Training Deep and Recurrent Networks with Hessian-Free Optimization

James Martens and Ilya Sutskever
jmartens@cs.toronto.edu, ilya@cs.utoronto.ca

December 1, 2012

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. It belongs to the broad class of approximate Newton methods that are practical for problems of very high dimensionality, such as the training objectives of large neural networks. Different algorithms that use many of the same key principles have appeared in the literatures of various communities under different names such as Newton-CG, CG-Steihaug, Newton-Lanczos, and Truncated Newton [27, 28], but applications to machine learning and especially neural networks, have been limited or non-existent until recently. With the work of Martens [22] and later Martens and Sutskever [23] it has been demonstrated that such an approach, if carefully designed and implemented, can work very well for optimizing non-convex functions such as the training objective for deep neural networks and recurrent neural networks (RNNs), given sensible random initializations. This was significant because gradient descent methods have been observed to be very slow and sometimes completely ineffective [16, 4, 17] on these problems, unless special non-random initializations schemes like layer-wise pre-training [16, 15, 3] are used. HF, which is a general optimization-based approach, can be used in conjunction with or as an alternative to existing pre-training methods and is more widely applicable, since it relies on fewer assumptions about the specific structure of the network.

In this report we will first describe the basic HF approach, and then examine well-known general purpose performance-improving techniques as well as others that are specific to HF (versus other Truncated-Newton type approaches) or to neural networks. We will also provide practical tips for creating efficient and correct implementations, and discuss the pitfalls which may arise when designing and using an HF-based approach in a particular application.

Notation	Description
$[x]_i$	The i-th entry of a vector x
$[A]_{i,j}$	The (i,j)-th entry a matrix A
$\mathbf{1}_m$	A vector of length m whose entries are 1
$\text{sq}(\cdot)$	The element-wise square of a vector or a matrix
$\text{vec}(A)$	The vectorization of a matrix A
f	The objective function
f_i	The objective function on case i
k	the current iteration of HF
θ_k	The parameter setting at the k-th HF iteration
n	The dimension of θ
δ_k	The variable being optimized by CG at the k-th HF iteration
M_{k-1}	A local quadratic approximation of f at θ_{k-1}
$\hat{M}_{k-1}$	A "damped" version of the above
B_{k-1}	The curvature matrix of M_{k-1}
$\hat{\text{B}}_{k-1}$	The curvature matrix of $\hat{M}_{k-1}$
h', ∇h	The gradient of a scalar function h
h'', $\nabla^2 h$	The Hessian of a scalar function h
$L(\cdot)$	The loss function
ρ	The reduction ratio $\dfrac{f(\theta_k) - f(\theta_{k-1})}{M_{k-1}(\delta_k)}$
$F(\theta)$	A function that maps parameters to predictions on all training cases
D	A damping matrix
P	A preconditioning matrix
$K_i(A, r_0)$	The subspace $\text{span}\{r_0, Ar_0, \ldots, A^{i-1}r_0\}$
ℓ	The number of layers of a feedforward net
z	The output of the network
m	The dimension of z
T	The number of time-steps of an RNN
λ	Strength constant for penalty damping terms
λ_j	j-th eigenvalue of curvature matrix
$\text{diag}(A)$	A vector consisting of the diagonal of the matrix A
$\text{diag}(v)$	A diagonal matrix A satisfying $[A]_{i,i} = [v]_i$

Table 1: A summary of the notation used. Note we will occasionally use some of these symbols to describe certain concepts that are local to a given sub-section. The subscripts "k" and "$k-1$", will often be dropped for compactness where they are implied from the context.

2 Feedforward Neural Networks

We now formalize feedforward neural networks (FNNs). Given an input x and setting of the parameters θ that determine weight matrices and the biases $(W_1, \ldots, W_{\ell-1}, b_1, \ldots, b_{\ell-1})$, the FNN computes its output y_ℓ by the following

recurrence:

$$y_{i+1} = s_i(W_i y_i + b_i)$$

where $y_1 = x$. The vectors y_i are the activations of the neural network, and the activation functions $s_i(\cdot)$ are some nonlinear functions, typically sigmoid or a tanh functions applied coordinate-wise.

Given a matching target t, the FNN's training objective for a single case $f(\theta; (x, t))$ is given by

$$f(\theta; (x, t)) = L(y_\ell; t)$$

where $L(z; t)$ is a loss function which quantifies how bad z is at predicting the target t. Note that L may not compare z directly to t, but instead may transform it first into some prediction vector p.

Finally, the training error, which is the objective of interest for learning, is obtained by averaging the losses $f(\theta; (x, t))$ over a set S of input-output pairs (aka training cases):

$$f(\theta) = \frac{1}{|S|} \sum_{(x,t)\in S} f(\theta; (x, t))$$

Algorithm 1 An algorithm for computing the gradient of a feedforward neural network

---

input: y_0; θ mapped to $(W_1, \ldots, W_{\ell-1}, b_1, \ldots, b_{\ell-1})$.
for all i **from** 1 **to** $\ell - 1$ **do**
  $x_{i+1} \leftarrow W_i y_i + b_i$
  $y_{i+1} \leftarrow s_{i+1}(x_{i+1})$
end for
$dy_\ell \leftarrow \partial L(y_\ell; t_\ell)/\partial y_\ell$ (t_ℓ is the target)
for all i **from** $\ell - 1$ **downto** 1 **do**
  $dx_{i+1} \leftarrow dy_{i+1} s'_{i+1}(x_{i+1})$
  $dW_i \leftarrow dx_{i+1} y_i^\top$
  $db_i \leftarrow dx_{i+1}$
  $dy_i \leftarrow W_i^\top dx_{i+1}$
end for
output: $\nabla f(\theta)$ as mapped from $(dW_1, \ldots, dW_{\ell-1}, db_1, \ldots, db_{\ell-1})$.

---

3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are the time-series analog of feedforward neural networks. RNNs model the mapping from an input sequence to an output sequence, and possess feedback connections in their hidden units that allow them to use information about past inputs to inform the predictions of future outputs. They may also be viewed as a special kind of feedforward net with a "layer" for each time-step of the sequence. But unlike in a feedforward network where each layer has its own parameters, the "layers" of an RNN share their parameters.

Their high-dimensional hidden state and nonlinear dynamics allow RNNs to learn very general and versatile representations, and to express highly complex sequential relationships. This representational power makes it possible, in principle, for RNNs to learn compact solutions for very difficult sequence modeling and labeling tasks. But despite their attractive qualities, RNNs did not enjoy widespread adoption after their initial discovery due to the perception that they were too difficult to properly train. The vanishing gradients problem [4, 17], where the derivative terms can exponentially decay to zero or explode during back-propagation through time is cited as one of the main reasons for this difficultly. In the case of decay, important back-propagated error signals from the output at future time-steps may decay nearly to zero by the time they have been back-propagated far enough to reach the relevant inputs.This makes the unmodified gradient a poor direction to follow if the RNN is to learn to exploit long-range input-output dependencies in the certain datasets.

Recent work by Martens & Sutskever [23] has demonstrated that HF is a viable method for optimizing RNNs on datasets that exhibit pathological long range dependencies that were believed difficult or impossible to learn with gradient descent. These problems were first examined by Hochreiter & Schmidhuber [18] where the proposed solution was to modify the RNN architecture with special memory units.

Basic RNNs are parameterized by three matrices and a special initial hidden state vector, so that $\theta \equiv (W_{xh}, W_{hh}, W_{zh}, h_0)$, where W_{xh} are the connections from the inputs to the hidden units, W_{hh} are the recurrent connections, and W_{zh} are the hidden-to-output connections. Given a sequence of vector-valued inputs $x = (x_1, \ldots, x_T)$ and vector-valued target outputs $t = (t_1, \ldots, t_T)$, the RNN computes a sequence of hidden states and predictions according to:

$$\begin{aligned} h_\tau &= s(W_{xh}x_\tau + W_{hh}h_{\tau-1}) \\ z_\tau &= W_{zh}h_\tau \end{aligned}$$

where h_0 is a special parameter vector of the initial state and $s(\cdot)$ is a nonlinear activation function (typically evaluated coordinate-wise).

The RNN learning objective for a single input-output pair of sequences (x, t) is given by:

$$f(\theta; (x, t)) = L(z; t) \equiv \sum_{\tau=1}^{T} L_\tau(z_\tau; t_\tau)$$

where L_τ is a loss function as in the previous section. As with FNNs, the objective function is obtained by averaging the loss over the training cases:

$$f(\theta) = \frac{1}{|S|} \sum_{(x,t)\in S} f(\theta; (x, t))$$

4 Hessian-free optimization basics

We consider the setting of unconstrained minimization of a twice continuously differentiable objective function $f : \mathbb{R}^n \to \mathbb{R}$ 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 previous setting of the parameters θ_{k-1}, iteration k produces a new iterate θ_k by minimizing a local quadratic model $M_{k-1}(\delta)$ of the objective $f(\theta_{k-1} + \delta)$, which is formed using gradient and curvature information local to θ_{k-1}. More precisely, we define

$$M_{k-1}(\delta) = f(\theta_{k-1}) + \nabla f(\theta_{k-1})^\top \delta + \frac{1}{2}\delta^\top \mathrm{B}_{k-1}\delta \tag{1}$$

where B_{k-1} is the "curvature matrix", and is chosen to be the Hessian $H(\theta_{k-1})$ of f at θ_{k-1} in the case of standard Newton's method. The new iterate θ_k is computed as $\theta_{k-1} + \alpha_k \delta_k^*$ where δ_k^* is the minimizer of eqn. 1, and the step-length $\alpha_k \in [0, 1]$ is chosen 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 8.8. 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 8.

When B_{k-1} is positive definite (PD), $M(\delta_k)$ will be bounded below and so its minimizer will exist, and will be given by $\delta_k^* = \theta_k - \mathrm{B}_{k-1}^{-1}\nabla f(\theta_{k-1})$, which is the standard Newton step. Unfortunately, for many good choices of B_{k-1}, such as the Hessian at θ_{k-1}, even computing the entire $n \times n$ curvature matrix B_{k-1}, let alone inverting it/solving the system $\mathrm{B}_{k-1}\delta_k = -\nabla f(\theta_{k-1})$ (at a computational cost of $O(n^3)$), 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) [14], 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 (PD), and $b \in \mathbb{R}^n$. CG 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. 1 we take $x = \delta$, $A = \mathrm{B}_{k-1}$ and $b = \nabla f(\theta_{k-1})$, noting that the constant term $f(\theta_{k-1})$ can be ignored.

