/2}B(\theta_0-\theta^*)\right\|^2,\ \frac{1}{2(k+1)\alpha}\left\|B^{1/2}(\theta_0-\theta^*)\right\|^2,\ h(\theta_0)\right\} \qquad \text{(A.15)}\end{aligned}$$

Theorem 8 now follows from eqn. A.2, eqn. A.9, eqn. A.15, eqn. A.11 and eqn. 5.31.

Appendix B

Supplemental appendices for Chapter 6

B.1 Derivation of the expression for the approximation from Section 6.3.1

In this section we will show that

$$\begin{aligned}\mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\, g^{(1)}g^{(2)}\right] - \mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\right]\mathrm{E}\left[g^{(1)}g^{(2)}\right] \\ = \kappa(\bar{a}^{(1)}, \bar{a}^{(2)}, g^{(1)}, g^{(2)}) + \kappa(\bar{a}^{(1)})\kappa(\bar{a}^{(2)}, g^{(1)}, g^{(2)}) + \kappa(\bar{a}^{(2)})\kappa(\bar{a}^{(1)}, g^{(1)}, g^{(2)})\end{aligned}$$

The only specific property of the distribution over $\bar{a}^{(1)}$, $\bar{a}^{(2)}$, $g^{(1)}$, and $g^{(2)}$ which we will require to do this is captured by the following lemma.

Lemma 15. *Suppose u is a scalar variable which is independent of y when conditioned on the network's output $f(x, \theta)$, and v is some intermediate quantity computed during the evaluation of $f(x, \theta)$ (such as the activities of the units in some layer). Then we have*

$$\mathrm{E}\left[u\, \mathcal{D}v\right] = 0$$

Our proof of this lemma (which is at the end of this section) makes use of the fact that the expectations are taken with respect to the network's predictive distribution $P_{y|x}$ as opposed to the training distribution $\hat{Q}_{y|x}$.

Intuitively, this lemma says that the intermediate quantities computed in the forward pass of Algorithm 1 (or various functions of these) are statistically uncorrelated with various derivative quantities computed in the backwards pass, provided that the targets y are sampled according to the network's predictive distribution $P_{y|x}$ (instead of coming from the training set). Valid choices for u include $\bar{a}^{(k)}$, $\bar{a}^{(k)} - \mathrm{E}\left[\bar{a}^{(k)}\right]$ for $k \in \{1, 2\}$, and products of these. Examples of invalid choices for u include expressions involving $g^{(k)}$, since these will depend on the derivative of the loss, which is not independent of y given $f(x, \theta)$.

According to a well-known general formula relating moments to cumulants we may write $\mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\, g^{(1)}g^{(2)}\right]$ as a sum of 15 terms, each of which is a product of various cumulants corresponding to one of the 15 possible ways to partition the elements of $\{\bar{a}^{(1)}, \bar{a}^{(2)}, g^{(1)}, g^{(2)}\}$ into non-overlapping sets. For example, the term corresponding to the partition $\{\{\bar{a}^{(1)}\}, \{\bar{a}^{(2)}, g^{(1)}, g^{(2)}\}\}$ is $\kappa(\bar{a}^{(1)})\kappa(\bar{a}^{(2)}, g^{(1)}, g^{(2)})$.

Observing that 1st-order cumulants correspond to means and 2nd-order cumulants correspond to covariances,

for $k \in \{1,2\}$ Lemma 15 gives

$$\kappa(g^{(k)}) = \mathrm{E}\left[g^{(k)}\right] = \mathrm{E}\left[\mathcal{D}x^{(k)}\right] = 0$$

where $x^{(1)} = [x_i]_{k_2}$, and $x^{(2)} = [x_j]_{k_4}$ (so that $g^{(k)} = \mathcal{D}x^{(k)}$). And similarly for $k, m \in \{1,2\}$ it gives

$$\kappa(\bar{a}^{(k)}, g^{(m)}) = \mathrm{E}\left[\left(\bar{a}^{(m)} - \mathrm{E}\left[\bar{a}^{(m)}\right]\right)\left(g^{(k)} - \mathrm{E}\left[g^{(k)}\right]\right)\right] = \mathrm{E}\left[\left(\bar{a}^{(m)} - \mathrm{E}\left[\bar{a}^{(m)}\right]\right) g^{(k)}\right] = 0$$

Using these identities we can eliminate 10 of the terms.

