Fast Curvature Matrix-Vector Products

Nicol N. Schraudolph

Institute of Computational Sciences, Eidgenössische Technische Hochschule, CH-8092 Zürich, Switzerland
nic@inf.ethz.ch

Abstract. The Gauss-Newton approximation of the Hessian guarantees positive semi-definiteness while retaining more second-order information than the Fisher information. We extend it from nonlinear least squares to all differentiable objectives such that positive semi-definiteness is maintained for the standard loss functions in neural network regression and classification. We give efficient algorithms for computing the product of extended Gauss-Newton and Fisher information matrices with arbitrary vectors, using techniques similar to but even cheaper than the fast Hessian-vector product [1]. The stability of SMD [2–5], a learning rate adaptation method that uses curvature matrix-vector products, improves when the extended Gauss-Newton matrix is substituted for the Hessian.

1 Definitions and Notation

Network. A neural network with m inputs, n weights, and o linear outputs is usually regarded as a mapping $\mathbb{R}^m \to \mathbb{R}^o$ from an input pattern $\boldsymbol{x}$ to the corresponding output $\boldsymbol{y}$, for a given vector $\boldsymbol{w}$ of weights. Here we formalize such a network instead as a mapping $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^o$ from weights to outputs (for given inputs), and write $\boldsymbol{y} = \mathcal{N}(\boldsymbol{w})$. To extend this formalism to networks with nonlinear outputs, we define the output nonlinearity $\mathcal{M} : \mathbb{R}^o \to \mathbb{R}^o$ and write $\boldsymbol{z} = \mathcal{M}(\boldsymbol{y}) = \mathcal{M}(\mathcal{N}(\boldsymbol{w}))$. For networks with linear outputs, $\mathcal{M}$ is the identity.

Loss function. We consider neural network learning as the minimization of a scalar loss function $\mathcal{L} : \mathbb{R}^o \to \mathbb{R}$ defined as the log-likelihood $\mathcal{L}(\boldsymbol{z}) \equiv -\log \Pr(\boldsymbol{z})$ of the network output $\boldsymbol{z}$ under a suitable statistical model [6]. For supervised learning, $\mathcal{L}$ may also implicitly depend on given targets $\boldsymbol{z}^*$ for the network outputs. Formally, the loss can now be regarded as a function $\mathcal{L}(\mathcal{M}(\mathcal{N}(\boldsymbol{w})))$ of the weights, for a given set of inputs and (if supervised) targets.

Jacobian. The Jacobian $J_\mathcal{F}$ of a function $\mathcal{F} : \mathbb{R}^m \to \mathbb{R}^n$ is the $n \times m$ matrix of partial derivatives of the outputs of $\mathcal{F}$ with respect to its inputs. For a neural network defined as above, the gradient of the loss with respect to the weights is given by

$$\frac{\partial}{\partial \boldsymbol{w}} \mathcal{L}(\mathcal{M}(\mathcal{N}(\boldsymbol{w}))) \;=\; J'_{\mathcal{L}\circ\mathcal{M}\circ\mathcal{N}} \;=\; J'_\mathcal{N} J'_\mathcal{M} J'_\mathcal{L}\,, \qquad (1)$$

where $\circ$ denotes function composition, and $'$ the matrix transpose. We use J as an abbreviation for $J_{\mathcal{L}\circ\mathcal{M}\circ\mathcal{N}}$.

Matching loss functions. We say that the loss function $\mathcal{L}$ *matches* the output nonlinearity $\mathcal{M}$ iff $J'_{\mathcal{L}\circ\mathcal{M}} = A\boldsymbol{z} + \boldsymbol{b}$, for some A and $\boldsymbol{b}$ not dependent on $\boldsymbol{w}$. The standard loss functions used in neural network regression and classification — sum-squared error for linear outputs, and cross-entropy error for softmax or logistic outputs — are all matching loss functions with $A = I$ and $\boldsymbol{b} = -\boldsymbol{z}^*$, so that $J'_{\mathcal{L}\circ\mathcal{M}} = \boldsymbol{z} - \boldsymbol{z}^*$ [6, chapter 6]. This will simplify some of the calculations described in Section 3 below.

Hessian. The instantaneous Hessian $H_\mathcal{F}$ of a scalar function $\mathcal{F} : \mathbb{R}^n \to \mathbb{R}$ is the $n \times n$ matrix of second derivatives of $\mathcal{F}(\boldsymbol{w})$ with respect to its inputs $\boldsymbol{w}$:

$$H_\mathcal{F} \;\equiv\; \frac{\partial J_\mathcal{F}}{\partial \boldsymbol{w}'}\,, \quad i.e., \quad (H_\mathcal{F})_{ij} \;=\; \frac{\partial^2 \mathcal{F}(\boldsymbol{w})}{\partial w_i \, \partial w_j}\,. \tag{2}$$

For a neural network as defined above, we abbreviate $H \equiv H_{\mathcal{L}\circ\mathcal{M}\circ\mathcal{N}}$. The Hessian proper, which we denote $\bar{H}$, is obtained by taking the expectation of H over inputs: $\bar{H} \equiv \langle H \rangle_{\boldsymbol{x}}$. For matching loss functions, $H_{\mathcal{L}\circ\mathcal{M}} = AJ_\mathcal{M} = J'_\mathcal{M} A'$.