> **Note:** From this point forward, we will abbreviate M_{k-1} with M and B_{k-1} 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 5), 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 $K_i(A, r_0) \equiv \mathrm{span}\{r_0, Ar_0, A^2r_0, ..., A^{i-1}r_0\}$, where $r_0 = Ax_0 - b$ and x_0 is the

initial solution [33]. Any other gradient based method applied directly to a quadratic function like M, even a very powerful one like Nesterov's accelerated gradient descent [29], 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 alg. 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. The preconditioning matrix P allows CG to operate within a transformed coordinate system and a good choice of P can substantially accelerate the method. This is possible despite the previously claimed optimality of CG because P induces a transformed Krylov subspace. 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 11.

[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 converge in only a few iteration while accelerated gradient descent will take much longer.

With practicality in mind, one can terminate CG according to various criteria, balancing the quality of the solution with the number of iterations required to obtain it (and hence number of matrix vector products – the main computational expense of the method). The approach taken by Martens [22] was to terminate CG based on a measure of relative progress optimizing M, computed as:

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

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 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

$$s_j < 0.0001 \tag{2}$$

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 8.7 of this paper 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 f than would be obtained by a fully converged solution (or equivalently, one produced by exact inversion of the curvature matrix).

When f is non-convex (as it is with 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 f, 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 include such techniques as restriction of the optimization over $M(\cdot)$ to a "trust-region", and the augmentation of M by penalty terms which are designed to encourage the minimizer of M to be somewhere in $\mathbb{R}^n$ where M remains a good approximation to f. Such approaches must be used with care, since restricting/penalizing the optimization of M too much will result in very reliable updates which are nonetheless useless due to being too "small". In section 8 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 [32] 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 will not eliminate the need for damping, Martens [22] nonetheless found that it was easier to use than the Hessian, producing better updates and requiring less damping. The computational and theoretical aspects of the GGN matrix and its use within HF will be discussed in detail in section 6.

Objective functions $f(\theta)$ that appear in machine learning are almost always defined as arithmetic averages over a training set S, and thus so can the gradient and the curvature-matrix vector products:

$$f(\theta) = \frac{1}{|S|} \sum_{(x,t) \in S} f(\theta; (x,t))$$

$$\nabla f(\theta) = \frac{1}{|S|} \sum_{(x,t) \in S} \nabla f(\theta; (x,t))$$

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

where $f(\theta; (x,t))$ is the objective and $\mathrm{B}(\theta; (x,t))$ the curvature matrix associated with the training pair (x,t).

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 "minibatches." And while it may seem natural to compute the matrix-vector products required by CG using a newly sampled minibatch at each iteration of alg. 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 [22] and independently by Byrd et al. [8] is to fix the minibatch used to define B for the entire run of CG. Minibatches and the practical issues which arise when using them will be discussed in more depth in section 12.

5 Exact Multiplication by the Hessian

To use the Hessian H of f 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

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_0, λ
Set $\delta_0 \leftarrow \vec{0}$
$k \leftarrow 1$
while solution is not satisfactory **do**
  Select a set of points S for the gradient — sec. 12
  $b \leftarrow -\nabla f(\theta_{k-1})$ on S — sec. 2
  Select a set of points S' for the curvature — sec. 12
  Compute a preconditioner P at θ_k — sec. 11
  Compute a damping matrix D_k — sec. 8
  Define $A(v) \equiv \mathrm{G}(\theta_{k-1})v + \lambda D_k v$ on S' — sec. 6
  Choose a decay constant $\zeta \in [0, 1]$ — sec. 10
  $\delta_k \leftarrow \mathrm{PCG}(b, A, \zeta\delta_{k-1}, P)$ — alg. 2
  Update λ with the Levenberg-Marquardt method — sec. 8.5
  Choose/compute a step-size α — sec. 8.8
  $\theta_k \leftarrow \theta_{k-1} + \alpha\delta_k$
  $k \leftarrow k + 1$
end while

$\nabla f(\theta)$ in the direction v, and so by the definition of directions derivatives,

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

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" [35, 30], although we follow the exposition of Pearlmutter [31], 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 differentiate 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 X in direction v:

$$\mathrm{R}_v X = \lim_{\varepsilon \to 0} \frac{X(\theta + \varepsilon v) - X(\theta)}{\varepsilon} = \frac{\partial X}{\partial \theta} v \tag{3}$$

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

$$\mathrm{R}_v(X+Y) = \mathrm{R}_v X + \mathrm{R}_v Y \qquad \text{linearity} \tag{4}$$

$$\mathrm{R}_v(XY) = (\mathrm{R}_v X)Y + X\mathrm{R}_v Y \qquad \text{product rule} \tag{5}$$

$$\mathrm{R}_v(h(X)) = (\mathrm{R}_v X)h'(X) \qquad \text{chain rule} \tag{6}$$

where h' denotes the Jacobian of h. From this point on we will abbreviate R_v as simply "R" to keep the notation compact.

Noting that $Hv = \mathrm{R}(\nabla f(\theta))$, computing the Hessian-vector product amounts to computing $\mathrm{R}(\nabla f(\theta))$ by applying these rules recursively to the computational graph for $\nabla f(\theta)$, 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 $h(\theta)$, 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 M 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 $a_i = \gamma_i(z_i)$ of their input z_i. Each input vector z_i is formally the concatenation the output of each of its parent nodes $a_j \in P_i$. The input θ is distributed over a set of input nodes $\mathcal{I} \subset \{1, ..., M\}$ and the outputs are computed at output nodes $\mathcal{O} \subset \{1, ..., M\}$.

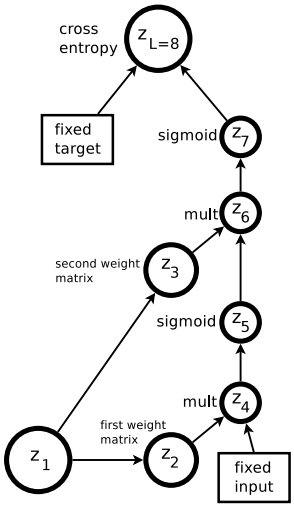

Figure 1: An example of a computational graph of the loss of a neural network objective. The weights are considered the inputs here.

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

1. For each $i \in \mathcal{I}$ set a_i according to entries of θ
2. For i from 1 to M such that $i \notin \mathcal{I}$:

$$z_i = \mathrm{concat}_{j \in P_i} a_j$$
$$a_i = \gamma_i(z_i)$$

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

where P_i is the set of parents of node i.

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 $\mathrm{R}(h(\theta))$ can be computed as follows:

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

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

$$\mathrm{R}z_i = \mathrm{concat}_{j \in P_i} \mathrm{R}a_j \tag{7}$$

$$\mathrm{R}a_i = \gamma_i'(z_i)\mathrm{R}z_i \tag{8}$$

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

where $\gamma_i'(z_i)$ is the Jacobian of γ_i.

In general, computing $\gamma_i'(z_i)$ (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 $\gamma_i'(z_i)$ in general, so all of the z_i's (or equivalently all of the a_i's) must either be computed in tandem with the $\mathrm{R}a_i$'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).

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 7 may require considerable extra storage when computing matrix-vector products over large minibatches.

Algorithm 5 gives the pseudo-code for computing the Hessian-vector product associated with the feedforward neural network defined in section 2. The parameter vector θ defines the weight matrices and the biases $(W_1, \ldots, W_{\ell-1}, b_1, \ldots, b_{\ell-1})$ and v maps analogously to $(\mathrm{R}W_1, \ldots, \mathrm{R}W_{\ell-1}, \mathrm{R}b_1, \ldots, \mathrm{R}b_{\ell-1})$. This algorithm was derived by applying the rules 4–6 to each line of alg. 2, where various required quantities such as y_i are assumed to be available either because they are cached, or by running the corresponding lines of alg. 2 in tandem.

6 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 the update δ. This problem can be addressed in a multitude of ways. For example, imposing a trust-region (sec. 8.6) will constrain the optimization, or a penalty-based damping method (sec. 8.1) will effectively add a positive semi-definite (PSD) contribution to B which may render it positive definite (PD). Another solution specific to truncated Newton methods 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 alg. 2), a solution which 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

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

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

input: v mapped to $(\mathrm{R}W_1, \ldots, \mathrm{R}W_{\ell-1}, \mathrm{R}b_1, \ldots, \mathrm{R}b_{\ell-1})$

$\mathrm{R}y_0 \leftarrow 0$ (since y_0 is not a function of the parameters)

for all i **from** 1 **to** $\ell - 1$ **do**

$\mathrm{R}x_{i+1} \leftarrow \mathrm{R}W_i y_i + W_i \mathrm{R}y_i + \mathrm{R}b_i$ (product rule)

$\mathrm{R}y_{i+1} \leftarrow \mathrm{R}x_{i+1} s'_{i+1}(x_{i+1})$ (chain rule)

end for

$$\mathrm{R}dy_\ell \leftarrow \mathrm{R}\left(\frac{\partial L(y_\ell; t_\ell)}{\partial y_\ell}\right) = \frac{\partial\{\partial L(y_\ell; t_\ell)/\partial y_\ell\}}{\partial y_\ell}\mathrm{R}y_\ell = \frac{\partial^2 L(y_\ell; t_\ell)}{\partial y_\ell^2}\mathrm{R}y_\ell$$

for all i **from** $\ell - 1$ **downto** 1 **do**

$$\begin{aligned}\mathrm{R}dx_{i+1} &\leftarrow \mathrm{R}dy_{i+1} s'_{i+1}(x_{i+1}) + dy_{i+1} R\left\{s'_{i+1}(x_{i+1})\right\} && \text{(product rule)}\\ &= dy_{i+1} s''_{i+1}(x_{i+1})\mathrm{R}x_{i+1} && \text{(chain rule)}\end{aligned}$$

$\mathrm{R}dW_i \leftarrow \mathrm{R}dx_{i+1} y_i^\top + dx_{i+1}\mathrm{R}y_i^\top$ (product rule)

$\mathrm{R}db_i \leftarrow \mathrm{R}dy_i$

$\mathrm{R}dy_i \leftarrow \mathrm{R}W_i^\top dx_{i+1} + W_i^\top \mathrm{R}dx_{i+1}$ (product rule)

end for

output: $H(\theta)v$ as mapped from $(\mathrm{R}dW_1, \ldots, \mathrm{R}dW_{\ell-1}, \mathrm{R}db_1, \ldots, Rdb_{\ell-1})$.