The remaining expression for $\mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\, g^{(1)}g^{(2)}\right]$ is thus

$$\begin{aligned}\kappa(\bar{a}^{(1)}, \bar{a}^{(2)}, g^{(1)}, g^{(2)}) + \kappa(\bar{a}^{(1)})\kappa(\bar{a}^{(2)}, g^{(1)}, g^{(2)}) + \kappa(\bar{a}^{(2)})\kappa(\bar{a}^{(1)}, g^{(1)}, g^{(2)}) \\ + \kappa(\bar{a}^{(1)}, \bar{a}^{(2)})\kappa(g^{(1)}, g^{(2)}) + \kappa(\bar{a}^{(1)})\kappa(\bar{a}^{(2)})\kappa(g^{(1)}, g^{(2)})\end{aligned}$$

Noting that

$$\begin{aligned}&\kappa(\bar{a}^{(1)}, \bar{a}^{(2)})\kappa(g^{(1)}, g^{(2)}) + \kappa(\bar{a}^{(1)})\kappa(\bar{a}^{(2)})\kappa(g^{(1)}, g^{(2)}) \\ &\qquad = \mathrm{cov}(\bar{a}^{(1)}, \bar{a}^{(2)})\,\mathrm{E}\left[g^{(1)}g^{(2)}\right] + \mathrm{E}\left[\bar{a}^{(1)}\right]\mathrm{E}\left[\bar{a}^{(2)}\right]\mathrm{E}\left[g^{(1)}g^{(2)}\right] = \mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\right]\mathrm{E}\left[g^{(1)}g^{(2)}\right]\end{aligned}$$

it thus follows that

$$\begin{aligned}&\mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\, g^{(1)}g^{(2)}\right] - \mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\right]\mathrm{E}\left[g^{(1)}g^{(2)}\right] \\ &\qquad = \kappa(\bar{a}^{(1)}, \bar{a}^{(2)}, g^{(1)}, g^{(2)}) + \kappa(\bar{a}^{(1)})\kappa(\bar{a}^{(2)}, g^{(1)}, g^{(2)}) + \kappa(\bar{a}^{(2)})\kappa(\bar{a}^{(1)}, g^{(1)}, g^{(2)})\end{aligned}$$

as required.

It remains to prove Lemma 15.

Proof of Lemma 15. The chain rule gives

$$\mathcal{D}v = -\frac{\mathrm{d}\log p(y|x,\theta)}{\mathrm{d}v} = -\frac{\mathrm{d}\log r(y|z)}{\mathrm{d}z}\bigg|_{z=f(x,\theta)}^{\top} \frac{\mathrm{d}f(x,\theta)}{\mathrm{d}v}$$

From which it follows that

$$\begin{aligned}\mathrm{E}\left[u\,\mathcal{D}v\right] &= \mathrm{E}_{\hat{Q}_x}\left[\mathrm{E}_{P_{y|x}}\left[u\,\mathcal{D}v\right]\right] = \mathrm{E}_{\hat{Q}_x}\left[\mathrm{E}_{R_{y|f(x,\theta)}}\left[u\,\mathcal{D}v\right]\right] \\ &= \mathrm{E}_{\hat{Q}_x}\left[\mathrm{E}_{R_{y|f(x,\theta)}}\left[-u\,\frac{\mathrm{d}\log r(y|z)}{\mathrm{d}z}\bigg|_{z=f(x,\theta)}^{\top} \frac{\mathrm{d}f(x,\theta)}{\mathrm{d}v}\right]\right] \\ &= \mathrm{E}_{\hat{Q}_x}\left[-u\,\mathrm{E}_{R_{y|f(x,\theta)}}\left[\frac{\mathrm{d}\log r(y|z)}{\mathrm{d}z}\bigg|_{z=f(x,\theta)}\right]^{\top} \frac{\mathrm{d}f(x,\theta)}{\mathrm{d}v}\right] = \mathrm{E}_{\hat{Q}_x}\left[-u\,\vec{0}^{\top}\,\frac{\mathrm{d}f(x,\theta)}{\mathrm{d}v}\right] = 0\end{aligned}$$

That the inner expectation above is $\vec{0}$ follows from the fact that the expected score of a distribution, when taken with respect to that distribution, is $\vec{0}$.

□

B.2 Efficient techniques for inverting $A \otimes B \pm C \otimes D$

It is well known that $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$, and that matrix-vector products with this matrix can thus be computed as $(A^{-1} \otimes B^{-1})v = \mathrm{vec}(B^{-1}VA^{-\top})$, where V is the matrix representation of v (so that $v = \mathrm{vec}(V)$).

Somewhat less well known is that there are also formulas for $(A \otimes B \pm C \otimes D)^{-1}$ which can be efficiently computed and likewise give rise to efficient methods for computing matrix-vector products.

First, note that $(A \otimes B \pm C \otimes D)^{-1}v = u$ is equivalent to $(A \otimes B \pm C \otimes D)u = v$, which is equivalent to the linear matrix equation $BUA^{\top} \pm DUC^{\top} = V$, where $u = \mathrm{vec}(U)$ and $v = \mathrm{vec}(V)$. This is known as a generalized Stein equation, and different examples of it have been studied in the control theory literature, where they have numerous applications. For a recent survey of this topic, see Simoncini (2014).