Schraudolph [32], which is a provably positive semidefinite curvature matrix that can be viewed as an approximation to the Hessian. We will denote this matrix as G.

The generalized Gauss-Newton matrix can be derived in at least two ways, and both require that the objective $f(\theta)$ be expressed as the composition of two functions as $f(\theta) = L(F(\theta))$ where L is convex. In a neural network setting, F maps the parameters θ to a m-dimensional vector of the neural network's outputs $z \equiv F(\theta)$, and $L(z)$ is a convex "loss function" which typically measures the difference between the network's outputs (which may be further transformed within L to produce "predictions" p) and the targets. For RNNs, z will be a vector of the outputs from *all* the time-steps and L computes the sum over losses at each one of them.

One way to view the GGN matrix is as an approximation of H where we drop certain terms that involve the 2nd-derivatives of F. Applying the chain rule to compute the Hessian of f (at θ_{k-1}), we get:

$$\begin{aligned} f &= L(F(\theta)) \\ \nabla f(\theta) &= J^\top \nabla L \\ f''(\theta) &= J^\top L'' J + \sum_{i=1}^{m} [\nabla L]_i ([F]_i)'' \end{aligned}$$

where J denotes the Jacobian of F, ∇L is the gradient of $L(z)$ w.r.t. z, and all 1st and 2nd derivatives are evaluated at θ_{k-1}. The first term $J^\top L'' J$ is a positive definite matrix whenever $L(z)$ is convex in z, and is defined as the GGN matrix. Note that in the special case where $L(z) = 1/2\|z\|^2$ (so that $L'' = I$)

we recover the standard Gauss-Newton matrix usually seen in the context of non-linear least squares optimization and the Levenberg-Marquardt algorithm [25].

Martens and Sutskever [23] showed that the GGN matrix can also be viewed as the Hessian of a particular approximation of f constructed by replacing F with its 1st-order approximation. Consider a local convex approximation $\hat{f}$ to f at θ_{k-1} that is obtained by taking the first-order approximation $F(\theta) \approx F(\theta_{k-1}) + J\delta$ (where $\delta = \theta - \theta_{k-1}$):

$$\hat{f}(\delta) = L(F(\theta_{k-1}) + J\delta) \tag{9}$$

The approximation $\hat{f}$ is convex because it is a composition of a convex function and an affine function. It is easy to see that $\hat{f}$ and f have the same derivative when $\delta = 0$, because

$$\nabla \hat{f} = J^\top \nabla L = J^\top \nabla L$$

which is precisely the derivative of f at θ_{k-1}. And the Hessian of $\hat{f}$ at $\delta = 0$ is precisely the GGN matrix:

$$\hat{f}'' = J^\top L'' J = \mathrm{G}$$

Note that it may be possible to represent a function f with multiple distinct compositions of the form $L(F(\theta))$, and each of these will give rise to a slightly different GGN matrix. For neural networks, a natural choice for the output vector z is often just to identify it as the output of the final layer (i.e., y_ℓ), however this may not always result in a convex L. As a rule of thumb, it is best to define L and F in way that L performs "as much of the computation of f as possible" (but this is a problematic concept due to the existence of multiple distinct sequences of operations for computing f). For the case of neural networks with a softmax output layer and cross-entropy error, it is best to define L so that it performs both the softmax *and* then the cross-entropy, while F computes only the inputs to the soft-max function. This is also the recommendation made by Schraudolph [32]. A possible reason that this choice works best is due to the fact that F is being replaced with its first-order approximation whose range is unbounded. Hence the GGN matrix makes sense only when L's input domain is $\mathbb{R}^m$ (as opposed to $[0, 1]^m$ for the cross-entropy error), since this is the range of the 1st-order approximation of F.

6.1 Multiplying by the Gauss-Newton matrix

For the GGN matrix to be useful in the context of HF, we need an efficient algorithm for computing the $\mathrm{G}v$ products. Methods for multiplying by the classical Gauss-Newton matrix are well-known in the optimization literature [30], and these methods were generalized by Schraudolph [32] for the GGN matrix, using an approach which we will now describe.

We know from the previous section that the GGN matrix can be expressed as the product of three matrices: $\mathrm{G}v = J^\top L'' J v$. Thus multiplication of a

vector v by the GGN matrix 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 $\mathrm{R}_v\{F(\theta)\}$, and thus can be efficiently computed with the R-method as in section 5. Next, given that the loss function L is usually simple, multiplication of Jv by L'' is also simple (sec. 6.2). Finally, we multiply the vector $L''Jv$ by the matrix $J^\top$ using the backpropagation algorithm. Note that the backpropagation algorithm takes the derivatives w.r.t. the predictions (∇L) as inputs, and returns the derivative w.r.t. the parameters, namely $J^\top \nabla L$, but we can replace ∇L with any vector we want.

Algorithm 5 An algorithm for computing $\mathrm{G}v$ of a feedforward neural network.

input: $\mathrm{R}W_1, \ldots, \mathrm{R}W_{\ell-1}, \mathrm{R}b_1, \ldots, \mathrm{R}b_{\ell-1}$.
$\mathrm{R}y_0 \leftarrow 0$ (y_0 is not a function of the parameters)
for all i **from** 1 **to** $\ell - 1$ **do**
  $\mathrm{R}x_{i+1} \leftarrow \mathrm{R}W_i y_i + W_i \mathrm{R}y_i + \mathrm{R}b_i$ (product rule)
  $\mathrm{R}y_{i+1} \leftarrow \mathrm{R}x_{i+1} s'_{i+1}(x_{i+1})$
end for
$\mathrm{R}dy_\ell \leftarrow \dfrac{\partial^2 L(y_\ell; t_\ell)}{\partial y_\ell^2} \mathrm{R}y_\ell$
for all i **from** $\ell - 1$ **downto** 1 **do**
  $\mathrm{R}dx_{i+1} \leftarrow \mathrm{R}dy_{i+1} s'_{i+1}(x_{i+1})$
  $\mathrm{R}dW_i \leftarrow \mathrm{R}dx_{i+1} y_i^\top$
  $\mathrm{R}db_i \leftarrow \mathrm{R}dx_{i+1}$
  $\mathrm{R}dy_i \leftarrow \mathrm{R}W_i^\top dx_{i+1}$
end for
output: $(\mathrm{R}dW_1, \ldots, \mathrm{R}dW_{\ell-1}, \mathrm{R}b_1, \ldots, \mathrm{R}b_{\ell-1})$.

As observed by Martens and Sutskever [23], the second interpretation of the GGN matrix given in the previous section immediately implies an alternative method for computing $\mathrm{G}v$ products. In particular, we can use the R method from sec. 5 to efficiently multiply by the Hessian of $\hat{f}$, given a computational graph for $\nabla \hat{f}$. While doing this would require one to replace the part of the forward pass corresponding to F with a multiplication by the analogous Jacobian evaluated at θ_{k-1} (which can be done using the R operator method applied to f), a simpler approach is just to modify the algorithm for computing ∇f so that all derivative terms involving intermediate quantities in the back-propagation through F are treated as "constants", which while they are computed from θ_{k-1}, are formally independent of θ. This version will only compute $\hat{f}$ properly for $\theta = \theta_{k-1}$, but this is fine for our purposes since this is the point at which we wish to evaluate the GGN matrix.

Algorithm 5 multiplies by the GGN matrix for the special case of a feedforward neural network and is derived using this second technique.

6.2 Typical losses

In this section we present a number of typical loss functions and their Hessians (table 2). The function $p(z)$ computes the predictions p from the network outputs z. These losses are convex and it is easy to multiply by their Hessians

Name	$L(z;t)$	$\nabla L(z;t)$	$L''(z;t)$	p
Squared error	$\frac{1}{2}\|p-t\|^2$	$-(p-t)$	I	$p=z$
Cross-entropy error	$-t\log p-(1-t)\log(1-p)$	$-(p-t)$	$\mathrm{diag}(p(1-p))$	$p=\mathrm{Sigmoid}(z)$
Cross-entropy error (multi-dim)	$-\sum_i [t]_i \log[p]_i$	$-(p-t)$	$\mathrm{diag}(p)-pp^\top$	$p=\mathrm{Softmax}(z)$

Table 2: Typical losses with their derivatives and Hessians. The loss L and the nonlinearity $p(z)$ are "matching", which means that the Hessian is independent of the target t and is PSD.

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, tote that because it formally includes the computation of the predictions p from the network outputs z (assumed to lie anywhere in $\mathbb{R}^m$) in the loss function itself (instead of in the activation function s_ℓ at the output layer), s_ℓ should be set to the identity function.

6.3 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 $J^\top L'' J$ will usually be PSD only when L'' is, which is a problem that can be addressed in one of several ways. For example, if our loss is $L(y;t) = \|\tanh(y) - t\|^2/2$, which is non-convex, we could formally treat the tanh nonlinearity as being part of F (replacing F with $\tanh \circ F$), and redefine the loss L as $\|y-t\|^2/2$. Another trick which may work would be to approximate the loss-Hessian L'' with a positive definite matrix, which could be done, say, by adding a scaled multiple of the diagonal to L'', or by taking the eigen-decomposition of L'' and discarding the eigenvectors that have negative eigenvalues.

7 Implementation details

7.1 Efficiency via parallelism

A good implementation of HF can make fruitful use of parallelization in two ways.

First, it can benefit from model parallelism, which is the ability to perform the input and output computations associated with each neuron in a given layer parallel. Although model parallelism accelerates any optimization algorithm

that is applied to neural networks, current hardware is incapable of fully taking advantage of it, mostly because weights are stored in a centralized memory with very limited bandwidth.

Second, an implementation of HF can benefit from data parallelism, where the computation of the gradient or curvature matrix vector products is performed independently and in parallel across the training cases in the current minibatch. Data parallelism is much easier to exploit in current hardware because it requires minimal communication, in stark contrast to model parallelism, which requires frequent and rapid communication of unit activations. The potential speedup offered by data parallelism is limited by the gains that can be derived from using larger minibatches to compute updates in HF, as well as the sheer amount of parallel computing power available.

HF tends to benefit from using relatively large minibatches, especially compared to first-order methods like stochastic gradient descent, and so exploiting data parallelism may bring significant reductions in computation time. Nonetheless, there is a point of diminishing returns after which making the minibatch larger provides limited or no benefit in terms of the quality of the update proposals (as measured by how much they reduce f).

Data parallelism is typically implemented using vectorization, which is a way of specifying a single computational process that is independently performed on every element of a vector. Since most implementations that use vectorization (e.g. GPU code) become more efficient per case as the size of the minibatch increases, there is a distinct benefit to using larger minibatches (up until the aforementioned point of diminishing returns, or the point where the implementations parallel computing resources are fully utilized).

Most of the computation performed by HF consists of computing the GGN vector products and fortunately it is possible to obtain a 50% speedup over a naive implementation of the GGN vector products using activity caching. Recall that a multiplication by the GGN matrix consists of a multiplication by J which is followed by a multiplication by $J^\top$, both of which require the neural network's unit activations (the y_i's in FNNs or y_τ's in RNNs). However, given that the network's activations are a function of only θ, and that CG multiplies different vectors by the same GGN matrix (so its setting of θ is fixed), it is possible to cache the network's activations y_i and to reuse them for all the GGN-vector products made during an entire run of CG.

When a model is very large, which is the case for a large RNN with a large number of time-steps T, the unit activations produced by even a modestly-sized minibatch may become too numerous to fit in memory. This is especially a problem for GPU implementations since GPUs typically have much less memory available than CPUs. This has two undesirable consequences. First, activity caching becomes impossible, and second, it necessitates the splitting of a large minibatch into many smaller "computational minibatches" (the results from which will be summed up after each has been processed), which can greatly reduce the cost-effectiveness of vectorization.

The problem can be addressed in at least 2 ways. One is to cache the activations in a larger but slower memory storage (e.g. the CPU memory), and

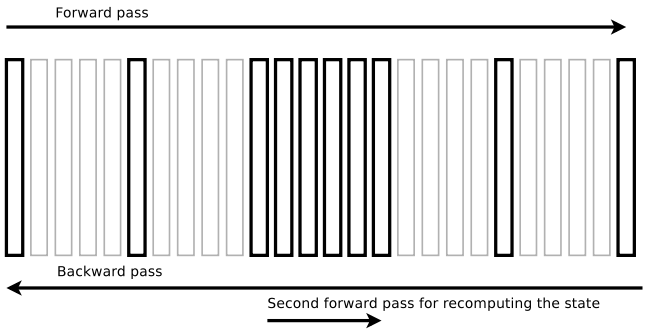

Figure 2: An illustration of the method for conserving the memory of the RNN. Each column represents a hidden state of an RNN, and only the highlighted columns reside in memory at any given time.

to retrieve them as needed. This is often faster than the use of many smaller minibatches.

Another way involves reducing the storage requirements at the cost of performing re-computation of some of the states. In particular, we store the hidden states at every multiple of $\sqrt{T}$ time-steps (thus reducing the storage requirement by a factor of $\sqrt{T}$), and recompute sequences of $\sqrt{T}$ between these "checkpoints" as they become needed, discarding them immediately after use. Due to the way the forward and backwards passes involved in computing gradients and matrix-vector products go through the time-steps in linear order, the state at each time-step needs to be recomputed at most once in the case of the gradient, and twice in the case of the matrix-vector product.

7.2 Verifying the correctness of G products

A well-known pitfall for neural networks practitioners is an incorrect implementation 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 Truncated Newton optimizer like HF, as we must also verify the correctness of the curvature-matrix vector products. When $\mathrm{B} = H$, 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 $\mathrm{G}v$ products.

Given that $\mathrm{G} = J^\top L'' J$ so $\mathrm{G}v = J^\top (L''(Jv))$, computing the G-vector products via finite differences reduces to doing this for Jw, $L''w$ and $J^\top w$ for arbitrary vectors w of appropriate dimension (not necessarily the same for each).

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

2. For $L''w$ we can simply approximate L'' using one of the aforementioned

finite-differences implementations that are available for approximating Hessians.

3. For $J^\top w$ we exploit $[J]_{j,i}^\top = [Je_j]_i$ 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 $\mathrm{G}e_j$ 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 [5]. 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).

8 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 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 net 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 f. And for some very simply 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 [7], 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 f 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 f).

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 f), 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-net training objectives, damping schemes which tend to produce updates that give the best reductions in f in the short term, may not always yield the best global optimization performance in the long term.

8.1 Tikhonov Damping

"Tikhonov regularization" or Tikhonov damping [3] 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

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

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

As $\lambda \to \infty$, the damped curvature matrix $\hat{\mathrm{B}}$ tends to a multiple of the identity and the minimizer δ^* has the property that $\delta^* \to \nabla f(\theta)/\lambda$, 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 $\mathrm{B} = V\Sigma V^\top$ where $V = [v_1|v_2|\dots|v_n]$ are eigenvectors of B (which are orthonormal since B is symmetric), and $\Sigma \equiv \mathrm{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$ the diagonal matrix of eigenvalues, then $\hat{\mathrm{B}} = V\Sigma V^\top + \lambda I = V\Sigma V^\top + \lambda V V^\top = V(\Sigma+\lambda I)V^\top$. Thus the curvature associated with each eigenvector v_j in the damped matrix is given by $v_j^\top\hat{\mathrm{B}}v_j = \lambda_j + \lambda$.

This modulation of the curvature has profound effect on the inverse of $\hat{\mathrm{B}}$ since $\hat{\mathrm{B}} = V^\top(\Sigma+\lambda I)^{-1}V$, 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

$$\delta^* = -\sum_j \frac{v_j^\top\nabla f(\theta_{k-1})}{\lambda_j + \lambda}v_j$$

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

[3] a name which we will use to avoid confusion with the other meaning of term regularization in the learning context

too small, the quadratic model $\hat{M}$ will be too aggressively optimized by CG, resulting in a very large parameter update (particular in directions of low curvature) which may cause an increase in f 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 f gives the same update as $\lambda = 2$ would for $2f$), and other more subtle properties of f, 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 8.5. Note that Tikhonov damping is the method used by Lecun et al. [21, 20], where the constant "μ" (which is *not* adapted) plays the role of λ.

It is worth noting that Vinyals and Povey [34] have recently developed an alternative approach to Tikhonov damping, based on the idea of directly optimizing f over a K-dimensional Krylov basis generated by CG (or equivalently a Lanczos iteration). Because the Krylov subspace generated using a $\hat{\mathrm{B}} = \mathrm{B} + \lambda I$ doesn't depend on λ (assuming a CG initialization of $x_0 = 0$), 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 f 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, 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.

8.2 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 L2 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 a 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 only layer j, and imposing a Tikhonov damping penalty consisting of an equally weighted sum of squared changes over *all* entries of θ (given by $\lambda/2\|\delta\|^2 = \lambda/2\sum_{i=1}^{n}\delta_i^2$) 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 $w = [\theta]_i$. 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 ϕ 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 $C^\top \nabla f$ and $C^\top \mathrm{G} C$ respectively[4]. The optimal update thus becomes:

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

For our particular example, C is a diagonal matrix satisfying $[C]_{i,i} = \exp([\theta]_i)$ for reparameterized entries of θ, and $[C]_{i,i} = 1$ for the rest.

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 reasonable (when taken in the new parameterization). In particular, a reparameterized weight w (and hence f) 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 below zero, resulting in an update which will fail to make a substantial change in $[\theta]_i$. Conversely, if $[\theta]_i$ is large 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 small 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 and any of the other quadratic penalty based damping methods we will discuss in the following sections, can all be made arbitrarily strong 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

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

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 sub-optimization of $\hat{M}$ to be less constrained and thus produce larger and more useful updates to θ.

8.3 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 $\hat{\mathrm{B}} = \lambda I$ and $\delta^* = -1/\lambda \nabla f(\theta)$.

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-1}) and is designed to better respect the local scale properties of f at θ_{k-1}. In particular, instead of adding the penalty term $\lambda/2\|\delta\|^2$ to M we may instead add $\lambda/2\|\delta\|^2_{D_{k-1}} = \lambda/2\delta^\top D_{k-1}\delta$, where D_{k-1} is some symmetric positive definite (PD) matrix that depends on θ_{k-1}. Such a term may provide a more meaningful measure of change in θ, by accounting for the sensitivity properties of f more precisely. We call the matrix D_{k-1}, the damping matrix, and will drop the subscript $k-1$ for brevity. Scale sensitive damping is implemented in HF by working with a "damped" curvature matrix given $\hat{\mathrm{B}} = \mathrm{B} + \lambda D$, where the required matrix-vector products can be computed using as $\hat{\mathrm{B}}v = \mathrm{B}v + \lambda D v$, 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:

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

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 $\frac{\lambda}{2}\|\theta^\dagger\|^2 = \frac{\lambda}{2}\|\phi(\theta)\|^2$ w.r.t. to the new parameters θ, where $\theta^\dagger = \phi(\theta)$ 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 (i.e. $D = \text{diag}(\text{diag}(\text{B}))$), a choice made in the classical Levenberg-Marquardt algorithm. With this choice, the update $\delta^{\dagger^*}$ 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 f (i.e. optimizing βf instead of f 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 f. This is because with this choice, the damping-modified curvature matrix $\hat{\text{B}}$ is simply $(1 + \lambda)\text{B}$, 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 $1/(1 + \delta)$. In section 8.8 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 approaches. While the Tikhonov approach arguably makes too few assumptions about the local properties of f, damping approaches based on $D = \text{B}$ 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 $D = \text{B}$ 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 fig. 8.3.