One well-known class of methods called Smith-type iterations (Smith, 1968) involve rewriting this matrix equation as a fixed point iteration and then carrying out this iteration to convergence. Interestingly, through the use of a special squaring trick, one can simulate 2^j of these iterations with only $\mathcal{O}(j)$ matrix-matrix multiplications.

Another class of methods for solving Stein equations involves the use of matrix decompositions (e.g. Chu, 1987; Gardiner et al., 1992). Here we will present such a method particularly well suited for our application, as it produces a formula for $(A \otimes B + C \otimes D)^{-1}v$, which after a fixed overhead cost (involving the computation of SVDs and matrix square roots), can be repeatedly evaluated for different choices of v using only a few matrix-matrix multiplications.

We will assume that A, B, C, and D are symmetric positive semi-definite, as they always are in our applications. We have

$$A \otimes B \pm C \otimes D = (A^{1/2} \otimes B^{1/2})(I \otimes I \pm A^{-1/2}CA^{-1/2} \otimes B^{-1/2}DB^{-1/2})(A^{1/2} \otimes B^{1/2})$$

Inverting both sides of the above equation gives

$$(A \otimes B \pm C \otimes D)^{-1} = (A^{-1/2} \otimes B^{-1/2})(I \otimes I \pm A^{-1/2}CA^{-1/2} \otimes B^{-1/2}DB^{-1/2})^{-1}(A^{-1/2} \otimes B^{-1/2})$$

Using the symmetric eigen/SVD-decomposition, we can write $A^{-1/2}CA^{-1/2} = E_1S_1E_1^{\top}$ and $B^{-1/2}DB^{-1/2} = E_2S_2E_2^{\top}$, where for $i \in \{1, 2\}$ the S_i are diagonal matrices and the E_i are unitary matrices.

This gives

$$\begin{aligned} I \otimes I \pm A^{-1/2}CA^{-1/2} \otimes B^{-1/2}DB^{-1/2} &= I \otimes I \pm E_1S_1E_1^{\top} \otimes E_2S_2E_2^{\top} \\ &= E_1E_1^{\top} \otimes E_2E_2^{\top} \pm E_1S_1E_1^{\top} \otimes E_2S_2E_2^{\top} \\ &= (E_1 \otimes E_2)(I \otimes I \pm S_1 \otimes S_2)(E_1^{\top} \otimes E_2^{\top}) \end{aligned}$$

so that

$$(I \otimes I \pm A^{-1/2}CA^{-1/2} \otimes B^{-1/2}DB^{-1/2})^{-1} = (E_1 \otimes E_2)(I \otimes I \pm S_1 \otimes S_2)^{-1}(E_1^{\top} \otimes E_2^{\top})$$

Note that both $I \otimes I$ and $S_1 \otimes S_2$ are diagonal matrices, and thus the middle matrix $(I \otimes I \pm S_1 \otimes S_2)^{-1}$ is just the inverse of a diagonal matrix, and so can be computed efficiently.

Thus we have

$$\begin{aligned}(A \otimes B \pm C \otimes D)^{-1} &= (A^{-1/2} \otimes B^{-1/2})(E_1 \otimes E_2)(I \otimes I \pm S_1 \otimes S_2)^{-1}(E_1^\top \otimes E_2^\top)(A^{-1/2} \otimes B^{-1/2}) \\ &= (K_1 \otimes K_2)(I \otimes I \pm S_1 \otimes S_2)^{-1}(K_1^\top \otimes K_2^\top)\end{aligned}$$

where $K_1 = A^{-1/2}E_1$ and $K_2 = B^{-1/2}E_2$.

And so matrix-vector products with $(A \otimes B \pm C \otimes D)^{-1}$ can be computed as

$$(A \otimes B \pm C \otimes D)^{-1}v = \operatorname{vec}\left(K_2\left[(K_2^\top V K_1) \oslash \left(\mathbf{1}\mathbf{1}^\top \pm s_2 s_1^\top\right)\right]K_1^\top\right)$$

where $E \oslash F$ denotes element-wise division of E by F, $s_i = \operatorname{diag}(S_i)$, and $\mathbf{1}$ is the vector of ones (sized as appropriate). Note that if we wish to compute multiple matrix-vector products with $(A \otimes B \pm C \otimes D)^{-1}$ (as we will in our application), the quantities K_1, K_2, s_1 and s_2 only need to be computed the first time, thus reducing the cost of any future such matrix-vector products, and in particular avoiding any additional SVD computations.