In order to explain such degeneracies and understand why choices like $D = \text{B}$ 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-

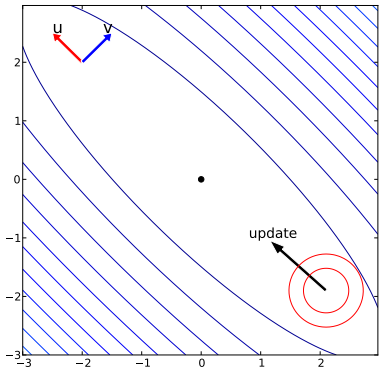

Figure 3: A 2D toy example of how using $D = \text{diag}(B)$ results in an overly restricted update. Let $u = [-1, 1]^\top$ and $v = [1, 1]^\top$, and let B be $uu^\top + avv^\top$ where a is large (e.g. 10^4, although we take $a = 15$ for display purposes). This matrix is full rank, and its diagonal 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.

order effects (and even certain unmodelled 2nd-order effects in the case of the GGN matrix) and it is the goal of damping to help compensate for this. By taking $D = \text{B}$ we are penalizing directions according to their curvature, and so are in some sense assuming that the relative strength of the contribution to f 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 f, 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 11.2) seems like an arbitrary choice, since in general there may not be anything particularly special or natural about whatever default parameterization of f 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 f. Or perhaps some useful information could be gleaned

from examining the structure of the computational graph of f. 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.

8.4 Structural Damping

Recurrent Neural Networks 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 [22] to RNNs can yield reasonable results, performing well on a family of synthetic pathological problems [18, 23] designed to have very long-range temporal dependencies of up to 100 time-steps. However Martens and Sutskever [23] found that performance could be made substantially better and more robust using an idea called "structural damping".

Martens and Sutskever [23] 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 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_{hh}. While these sorts of problems exist with deep feedforward neural networks like the autoencoders considered in Hinton and Salakhutdinov [16] and Martens [22], 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 [4, 17]. 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 cetain intermediate quantities that appear in the evaluation of f, 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 the large changes in the hidden state sequence, which due to their highly nonlinear nature, tend to correspond to catastrophic breakdowns in the accuracy of the quadratic approximation.

Speculatively, structural damping may have another more subtle benefit for RNN learning. It is known [19, 23] 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. 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 we first re-express the nonlinear objective $f(\theta)$ as a composition of functions $L(z(h(\theta), \theta))$, where $h(\theta)$ computes the hidden states (whose change we wish to penalize), $z(h, \theta)$ computes the outputs, and $L(z)$ computes the loss.

Given the current parameter setting θ_{k-1}, the local (undamped) quadratic approximation is given by $M(\delta)$ and its curvature is B. We prevent large changes in the hidden state by penalizing the distance between $h(\theta_{k-1} + \delta)$ and $h(\theta_{k-1})$ according to the penalty function

$$S(\delta) = d(h(\theta_{k-1} + \delta); h(\theta_{k-1}))$$

where d is a distance function or loss function such as a squared error or the cross-entropy[5].

Ideally, we would define the damped local objective as:

$$\hat{M}^{\dagger}(\delta) = M(\delta) + \mu S(\delta) + \lambda I$$

where μ is a strength constant similar to λ. But since we cannot minimize a non-quadratic objective with CG, we must resort to using a local quadratic approximation to $S(\delta)$. This will be given by $\delta^{\top} D_{k-1} \delta / 2$ where D_{k-1} is the Gauss-Newton matrix of the penalty function $S(\delta)$ at $\delta = 0$. Note that the quadratic approximation to $S(\delta)$ does not have a linear term because $\delta = 0$ is a minimum of S.

Fortunately, it is straightforward to multiply by the generalized Gauss-Newton matrix of S using the techniques outlined in section 6. Thus we could compute the products $\mathrm{B}v$ and $\mu D_{k-1} v$ using two separate Gauss-Newton matrix-vector products, adding together the results, approximately doubling the computational burden. In order to avoid this, we can instead compute the sum $(B_{k-1} + \mu D_{k-1})v$ directly by exploiting the fact that $S(\delta)$ can be computed much more efficiently by reusing the $h(\theta_k + \delta)$ which gets computed as an intermediate quantity for $f(\theta_k + \delta)$. Indeed, consider a neural network whose output units include the hidden state as well as the predictions:

$$c(\theta) \equiv [h(\theta), z(h(\theta), \theta)]$$

and whose loss function is given by $L_{\mu}(h, y) = \mu d(h; h(\theta_{k-1}) + L(y)$, so that we have $f(\theta) + \mu S(\theta) = L_{\mu}(c(\theta))$. This "new" neural network includes structural damping in its loss, and any automatic routines computing the required Jacobian-vector products for c will be no more expensive than the analogous

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

routines in the original network. Multiplication by the Hessian of the loss $L''_\mu = \begin{pmatrix} L''(y) & 0 \\ 0 & \mu d''(h; h(\theta_{k-1})) \end{pmatrix}$ is also easy and can be done block-wise.

8.5 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 sub-section) must be constantly adapted to keep up with the changing local curvature properties of f.

Fortunately, we have found that the well-known Levenberg-Marquardt (LM) heuristic, which is usually used in the context of the LM method [25] to be 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

$$\rho \equiv \frac{f(\theta_{k-1} + \delta_k) - f(\theta_{k-1})}{M_{k-1}(\delta_k)} \tag{10}$$

The reduction ratio measures the ratio of the reduction in the objective $f(\theta_{k-1} + \delta_k) - f(\theta_{k-1})$ 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 more conservative and thus lie somewhere that the quadratic model more 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:

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

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

Note that the above heuristic is valid in the situation where $\rho < 0$, which can arise in one of two ways. The first way is where $M_{k-1} < 0$ and $f(\theta_{k-1} + \delta_k) - f(\theta_{k-1}) > 0$, which means that the quadratic approximation is very inaccurate around δ^* and doesn't even get the sign of the change right. The other possibility is that $M_{k-1} > 0$ and $f(\theta_{k-1} + \delta_k) - f(\theta_{k-1}) < 0$, which can only occur if CG is initialized from a nonzero previous solution and doesn't make sufficient progress from that point to obtain a negative value M_{k-1} before being terminated/truncated. When this occurs one should either allow CG to use more iterations or possibly initialize the next run of CG from 0 (as this will guarantee that $M_{k-1} < 0$, since $M_{k-1}(0) = 0$ and CG decreases M_{k-1} monotonically).

While the definition of ρ in eqn. 10 uses the undamped quadratic in the denominator, the damped quadratic approximation $\hat{M_{k-1}}$ 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 8.7), 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 10). 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 giving provable strong local convergence guarantees. 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 f 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 attempt to be smarter and terminate CG based on the value of f, for example, 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 f 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.

8.6 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 by:

$$\delta_R^* = \text{argmin}_{\delta:\delta\in R} M(\delta)$$

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 f 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 f. 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 connection between 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

$$\text{argmin}_{\delta:\delta\in R} M(\delta) = \text{argmin}_\delta M(\delta) + \frac{\lambda}{2}\|\delta\|_Q^2$$

This result is valid for all quadratic functions M, even when B is indefinite, and can be proved using Lagrange multipliers.[6]

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[7], 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 [26], 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 [12] 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 "optimal" λ over the parameter space, it still needs to be adjusted using some heuristic. Indeed, one heuristic which is advocated for adjusting r [30] is precisely the Levenberg-Marquardt heuristic discussed in section 8.5 which is already quite

[6] For completeness, we present an outline of the proof here: Consider the Lagrangian $L(\delta, \lambda) = M(\delta) + (r - \frac{1}{2}\delta^\top Q\delta)\lambda$. It is known that there exists a λ^* such that the minimizer of the trust region problem δ^* satisfies $\partial L(\delta^*, \lambda^*)/\partial(\delta, \lambda) = 0$. For $M(\delta) = g^\top\delta - \delta^\top B\delta/2$, this is equivalent to $g - B\delta^* - \lambda^* Q\delta^* = 0$, and thus $(B+\lambda^* Q)\delta^* = g$. If the matrix $B+\lambda^* Q$ is positive definite, then δ^* can be expressed as the unconstrained minimization of $g^\top\delta - \delta^\top(B+\lambda^* I)\delta/2$.

[7] 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.

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, and it can be shown [12] 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 $M(\delta_k^\dagger) \leq \frac{1}{2} M(\delta_k^*)$ where δ_k^* is the optimal solution to the trust-region sub-problem. 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 10.

Another argument against using the Steihaug-Toint method is that if the trust-region is left after only a few steps of CG, it will likely be the case that few, if any, of the low-curvature directions have converged to a significant degree (see section 9). And while we know that this will not affect the optimized value of M by more than a factor of 2, it will nonetheless produce a qualitatively different type of update than one produced using a penalty method, which will have a possibly negative effect on the overall optimization trajectory.

Another possible argument against using the Steihaug-Toint method is that it cannot 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.

8.7 CG truncation as damping

Within HF, the main reason for terminating CG before it has converged is one of a practical nature: 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: CG truncation may be viewed as 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 $M(\delta)$ tends to go down, even while $f(\theta_{k-1} + \delta)$ may still be improving. One way to explain this, assuming a zero initialization, is that the norm of δ will increase monotonically with each step, and thus be more likely to leave the implicit region around $\delta = 0$ where M is a reasonable

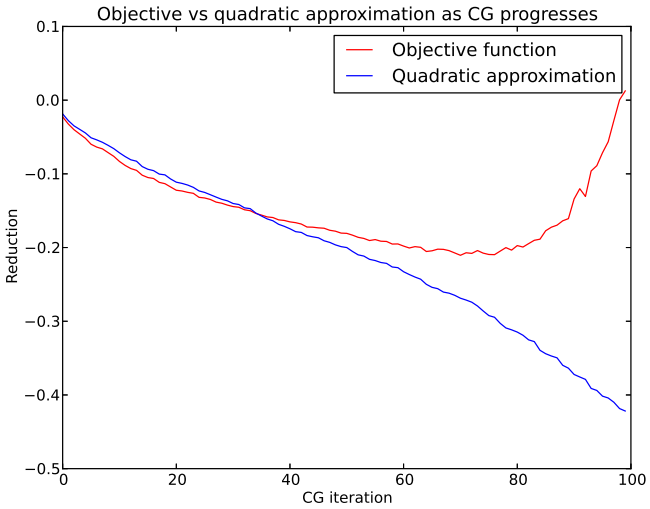

Figure 4: A plot of the objective function (red) versus the quadratic approximation (blue) as a function of the number of CG steps. This was selected as a typical example of a single run of CG performed by HF on a typical deep auto-encoder training task.

approximation of f. 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 9). 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 8.6 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 [13].

Although it wasn't emphasize this in the paper, Martens [22] 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 [16], and that it plays an important role which is not limited to saving computation time. Instead, we can observe that the improvement in the value of the objective f is non-monotonic in the number of CG steps, and may peak long before condition 2 is satisfied (see fig. 4).

Martens [22] 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 f. One possible approach which is reasonably efficient is to compute the objective f only on a small subset of the current minibatch or training set, and only at every multiple of c steps for some fixed c, or every power of ν steps (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 11), and on the particular manner in which CG is initialized (as discussed in sec. 10). In the extreme case where $P = \mathrm{B}$, the eigenvalues will all be equal and CG will converge in one step, rendering CG truncation trivial/useless. And for "stronger" preconditioners that let CG converge faster it can be argued that truncation at a particular iteration i will have a smaller effect than with "weaker" preconditioners. But this perspective oversimplies 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 11.1). This in turn affects the "order" in which CG tends to prioritize convergence along different directions (see section 9). 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 the modify definition of the trust region appropriately.

8.8 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 so that $\theta_k = \theta_{k-1} + \alpha\delta_k$. Usually α is set to 1 as long as certain conditions hold (see Nocedal et al. [30]), 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 ($\delta_k^\top \nabla f(\theta_{k-1}) < 0$) we know that for a sufficiently small (but non-zero) value of α, we will have:

$$f(\theta_{k-1}) > f(\theta_k) = f(\theta_{k-1} + \alpha\delta_k) \tag{11}$$

δ_k will be descent direction as long as $\hat{\mathrm{B}}_{k-1}$ is positive definite, ∇f 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 f, although it may be very small. And if we are content with the weaker condition that only the terms of f corresponding to the minibatches used to estimate $\nabla f(\theta_{k-1})$ must decrease, then we can drop the requirement that ∇f 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

$$f(\theta_{k-1} + \alpha\delta_k) \leq f(\theta_{k-1}) + c\alpha\nabla f(\theta_{k-1})^\top \delta_k$$

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 [8].

Unlike 1st-order methods where the total number of updates to θ can often be on the order of $10^4 - 10^6$ 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 of the form $\frac{1}{2\alpha}\mathrm{B}$. 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 $\vec{0}$.

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., very often chooses an α strictly less than 1) there are two probable causes, which should be addressed directly instead of by relying on the line-search. The first is poorly designed/inappropriate damping methods and/or poor heuristics for adjusting their non-static meta-parameters. 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 minibatch.

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

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

or more, and a 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 8.1) 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.

9 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 11) and CG truncation damping (see section 8.7).

Before we begin it should be noted that due to inexact computer arithmetic, in practice the conclusions of this analysis (which implicitly assume exact arithmetic) will only hold approximately. 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 exact implementation.

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 $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$. This can be related by to HF by taking $A = \mathrm{B}$ (or $A = \hat{\mathrm{B}}$ in the case of a damped quadratic approximation) and $b = -\nabla f$.

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 a neuron.

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 [30, 33] that:

$$\|x_i - x^*\|_A^2 = \min_p \sum_{j=1}^n \lambda_j p(\lambda_j)^2 \xi_j^2 \tag{12}$$

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 [30]. 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. 12 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 $\alpha \in \mathbb{R}$:

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

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 $\beta_j = v_j^\top (x - x_0)$. Note that the v_j's are also mutually conjugate (which follows from them being both orthonormal and eigenvectors) and that since $q(z + x_0) - q(x_0)$ is quadratic in z with constant term 0 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. 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 size β_j of the contribution of each eigenvector/eigendirection to x, 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 $\omega_j \equiv \frac{\eta_j^2}{\lambda_j}$. The difference between the current value of $q(\beta_j v_j + x_0)$ and its minimum has a particularly nice form:

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

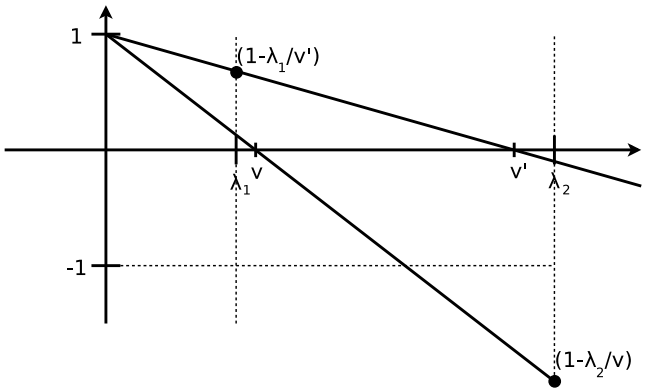

Figure 5: This figure demonstrates geometrically how the contribution to the polynomial $p(z)$ of an additional root ν or ν' in the vicinity of a small eigenvalue λ_1 or a large eigenvalue λ_2 (resp.) affects the loss term associated with the other eigenvalue. In particular, the distance of the lines 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:

$$\begin{aligned} q(x) - q(x^*) = (q(x) - q(x_0)) - (q(x^*) - q(x_0)) &= \sum_{j=1}^{n} (q(\beta_j v_j + x_0) - q(\beta_j^* v_j + x_0)) \\ &= \sum_{j=1}^{n} \omega_j p(\lambda_j)^2 \end{aligned} \tag{13}$$

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 p with the requirement that p has degree i with constant coefficient 1, or in other words:

$$q(x_i) - q(x^*) = \min_p \sum_{j=1}^{n} \omega_j p(\lambda_j)^2 \tag{14}$$

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 prove results about how the distribution of the eigenvalues of A determine how quickly CG can make progress optimizing q. 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 [30].

However, the particular form of eqn. 14 and its derivation allow us to paint a more intuitive picture of how CG operates. Each of the terms in the sum 13 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 f when $x_0 = 0$), and 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 for 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 $\left(1 - \frac{\lambda_j}{\nu_k}\right)^2 \gg 1$ for such λ_j's, 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.

10 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 the performance of CG, 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, depending on how the eigenvalues and associated weights are distributed.

Since the local geometry of the error surface of f (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 [22].

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 have small and spread-out eigenvalues (i.e. curvatures), and meanwhile less converged than 0 (perhaps severely so) only along eigen-directions which are fewer in number, or have larger or more tightly clustered eigenvalues. If the later 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 former directions will be easier for CG to optimize than the latter, 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 later group of badly initialized but "easy-to-fix" eigen-directions.

We surmise that the choice $x_0 = \delta_{k-1}$ fits into the situation described above, where the later 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[9].

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, 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 while they may have been helpful at the previous θ, are no longer appropriate to follow from the current θ. In particular, x_0 will be quickly "corrected" along turbulent directions of high-curvature, reducing (but not completely eliminating) the need for a decay to help "clean up" these directions.

Our experience suggests that properly tuning the decay constant becomes more important as aggressive CG truncation, 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 later 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

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

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.

11 Preconditioning

As powerful as CG is, there are quadratics optimization problems which can be easily solved using more direct methods, that CG will 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) using a total of $O(in)$ time. Meanwhile, we could easily solve the entire system by straightforward inversion of the diagonal curvature matrix in time $O(n)$. 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 $\hat{M}$[10] according to some linear transformation relatively easy to invert, with the idea that CG will make more rapid progress per iteration optimizing w.r.t. the new parameterization.

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

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

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 alg. 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 $\hat{\mathrm{B}}$) 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 f 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).

11.1 The effects of preconditioning

In section 9, we saw how the eigenvalues of the curvature matrix and the corresponding sizes of the contributions to the initial residual of each eigen-direction

[10] We will use the $\hat{}$ notation for the damped quadratic and associated damped curvature matrix $\hat{\mathrm{B}}$ since this is what CG will actually optimize when used within HF.

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. Because CG is optimal, it will tend to make faster progress along directions whose the eigenvalues are close proximity to many other ones that are associated with directions of high-weight (due to its ability to make progress on many such directions at once when their eigenvalues are closely packed).

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 $C^{-\top}\hat{\mathrm{B}}C^{-1}$, and the associated weights, which depend on the transformed initial residual $C^{-\top}(\hat{\mathrm{B}}x_0 - \nabla f)$. 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 as discussed in section 8.7, the eigenvectors and corresponding eigenvalue distribution will affect the order in which various directions converge, and this will interact in a non-trivial way with CG truncation damping. In particular, certain directions which would otherwise never be touched by CG in the original parameterization, either because their eigenvalues are located far away from any high-weight eigenvalue clusters, or because they have very low curvature (i.e., low eigenvalue), could, within the reparameterization, become part of eigen-directions with the opposite properties, and thus be partially optimized by CG even when it is aggressively truncated.

This is a potential problem, since it is our experience that certain very low curvature directions tend to be highly non-trustworthy for neural network training objectives (in the default parameterization). 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 12.1), or to directions which yield small reductions in q for the current minibatch but large increases on other training data (an issue called "minibatch overfitting", which is discussed in section 12.2).

11.2 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. There will often be a trade-off between the computational efficiency of implementing the preconditioner and its effectiveness, both in terms of how it speeds of convergence of CG, and how it may reduce the effectiveness of CG truncation damping.

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.

An 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, $\hat{\mathrm{B}}$). 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 $\hat{\mathrm{B}}$ may be an easier goal to approach directly. Justifying this idea is the fact that when $P = \hat{\mathrm{B}}$, the preconditioned matrix is I, so CG will converge in one step.