In the considerably simpler case where A and B are both scalar multiples of the identity, and ξ is the product of these multiples, we have

$$(\xi I \otimes I \pm C \otimes D)^{-1} = (E_1 \otimes E_2)(\xi I \otimes I \pm S_1 \otimes S_2)^{-1}(E_1^\top \otimes E_2^\top)$$

where $E_1 S_1 E_1^\top$ and $E_2 S_2 E_2^\top$ are the symmetric eigen/SVD-decompositions of C and D, respectively. And so matrix-vector products with $(\xi I \otimes I \pm C \otimes D)^{-1}$ can be computed as

$$(\xi I \otimes I \pm C \otimes D)^{-1}v = \operatorname{vec}\left(E_2\left[(E_2^\top V E_1) \oslash \left(\xi\mathbf{1}\mathbf{1}^\top \pm s_2 s_1^\top\right)\right]E_1^\top\right)$$

B.3 Computing $v^\top F v$ and $u^\top F v$ more efficiently

Note that the Fisher is given by

$$F = \mathrm{E}_{\hat{Q}_x}\left[J^\top F_R J\right]$$

where J is the Jacobian of $f(x, \theta)$ and F_R is the Fisher information matrix of the network's predictive distribution $R_{y|z}$, evaluated at $z = f(x, \theta)$ (where we treat z as the "parameters").

To compute the matrix-vector product Fv as estimated from a mini-batch we simply compute $J^\top F_R J v$ for each x in the mini-batch, and average the results. This latter operation can be computed in 3 stages, which correspond to multiplication of the vector v first by J, then by F_R, and then by $J^\top$.

Multiplication by J can be performed by a forward pass which is like a linearized version of the standard forward pass of Algorithm 1. As F_R is usually diagonal or diagonal plus rank-1, matrix-vector multiplications with it are cheap and easy. Finally, multiplication by $J^\top$ can be performed by a backwards pass which is essentially the same as that of Algorithm 1. See Sections 4.4.1 and 5.4 for further details.

The naive way of computing $v^\top F v$ is to compute Fv as above, and then compute the inner product of Fv with v. Additionally computing $u^\top F v$ and $u^\top F u$ would require another such matrix-vector product Fu.

However, if we instead just compute the matrix-vector products Jv (which requires only half the work of computing Fv), then computing $v^\top F v$ as $(Jv)^\top F_R (Jv)$ is essentially free. And with Ju computed, we can similarly obtain $u^\top F v$ as $(Ju)^\top F_R (Jv)$ and $u^\top F u$ as $(Ju)^\top F_R (Ju)$.

This trick thus reduces the computational cost associated with computing these various scalars by roughly half.

B.4 Proofs for Section 6.10

Proof of Theorem 11. First we will show that the given network transformation can be viewed as reparameterization of the network according to an invertible linear function ζ.

Define $\theta^\dagger = [\mathrm{vec}(W_1^\dagger)^\top \mathrm{vec}(W_2^\dagger)^\top \dots \mathrm{vec}(W_\ell^\dagger)^\top]^\top$, where $W_i^\dagger = \Phi_i^{-1} W_i \Omega_{i-1}^{-1}$ (so that $W_i = \Phi_i W_i^\dagger \Omega_{i-1}$) and let ζ be the function which maps $\theta^\dagger$ to θ. Clearly ζ is an invertible linear transformation.

If the transformed network uses $\theta^\dagger$ in place of θ then we have

$$\bar{a}_i^\dagger = \Omega_i \bar{a}_i \qquad \text{and} \qquad s_i^\dagger = \Phi_i^{-1} s_i$$

which we can prove by a simple induction. First note that $\bar{a}_0^\dagger = \Omega_0 \bar{a}_0$ by definition. Then, assuming by induction that $\bar{a}_{i-1}^\dagger = \Omega_{i-1} \bar{a}_{i-1}$, we have

$$s_i^\dagger = W_i^\dagger \bar{a}_{i-1}^\dagger = \Phi_i^{-1} W_i \Omega_{i-1}^{-1} \Omega_{i-1} \bar{a}_{i-1} = \Phi_i^{-1} W_i \bar{a}_{i-1} = \Phi_i^{-1} s_i$$

and therefore also

$$\bar{a}_i^\dagger = \Omega_i \bar{\phi}_i(\Phi_i s_i^\dagger) = \Omega_i \bar{\phi}_i(\Phi_i \Phi_i^{-1} s_i) = \Omega_i \bar{\phi}_i(s_i) = \Omega_i \bar{a}_i$$

And because $\Omega_\ell = I$, we have $\bar{a}_\ell^\dagger = \bar{a}_\ell$, or more simply that $a_\ell^\dagger = a_\ell$, and thus both the original network and the transformed one have the same output (i.e. $f(x, \theta) = f^\dagger(x, \theta^\dagger)$). From this it follows that $f^\dagger(x, \theta^\dagger) = f(x, \theta) = f(x, \zeta(\theta^\dagger))$, and thus the transformed network can be viewed as a reparameterization of the original network by $\theta^\dagger$. Similarly we have that $h^\dagger(\theta^\dagger) = h(\theta) = h(\zeta(\theta^\dagger))$.

The following lemma is adapted from Corollary 10.

Lemma 16. *Let ζ be some invertible affine function mapping $\theta^\dagger$ to θ, which reparameterizes the objective $h(\theta)$ as $h(\zeta(\theta^\dagger))$. Suppose that B and $B^\dagger$ are invertible matrices satisfying*

$$J_\zeta^\top B J_\zeta = B^\dagger$$

Then, additively updating θ by $\delta = -\alpha B^{-1} \nabla h$ is equivalent to additively updating $\theta^\dagger$ by $\delta^\dagger = -\alpha B^{\dagger -1} \nabla_{\theta^\dagger} h(\zeta(\theta^\dagger))$, in the sense that $\zeta(\theta^\dagger + \delta^\dagger) = \theta + \delta$.

Because $h^\dagger(\theta^\dagger) = h(\theta) = h(\zeta(\theta^\dagger))$ we have that $\nabla h^\dagger = \nabla_{\theta^\dagger} h(\zeta(\theta^\dagger))$. So, by the above lemma, to prove the theorem it suffices to show that $J_\zeta^\top \breve{F} J_\zeta = \breve{F}^\dagger$ and $J_\zeta^\top \tilde{F} J_\zeta = \tilde{F}^\dagger$.

Using $W_i = \Phi_i W_i^\dagger \Omega_{i-1}$ it is straightforward to verify that

$$J_\zeta = \mathrm{diag}(\Omega_0^\top \otimes \Phi_1, \Omega_1^\top \otimes \Phi_2, \, \dots, \, \Omega_{\ell-1}^\top \otimes \Phi_\ell)$$

Because $s_i = \Phi_i s_i^\dagger$ and the fact that the networks compute the same outputs (so the loss derivatives are identical), we have by the chain rule that, $g_i^\dagger = \mathcal{D} s_i^\dagger = \Phi_i^\top \mathcal{D} s_i = \Phi_i^\top g_i$, and therefore

$$G_{i,j}^\dagger = \mathrm{E}\left[g_i^\dagger g_j^{\dagger\top}\right] = \mathrm{E}\left[\Phi_i^\top g_i (\Phi_i^\top g_i)^\top\right] = \Phi_i^\top \mathrm{E}\left[g_i g_i^\top\right] \Phi_j = \Phi_i^\top G_{i,j} \Phi_j$$

Furthermore,

$$\bar{A}^{\dagger}_{i,j} = \mathrm{E}\left[\bar{a}^{\dagger}_i \bar{a}^{\dagger\top}_j\right] = \mathrm{E}\left[(\Omega_i \bar{a}_i)(\Omega_j \bar{a}_j)^\top\right] = \Omega_i \,\mathrm{E}\left[\bar{a}_i \bar{a}_j^\top\right] \Omega_j^\top = \Omega_i \bar{A}_{i,j} \Omega_j^\top$$

Using these results we may express the Kronecker-factored blocks of the approximate Fisher $\tilde{F}^{\dagger}$, as computed using the transformed network, as follows:

$$\begin{aligned}\tilde{F}^{\dagger}_{i,j} = \bar{A}^{\dagger}_{i-1,j-1} \otimes G^{\dagger}_{i,j} = \Omega_{i-1}\bar{A}_{i-1,j-1}\Omega_{j-1}^\top \otimes \Phi_i^\top G_{i,j}\Phi_j &= (\Omega_{i-1} \otimes \Phi_i^\top)(\bar{A}_{i-1,j-1} \otimes G_{i,j})(\Omega_{j-1}^\top \otimes \Phi_j) \\ &= (\Omega_{i-1} \otimes \Phi_i^\top)\tilde{F}_{i,j}(\Omega_{j-1}^\top \otimes \Phi_j)\end{aligned}$$