Adopting the perspective that P should approximate $\hat{\mathrm{B}}$, 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 $\beta\hat{\mathrm{B}}$ would be an optimal preconditioner for any $\beta > 0$, it could be considered an arbitrarily poor approximation to $\hat{\mathrm{B}}$ 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 feedforward neural networks, due to how the scale of the gradient and curvature w.r.t. the hidden activities grows or shrinks exponentially as we proceed backwards through the layers [4, 17], and how each parameter is associated with a single layer. 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 [23]. Despite how RNNs can possess per-timestep scale variations analogous to the per-layer scale variations sometimes seen with feedforward nets, 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 $O((k+1)n)$ 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 $d = \mathrm{diag}(\hat{\mathrm{B}})$ (or an approximation of this) is a sensible idea, and forms the basis of the approaches taken by Martens [22] and Chapelle and Erhan [10]. However, it may be beneficial, as Martens [22] found, not to use d directly, but to choose P to be somewhere between $\mathrm{diag}(d)$ 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:

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

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 sec. 8.2, it may better to use $\kappa \text{diag}(s)$ instead of κI.

It is important to emphasize that d should approximate $\text{diag}(\hat{\text{B}})$ and not $\text{diag}(\text{B})$, 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

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

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

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

where we might in fact use the aforementioned methods in order to approximate the diagonal of $\text{B} + \lambda D$ 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. [24, sec. 4]), so approximations must be used. Lecun et al. [2] 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 backpropagation, where k is the number of output units [6]. 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.

11.3 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 [22] selected the Empirical Fisher Information matrix F, which is an approximation to the well-known Fisher information matrix [1] (which is itself related to the generalized Gauss-Newton matrix). The empirical Fisher Information matrix is given by

$$\mathrm{F} \equiv \sum_i \nabla f_i \nabla f_i^\top$$

where ∇f_i is the gradient on case i. However, because of its special low-rank form, its diagonal is readily computable as:

$$\mathrm{diag}(\mathrm{F}) = \sum_i \mathrm{sq}(\nabla f_i)$$

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

Because the ∇f_i's are available from the gradient computation $\nabla f = \sum_i \nabla f_i$ over the minibatch, additionally computing diag(F) over the same minibatch in parallel incurs no extra cost, save for the possible requirement of storing the ∇f_i's, which can be avoided for feedforward networks but not RNNs. Algorithm 11.3 computes the diagonal of the Empirical Fisher Information matrix without the extra storage. In the algorithm, each y_i is a matrix with B columns which represent the activations of a minibatch with B cases, and $\mathrm{sq}(\cdot)$ is the coordinate-wise square. It differs from algorithm 2 only in lines 9 and 10.

Algorithm 6 An algorithm for computing the diagonal $\mathrm{diag}(F)$ of the Empirical Fisher Information matrix of a feedforward neural network (includes the standard forward pass)

1: **input:** y_0, θ mapped to $(W_1, \ldots, W_{\ell-1}, b_1, \ldots, b_{\ell-1})$
2: **for all** i **from** 1 **to** $\ell - 1$ **do**
3: $\quad x_{i+1} \leftarrow W_i y_i + b_i$
4: $\quad y_{i+1} \leftarrow s_{i+1}(x_{i+1})$
5: **end for**
6: $dy_\ell \leftarrow \partial L(y_\ell; t_\ell)/\partial y_\ell$ (t_ℓ is the target)
7: **for** i **from** $\ell - 1$ **downto** 1 **do**
8: $\quad dx_{i+1} \leftarrow dy_{i+1} s'_{i+1}(x_{i+1})$
9: $\quad$ Set entries of $\mathrm{diag}([F])$ corresponding to W_i to be $\mathrm{sq}(dx_{i+1})\mathrm{sq}(y_i)^\top$
10: $\quad$ Set entries of $\mathrm{diag}([F])$ corresponding to b_i to be $\mathrm{sq}(dx_{i+1})\mathbf{1}_B^\top$
11: $\quad dy_i \leftarrow W_i^\top dx_{i+1}$
12: **end for**
13: **output:** $\mathrm{diag}(F)$

In general, it is possible to compute the sum of squares of gradients in a minibatch in parallel without storing the squares of the individual gradients (which is often prohibitively expensive) whenever the computational graph of the gradient makes precisely one additive contribution to every parameter for each case. In this case, it possible to add $[\nabla f_i]_j^2$ to the appropriate entry of

$\text{diag}(F)$ as soon as it is computed, so we need not allocate temporary storage for $[\nabla f_i]_j$ for each i and j (rather, only each j).

However, when the computational graph of the derivative (for a given case i) makes multiple additive contributions to each $[\nabla f_i]_j$, it is necessary to allocate temporary storage for this quantity since we must square its total contribution *before* summing over the cases. Interestingly, this issue does not occur for the RNN's gradient computation, since without the per-i squaring, each contribution to $[\nabla f_i]_j$ can be stored in a single vector for all the i's.

11.4 An unbiased estimator for the diagonal of G

Chapelle and Erhan [10] 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 7 below:

Algorithm 7 Computing an unbiased estimate of the diagonal of the GGN matrix

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

As discussed in section 6, multiplication of an arbitrary $v \in \mathbb{R}^m$ by the Jacobian J of F can be performed efficiently by the usual back-propagation algorithm. The correctness of algorithm 7 is easy to prove:

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

where we have used the identity $\text{sq}(x) = \text{diag}(xx^\top)$.

Martens et al. [24], following the work of Chapelle and Erhan [10], 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 el. [24] also proved that sampling the components of v uniformly from $-1, 1$ will produce lower variance estimates than will be obtained by sampling them from $N(0, 1)$.

Computationally, the method of Chapelle and Erhan is very similar to the method for computing the diagonal of the Empirical Fisher Information that was described in the previous section. Indeed, while the latter compute $\text{sq}\left(J^\top \nabla L\right)$, this method computes $\text{sq}\left(J^\top L''^{1/2} v\right)$ for random v's, and so the methods have similar implementations. In particular, both estimates can be computed in parallel over cases in the minibatch, and they share the issue with temporary stored discussed in the previous section, which can be overcome in for feedforward networks but not RNNs.

In our experience, both methods tend to produce preconditioners with similar properties and performance characteristics, although Chapelle and Erhan [10] 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 matrix, 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 Fisher matrix). The diagonal of the Fisher matrix also has the additional advantage that it can be computed in tandem with the gradient at virtually no extra cost.

12 Minibatching

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 [7]. 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 at all.

"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 "minibatches", as long as the learning rate is sufficiently small. HF, on the other hand, uses minibatches to estimate the curvature matrix, which is used in a very strong way to produce large and sometimes aggressive parameter updates. These curvature matrix estimates may become increasingly low-rank and degenerate as the minibatch size shrinks (assuming no contribution from damping or weight-decay), which in some cases (see subsection 12.1) may lead to unbounded and nonsensical updates, although the damping mechanisms discussed in sec. 8 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 approximated in the curvature

estimated from the minibatch, the update δ obtained from optimizing M could easily make the objective f worse on such a case, perhaps severely so.

Thus 2nd-order methods like HF which must estimate the curvature only from the current minibatch, may not work nearly as well with very small minibatches. And while there are several strategies to deal with minibatches that are "too small", (as we will discuss in subsection 12.2), 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 minibatches (see section 7). For highly parallel architectures like GPUs, these tend to give an increase in computational cost which remains sublinear (as a function of minibatch size) up to and beyond minibatch sizes which are useful in HF.

12.1 Higher quality gradient estimates

Unlike 1st order optimization schemes like SGD where the number of iterations required to approach a good solution can reach as high as $10^5 - 10^7$, the number of iterations required by a strong 2nd-order optimizer like HF is in our experience orders of magnitude smaller, and usually around $10^2 - 10^3$. While the linear term $b = -\nabla f(\theta_{k-1})$ 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 $A = \mathrm{B}$ 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 much larger minibatch than is used to compute the matrix-vector products. Martens [22] recommended using this technique (as does Byrd et al. [8]), 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 minibatch to compute gradient and curvature matrix-vector products. These have been corroborated by our practical experience.

We will refer to the minibatches used to compute the gradient and curvature matrix-vector products as the "gradient minibatch" and "curvature minibatch", respectively.

When the gradient and curvature minibatches 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 f obtained by linearizing F (see eqn. 9), but restricted to the data in the current minibatch. 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:

$$
\begin{aligned}
M(\delta) &= \frac{1}{2}\delta^\top \mathrm{B}\delta + \nabla f_{k-1}^\top \delta + f(\theta_{k-1}) = \frac{1}{2}\delta^\top J^\top L'' J\delta + \delta^\top (J^\top \nabla L) + f(\theta_{k-1}) \\
&= \frac{1}{2}(J\delta)^\top L''(J\delta) + (J\delta)^\top L''L''^{-1}\nabla L + \nabla L^\top L''^{-1} L'' L''^{-1}\nabla L \\
&\qquad\qquad - \nabla L^\top L''^{-1} L'' L''^{-1}\nabla L + f(\theta_{k-1}) \\
&= \frac{1}{2}(J\delta + \nabla L)^\top L''(J\delta + \nabla L)^\top + c \\
&= \frac{1}{2}\|J\delta + L''^{-1}\nabla L\|_{L''}^2 + c
\end{aligned}
$$

where $c = f(\theta_{k-1}) - \nabla L^\top L''^{-1}\nabla L$ and all quantities are computed only on the current minibatch. Here we have used the fact that 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 f that the quadratic model M can ever predict: $\nabla L^\top L''^{-1}\nabla L$, a quantity which does not depend on the properties of the network, only on its current predictions and the associated convex loss function.

Such a boundedness result does not exist whenever M is estimated using different minibatches 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 $d^\top \mathrm{B} d = 0$. 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 $\hat{\mathrm{B}}_k$ 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 the sense that $d^\top \mathrm{B} d$ may be non-zero but extremely small, leading to a nearly degenerate update δ. For example, when using Tikhonov damping then we know that $d^\top \hat{\mathrm{B}}_k d \geq \lambda$, 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 f worse even on cases contained in both the gradient and curvature minibatches, for reasons that have nothing directly to do with a breakdown in the reliability of the quadratic approximation to f. On the other hand, if the curvature and gradient minibatches are equal *and* the quadratic approximation to f is otherwise reliable (or properly damped), then using equal gradient and

curvature minibatches provides a minimal guarantee that f will improve on the current minibatch after the proposed update is applied.