Given this identity we thus have

$$\begin{aligned}\breve{F}^{\dagger} &= \mathrm{diag}\left(\tilde{F}^{\dagger}_{1,1}, \tilde{F}^{\dagger}_{2,2}, \ldots, \tilde{F}^{\dagger}_{\ell,\ell}\right) \\ &= \mathrm{diag}\left((\Omega_0 \otimes \Phi_1^\top)\tilde{F}_{1,1}(\Omega_0^\top \otimes \Phi_1), (\Omega_1 \otimes \Phi_2^\top)\tilde{F}_{2,2}(\Omega_1^\top \otimes \Phi_2), \ldots, (\Omega_{\ell-1} \otimes \Phi_\ell^\top)\tilde{F}_{\ell,\ell}(\Omega_{\ell-1}^\top \otimes \Phi_\ell)\right) \\ &= \mathrm{diag}(\Omega_0 \otimes \Phi_1^\top, \Omega_1 \otimes \Phi_2^\top, \ldots, \Omega_{\ell-1} \otimes \Phi_\ell^\top)\,\mathrm{diag}\left(\tilde{F}_{1,1}, \tilde{F}_{2,2}, \ldots, , \tilde{F}_{\ell,\ell}\right) \\ &\qquad \cdot \mathrm{diag}(\Omega_0^\top \otimes \Phi_1, \Omega_1^\top \otimes \Phi_2, \ldots, \Omega_{\ell-1}^\top \otimes \Phi_\ell) \\ &= J_\zeta^\top \breve{F} J_\zeta\end{aligned}$$

We now turn our attention to the $\hat{F}$ (see Section 6.4.3 for the relevant notation).

First note that

$$\begin{aligned}\Psi^{\dagger}_{i,i+1} = \tilde{F}^{\dagger}_{i,i+1}\tilde{F}^{\dagger-1}_{i+1,i+1} &= (\Omega_{i-1} \otimes \Phi_i^\top)\tilde{F}_{i,i+1}(\Omega_i^\top \otimes \Phi_{i+1})\left((\Omega_i \otimes \Phi_{i+1}^\top)\tilde{F}_{i+1,i+1}(\Omega_i^\top \otimes \Phi_{i+1})\right)^{-1} \\ &= (\Omega_{i-1} \otimes \Phi_i^\top)\tilde{F}_{i,i+1}(\Omega_i^\top \otimes \Phi_{i+1})(\Omega_i^\top \otimes \Phi_{i+1})^{-1}\tilde{F}^{-1}_{i+1,i+1}(\Omega_i \otimes \Phi_{i+1}^\top)^{-1} \\ &= (\Omega_{i-1} \otimes \Phi_i^\top)\tilde{F}_{i,i+1}\tilde{F}^{-1}_{i+1,i+1}(\Omega_i \otimes \Phi_{i+1}^\top)^{-1} \\ &= (\Omega_{i-1} \otimes \Phi_i^\top)\Psi_{i,i+1}(\Omega_i \otimes \Phi_{i+1}^\top)^{-1}\end{aligned}$$

and so

$$\begin{aligned}\Sigma^{\dagger}_{i|i+1} &= \tilde{F}^{\dagger}_{i,i} - \Psi^{\dagger}_{i,i+1}\tilde{F}^{\dagger}_{i+1,i+1}\Psi^{\dagger\top}_{i,i+1} \\ &= (\Omega_{i-1} \otimes \Phi_i^\top)\tilde{F}_{i,i}(\Omega_{i-1}^\top \otimes \Phi_i) \\ &\quad - (\Omega_{i-1} \otimes \Phi_i^\top)\Psi_{i,i+1}(\Omega_i \otimes \Phi_{i+1}^\top)^{-1}(\Omega_i \otimes \Phi_{i+1}^\top)\tilde{F}_{i+1,i+1}(\Omega_i^\top \otimes \Phi_{i+1})(\Omega_i \otimes \Phi_{i+1}^\top)^{-\top} \\ &\qquad \cdot \Psi_{i,i+1}^\top(\Omega_{i-1} \otimes \Phi_i^\top)^\top \\ &= (\Omega_{i-1} \otimes \Phi_i^\top)(\tilde{F}_{i,i} - \Psi_{i,i+1}\tilde{F}_{i+1,i+1}\Psi_{i,i+1}^\top)(\Omega_{i-1}^\top \otimes \Phi_i) \\ &= (\Omega_{i-1} \otimes \Phi_i^\top)\Sigma_{i|i+1}(\Omega_{i-1}^\top \otimes \Phi_i)\end{aligned}$$

Also, $\Sigma^{\dagger}_{\ell} = \tilde{F}^{\dagger}_{\ell,\ell} = (\Omega_{\ell-1} \otimes \Phi_\ell^\top)\tilde{F}_{\ell,\ell}(\Omega_{\ell-1}^\top \otimes \Phi_\ell) = (\Omega_{\ell-1} \otimes \Phi_\ell^\top)\Sigma_\ell(\Omega_{\ell-1}^\top \otimes \Phi_\ell)$.

From these facts it follows that

$$\begin{aligned}
\Lambda^{\dagger-1} &= \operatorname{diag}\left(\Sigma_{1|2}^{\dagger}, \Sigma_{2|3}^{\dagger}, \ldots, \Sigma_{\ell-1|\ell}^{\dagger}, \Sigma_{\ell}^{\dagger}\right) \\
&= \operatorname{diag}\big((\Omega_0 \otimes \Phi_1^{\top})\Sigma_{1|2}(\Omega_0 \otimes \Phi_1^{\top}), (\Omega_1 \otimes \Phi_2^{\top})\Sigma_{2|3}(\Omega_1 \otimes \Phi_2^{\top}), \ldots, \\
&\qquad\qquad (\Omega_{\ell-2} \otimes \Phi_{\ell-1}^{\top})\Sigma_{\ell-1|\ell}(\Omega_{\ell-2} \otimes \Phi_{\ell-1}^{\top}), (\Omega_{\ell-1} \otimes \Phi_{\ell}^{\top})\Sigma_{\ell}(\Omega_{\ell-1}^{\top} \otimes \Phi_{\ell})\big) \\
&= \operatorname{diag}(\Omega_0 \otimes \Phi_1^{\top}, \Omega_1 \otimes \Phi_2^{\top}, \ldots, \Omega_{\ell-2} \otimes \Phi_{\ell-1}^{\top}, \Omega_{\ell-1} \otimes \Phi_{\ell}^{\top}) \operatorname{diag}\left(\Sigma_{1|2}, \Sigma_{2|3}, \ldots, \Sigma_{\ell-1|\ell}, \Sigma_{\ell}\right) \\
&\qquad\qquad \operatorname{diag}(\Omega_0^{\top} \otimes \Phi_1, \Omega_1^{\top} \otimes \Phi_2, \ldots, \Omega_{\ell-2}^{\top} \otimes \Phi_{\ell-1}, \Omega_{\ell-1}^{\top} \otimes \Phi_{\ell}) \\
&= J_{\zeta}^{\top} \Lambda^{-1} J_{\zeta}
\end{aligned}$$

Inverting both sides gives $\Lambda^{\dagger} = J_{\zeta}^{-1} \Lambda J_{\zeta}^{-\top}$.

Next, observe that

$$\Psi_{i,i+1}^{\dagger\top}(\Omega_{i-1}^{\top} \otimes \Phi_i)^{-1} = (\Omega_i \otimes \Phi_{i+1}^{\top})^{-\top} \Psi_{i,i+1}^{\top} (\Omega_{i-1} \otimes \Phi_i^{\top})^{\top} (\Omega_{i-1}^{\top} \otimes \Phi_i)^{-1} \quad = (\Omega_i^{\top} \otimes \Phi_{i+1})^{-1} \Psi_{i,i+1}^{\top}$$

from which it follows that

$$\begin{aligned}
\Xi^{\dagger\top} J_{\zeta}^{-1} &= \begin{bmatrix} I & & & & \\ -\Psi_{1,2}^{\dagger\top} & I & & & \\ & -\Psi_{2,3}^{\dagger\top} & I & & \\ & & \ddots & \ddots & \\ & & & -\Psi_{\ell-1,\ell}^{\dagger\top} & I \end{bmatrix} \operatorname{diag}((\Omega_0^{\top} \otimes \Phi_1)^{-1}, (\Omega_1^{\top} \otimes \Phi_2)^{-1}, \ldots, (\Omega_{\ell-1}^{\top} \otimes \Phi_{\ell})^{-1}) \\
&= \begin{bmatrix} (\Omega_0^{\top} \otimes \Phi_1)^{-1} & & & & \\ -\Psi_{1,2}^{\dagger\top}(\Omega_0^{\top} \otimes \Phi_1)^{-1} & (\Omega_1^{\top} \otimes \Phi_2)^{-1} & & & \\ & -\Psi_{2,3}^{\dagger\top}(\Omega_1^{\top} \otimes \Phi_2)^{-1} & (\Omega_2^{\top} \otimes \Phi_3)^{-1} & & \\ & & \ddots & \ddots & \\ & & & -\Psi_{\ell-1,\ell}^{\dagger\top}(\Omega_{\ell-2}^{\top} \otimes \Phi_{\ell-1})^{-1} & (\Omega_{\ell-1}^{\top} \otimes \Phi_{\ell})^{-1} \end{bmatrix} \\
&= \begin{bmatrix} (\Omega_0^{\top} \otimes \Phi_1)^{-1} & & & & \\ -(\Omega_0^{\top} \otimes \Phi_1)^{-1}\Psi_{1,2}^{\top} & (\Omega_1^{\top} \otimes \Phi_2)^{-1} & & & \\ & -(\Omega_1^{\top} \otimes \Phi_2)^{-1}\Psi_{2,3}^{\top} & (\Omega_2^{\top} \otimes \Phi_3)^{-1} & & \\ & & \ddots & \ddots & \\ & & & -(\Omega_{\ell-2}^{\top} \otimes \Phi_{\ell-1})^{-1}\Psi_{\ell-1,\ell}^{\top} & (\Omega_{\ell-1}^{\top} \otimes \Phi_{\ell})^{-1} \end{bmatrix} \\
&= \operatorname{diag}((\Omega_0^{\top} \otimes \Phi_1)^{-1}, (\Omega_1^{\top} \otimes \Phi_2)^{-1}, \ldots, (\Omega_{\ell-1}^{\top} \otimes \Phi_{\ell})^{-1}) \begin{bmatrix} I & & & & \\ -\Psi_{1,2}^{\top} & I & & & \\ & -\Psi_{2,3}^{\top} & I & & \\ & & \ddots & \ddots & \\ & & & -\Psi_{\ell-1,\ell}^{\top} & I \end{bmatrix} \\
&= J_{\zeta}^{-1} \Xi^{\top}
\end{aligned}$$