Another more subtle way in which using a smaller-sized curvature minibatch than gradient minibatch 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-minibatch. This is because when we compute estimates of the curvature by averaging, we must divide by the number of cases in the minibatch, 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. [8] 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. $\hat{\mathrm{B}} - \mu I$ is PSD for all possible $\hat{\mathrm{B}}$'s which we might produce, then assuming the use of a basic line-search and other mild conditions, an 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 is as a result very weak, saying nothing about the rate of convergence, or how small the updates will have to be. As far as 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 f at all.

Despite all of these objections, the higher quality estimates of the gradient may nonetheless provide superior convergence properties in some 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 training dataset, or the network's responses to it, are particular "diverse".

12.2 Minibatch overfitting and methods to combat it

As mentioned in the previous section, the updates produced by HF may be effectively "overfit" to the current minibatch 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 minibatch. 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 a stochastic method.

The simplest solution to dealing with the minibatch overfitting problem is

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

The damping methods discussed in section 8 were developed to compensate for untrustworthiness of the local quadratic approximations M being made to f, which exists due to the simple fact that f is not actually a convex quadratic, and so M may fail to be a sensible approximation to f at its minimum δ^*. These methods work by imposing various soft or hard constraints on the update δ in order to keep it "closer" to 0 (where M trustworthy by construction), according to some metric.

Using minibatches 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 these, 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 8.3) 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 minibatch overfitting, this will make HF behave asymptotically like SGD with a dynamically adjusted learning rate.

There is a compelling analogy between the minibatch 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 L2 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 a method 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 super-linear terms in the Taylor series expansion of f, 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 minibatch-related estimation errors with update damping methods. Indeed, the negative gradient computed on a given minibatch 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 minibatch may actually make the total objective worse.

Thus, when tuning damping mechanisms to handle minibatch overfitting (either by hand, or dynamically using an automatic heuristic), one shouldn't

aim for obtaining a reduction in the total f that is a fixed multiple of that which is predicted by the minibatch-computed M (as is done in section 8.5), but rather to simply obtain a more modest reduction which is proportional to the relative contribution of the current minibatch 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 10) seems to bias CG towards finding solutions that generalize better to data outside of the current minibatch. 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[11] 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 minibatch and quickly zeroed out by CG before ever having a chance grow large in δ.

Finally, Byrd et al. [9] has developed methods to deal with the minibatch overfitting problem, which are based on heuristics that increase the minibatch 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 minibatch, which is only one method to deal with minibatch overfitting, is not a strategy which will remain practical for very long. Thirdly, aggressive early termination heuristics for CG, similar to this one, tend to interact badly with non-zero CG initializations[12] and other forms of damping. And finally, there are other more direct ways of measuring how well updates will generalize, such as simply monitoring f on some training data outside of the current minibatch.

13 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

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

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

- the CG initialization technique described in section 10
- 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 11.2 with $\kappa = 0$ and $\psi = 0.75$. When using one of the scale-sensitive methods described in section 8.3, 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 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 LM heuristic or a similar method

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

First, there is the original algorithm described in [22], which works pretty well. Here, the "CG-backtracking" approach is used to select an iterate based on the objective function value (see section 8.7), 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 particular, CG is terminated as soon as the objective function, as evaluated on the data in the "curvature minibatch" (see section 12.1) 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 f) as evaluated on some data which is *not* contained in the curvature minibatch. 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 minibatch (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 10) to better distribute work involved in

optimizing the local quadratics across many iterations. To do this, we use a smaller maximum number of CG steps (around 25 to 50), smaller minibatches 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 minibatch overfitting, and makes it feasible to use smaller minibatches and also compute the gradient and curvature using the same data. And as discussed in section 12.1, 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 12.2, such as lowering the learning rate, using shorter CG runs, increasing the minibatch size, and/or switching back to using larger minibatches 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 in [22], and the method of Glorot & Bengio [11], and of course the pre-training techniques pioneered in [16].

14 Summary

We described various components of the Hessian-free optimization, how they can interact non-trivially, and how their effectiveness (or possibly harmfulness) is situation dependent. The main points to keep in mind are:

- for non-convex optimizations it is usually preferable to use the generalized Gauss-Newton matrix which is guaranteed to be PSD
- updates must be "damped" due to the untrustworthiness of the quadratic model
- there are various damping techniques that can be used, and their effectiveness depends highly on f and how it is parameterized
- truncating CG before convergence, in addition to making HF practical, can also provide a beneficial damping effect
- the strategy for terminating CG is usually a combination of progress-based heuristic and a hard-limit anywhere between 10 and 300+ (which should be considered an important meta-parameter)

- preconditioning can sometimes enable CG to make more rapid progress per step, but only if used correctly
- simple diagonal preconditioning methods tend to work well for feedforward nets but not for RNNs
- preconditioning interacts non-trivially with certain forms of damping such as CG truncation, which must be kept in mind
- initializing CG from the update computed by the previous run of CG can have a beneficial "momentum-type effect"
- HF tends to require much larger minibatches than are used in SGD
- minibatch-overfitting can cause HF's update proposals to be poor even for δ's where the quadratic model is trustworthy
- using more data to compute the gradients than the curvature matrix-vector products is a low-cost method of potentially increasing the quality of the updates, but it can sometimes do more harm than good
- minibatch-overfitting can also be combated using some of the standard damping methods, along with simply increasing the minibatch size
- structural damping works well for training RNNs, particularly on problems with pathological long-range dependencies
- exploiting data-parallelism is very important for producing an efficient implementation
- correctness of curvature matrix-vector products should be checked using finite difference methods
- the extra memory costs associated with the parallel computation of gradients and curvature matrix-vector products can be mitigated

The difficulty of customizing an HF approach for particular application will no doubt depend on the specifics of the model and the dataset. While in many cases a generic approach can be used to good effect, some more difficult problems like RNNs or feedforward networks with non-standard parameterizations may require additional care. And even on "easier" problems, a better designed approach may allow one to surpass performance barriers that may have previously been mistaken for convergence.

This report has described many of the tricks and ideas, along with their theoretical justifications, which may useful in this regard. While we can try to predict what combination of ideas will work best for a given problem, based on previous experience and/or mathematical/intuitive reasoning, the only way to be sure is with careful experimentation. Unfortunately, optimization theory has a long way to go before being able to predict the performance of a method like HF applied to the highly non-convex objectives functions associated with neural networks.

References

[1] S.I. Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276, 1998.

[2] S. Becker and Y. Le Cun. Improving the convergence of back-propagation learning with second order methods. In *Proceedings of the 1988 connectionist models summer school*, pages 29–37. San Matteo, CA: Morgan Kaufmann, 1988.

[3] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153, 2007.

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

[5] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, volume 4, 2010.

[6] C. Bishop. Exact calculation of the hessian matrix for the multilayer perceptron. *Neural Computation*, 4(4):494–501, 1992.

[7] L. Bottou and O. Bousquet. The tradeoffs of large scale learning. *Advances in neural information processing systems*, 20:161–168, 2008.

[8] R.H. Byrd, G.M. Chin, W. Neveitt, and J. Nocedal. On the use of stochastic hessian information in optimization methods for machine learning. *SIAM Journal on Optimization*, 21:977, 2011.

[9] R.H. Byrd, G.M. Chin, J. Nocedal, and Y. Wu. Sample size selection in optimization methods for machine learning. *Mathematical Programming*, pages 1–29, 2012.

[10] O. Chapelle and D. Erhan. Improved preconditioner for hessian free optimization. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[11] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10). Society for Artificial Intelligence and Statistics*, 2010.

[12] N.I.M. Gould, S. Lucidi, M. Roma, and P.L. Toint. Solving the trust-region subproblem using the lanczos method. *SIAM Journal on Optimization*, 9(2):504–525, 1999.

[13] P.C. Hansen and D.P. OLeary. The use of the l-curve in the regularization of discrete ill-posed problems. *SIAM Journal on Scientific Computing*, 14:1487, 1993.

[14] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems, 1952.

[15] G.E. Hinton, S. Osindero, and Y.W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[16] G.E. Hinton and R.R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

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

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

[19] H. Jaeger and H. Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667):78–80, 2004.

[20] Y. LeCun, L. Bottou, G. Orr, and K. Müller. Efficient backprop. *Neural networks: Tricks of the trade*, pages 546–546, 1998.

[21] Y. LeCun, L. Bottou, G.B. Orr, and K.R. Muller. Neural networks-tricks of the trade. *Springer Lecture Notes in Computer Sciences*, 1524:5–50, 1998.

[22] J. Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, volume 951, page 2010, 2010.

[23] J. Martens and I. Sutskever. Learning recurrent neural networks with hessian-free optimization. In *Proc. ICML*, 2011.

[24] J. Martens, I. Sutskever, and K. Swersky. Estimating the hessian by backpropagating curvature. In *Proc. ICML*, 2012.

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

[26] J.J. Moré and D.C. Sorensen. Computing a trust region step. *SIAM Journal on Scientific and Statistical Computing*, 4:553, 1983.

[27] S.G. Nash. Newton-type minimization via the lanczos method. *SIAM Journal on Numerical Analysis*, pages 770–788, 1984.

[28] S.G. Nash. A survey of truncated-newton methods. *Journal of Computational and Applied Mathematics*, 124(1):45–59, 2000.

[29] Y. Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/k2). In *Doklady AN SSSR*, volume 269, pages 543–547, 1983.

[30] J. Nocedal and S.J. Wright. *Numerical optimization*. Springer verlag, 1999.

[31] B.A. Pearlmutter. Fast exact multiplication by the hessian. *Neural Computation*, 6(1):147–160, 1994.

[32] Nicol N. Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, 14:2002, 2002.

[33] J.R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[34] O. Vinyals and D. Povey. Krylov subspace descent for deep learning. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2012.

[35] RE Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.