Combining $\Lambda^\dagger = J_\zeta^{-1}\Lambda J_\zeta^{-\top}$ and $\Xi^{\dagger\top} J_\zeta^{-1} = J_\zeta^{-1}\Xi^\top$ we have

$$\begin{aligned}\hat{F}^{\dagger-1} = \Xi^{\dagger\top}\Lambda^\dagger\Xi^\dagger = \Xi^{\dagger\top}J_\zeta^{-1}\Lambda J_\zeta^{-\top}\Xi^\dagger = (\Xi^{\dagger\top}J_\zeta^{-1})\Lambda(\Xi^{\dagger\top}J_\zeta^{-1})^\top &= (J_\zeta^{-1}\Xi^\top)\Lambda(J_\zeta^{-1}\Xi^\top)^\top \\ &= J_\zeta^{-1}\Xi^\top\Lambda\Xi J_\zeta^{-\top} \\ &= J_\zeta^{-1}\hat{F}^{-1}J_\zeta^{-\top}\end{aligned}$$

Inverting both sides gives $\hat{F}^\dagger = J_\zeta^\top \hat{F} J_\zeta$ as required.

□

Proof of Corollary 13. First note that a network which is transformed so that $G_{i,i}^\dagger = I$ and $\bar{A}_{i,i}^\dagger = I$ will satisfy the required properties. To see this, note that $\mathrm{E}[g_i^\dagger g_i^{\dagger\top}] = G_{i,i}^\dagger = I$ means that $g_i^\dagger$ is whitened with respect to the model's distribution by definition (since the expectation is taken with respect to the model's distribution), and furthermore we have that $\mathrm{E}[g_i^\dagger] = 0$ by default (e.g. using Lemma 15), so $g_i^\dagger$ is centered. And since $\mathrm{E}[a_i^\dagger a_i^{\dagger\top}]$ is the square submatrix of $\bar{A}_{i,i}^\dagger = I$ which leaves out the last row and column, we also have that $\mathrm{E}[a_i^\dagger a_i^{\dagger\top}] = I$ and so $a_i^\dagger$ is whitened. Finally, observe that $\mathrm{E}[a_i^\dagger]$ is given by the final column (or row) of $\bar{A}_{i,i}$, excluding the last entry, and is thus equal to 0, and so $a_i^\dagger$ is centered.

Next, we note that if $G_{i,i}^\dagger = I$ and $\bar{A}_{i,i}^\dagger = I$ then

$$\breve{F}^\dagger = \mathrm{diag}\left(\bar{A}_{0,0}^\dagger \otimes G_{1,1}^\dagger, \bar{A}_{1,1}^\dagger \otimes G_{2,2}^\dagger, \ \ldots, \ \bar{A}_{\ell-1,\ell-1}^\dagger \otimes G_{\ell,\ell}^\dagger\right) = \mathrm{diag}\left(I \otimes I, I \otimes I, \ \ldots, \ I \otimes I\right) = I$$

and so $-\alpha\breve{F}^{-1}\nabla h^\dagger = -\alpha\nabla h^\dagger$ is indeed a standard gradient descent update.

Finally, we observe that there are choices of Ω_i and Φ_i which will make the transformed model satisfy $G_{i,i}^\dagger = I$ and $\bar{A}_{i,i}^\dagger = I$. In particular, from the proof of Theorem 11 we have that $G_{i,j}^\dagger = \Phi_i^\top G_{i,j}\Phi_j$ and $\bar{A}_{i,j}^\dagger = \Omega_i \bar{A}_{i,j}\Omega_j^\top$, and so taking $\Phi_i = G_{i,i}^{-1/2}$ and $\Omega_i = \bar{A}_{i,i}^{-1/2}$ works.

The result now follows from Theorem 11.

□