Fisher information. The instantaneous Fisher information matrix $F_\mathcal{F}$ of a scalar log-likelihood function $\mathcal{F} : \mathbb{R}^n \to \mathbb{R}$ is the $n \times n$ matrix formed by the outer product of its first derivatives:

$$F_\mathcal{F} \;\equiv\; J'_\mathcal{F} J_\mathcal{F}\,, \quad i.e., \quad (F_\mathcal{F})_{ij} \;=\; \frac{\partial \mathcal{F}(\boldsymbol{w})}{\partial w_i} \, \frac{\partial \mathcal{F}(\boldsymbol{w})}{\partial w_j}\,. \tag{3}$$

Note that $F_\mathcal{F}$ always has rank one. As before, we abbreviate $F \equiv F_{\mathcal{L}\circ\mathcal{M}\circ\mathcal{N}}$. The Fisher information matrix proper, $\bar{F} \equiv \langle F \rangle_{\boldsymbol{x}}$, describes the geometric structure of weight space [7] and is used in the *natural gradient* descent approach [8].

2 Extended Gauss-Newton Approximation

Problems with the Hessian. The use of the Hessian in second-order gradient descent for neural networks is problematic: for nonlinear systems, $\bar{H}$ is not necessarily positive definite, so Newton's method may diverge, or even take steps in uphill directions. Practical second-order gradient methods should therefore use approximations or modifications of the Hessian that are known to be reasonably well-behaved, with positive semi-definiteness as a minimum requirement.

Fisher information. One alternative that has been proposed is the Fisher information matrix $\bar{F}$ [8], which — being a quadratic form — is positive semi-definite by definition. On the other hand, $\bar{F}$ ignores *all* second-order interactions, thus throwing away a lot of potentially useful information. By contrast, we shall derive an approximation of the Hessian that is positive semi-definite even though it does retain certain second-order terms.

Gauss-Newton. An entire class of popular optimization techniques for nonlinear least squares problems — as implemented by neural networks with linear outputs and sum-squared loss function — is based on the well-known Gauss-Newton (*aka* "linearized", "outer product", or "squared Jacobian") approximation of the Hessian. Here we extend the Gauss-Newton approach to other standard loss functions — in particular, the cross-entropy loss used in neural network classification — in such a way that even though some second-order information is retained, positive semi-definiteness can still be proven. Using the product rule, the instantaneous Hessian of our neural network can be written as

$$H \;=\; \frac{\partial}{\partial \boldsymbol{w}'} (J_{\mathcal{L}\circ\mathcal{M}}\, J_{\mathcal{N}}) \;=\; J_{\mathcal{N}}'\, H_{\mathcal{L}\circ\mathcal{M}}\, J_{\mathcal{N}} \;+\; \sum_{i=1}^{o} (J_{\mathcal{L}\circ\mathcal{M}})_i\, H_{\mathcal{N}_i}\,, \tag{4}$$

where i ranges over the o outputs of $\mathcal{N}$ (the network proper), with $\mathcal{N}_i$ denoting the subnetwork that produces the ith output. Ignoring the second term above, we define the extended, instantaneous Gauss-Newton matrix

$$G \;\equiv\; J_{\mathcal{N}}'\, H_{\mathcal{L}\circ\mathcal{M}}\, J_{\mathcal{N}}\,. \tag{5}$$

Note that G has rank $\leq o$ (the number of network outputs), and is positive semi-definite, regardless of the choice of network $\mathcal{N}$, provided that $H_{\mathcal{L}\circ\mathcal{M}}$ is.

G models the second-order interactions among the network outputs (via $H_{\mathcal{L}\circ\mathcal{M}}$) while ignoring those arising within the network itself ($H_{\mathcal{N}_i}$). This constitutes a compromise between the Hessian (which models all second-order interactions) and the Fisher information (which ignores them all). For systems with a single, linear output and sum-squared error, G reduces to F; in all other cases it provides a richer source of curvature information.

Standard loss functions. For the standard loss functions used in neural network regression and classification, G has additional interesting properties:

Firstly, the residual $J_{\mathcal{L}\circ\mathcal{M}}' = \boldsymbol{z} - \boldsymbol{z}^*$ vanishes at the optimum for realizable problems, so that the Gauss-Newton approximation (5) of the Hessian (4) becomes exact in this case. For unrealizable problems, the residuals at the optimum have zero mean; this will tend to make the last term in (4) vanish in expectation, so that we can still assume $\bar{G} \approx \bar{H}$ near the optimum.

Secondly, in each case we can show that $H_{\mathcal{L}\circ\mathcal{M}}$ (and hence G, and hence $\bar{G}$) is positive semi-definite: for linear outputs with sum-squared loss — *i.e.,* conventional Gauss-Newton — $H_{\mathcal{L}\circ\mathcal{M}} = J_{\mathcal{M}}$ is just the identity I; for independent logistic outputs with cross-entropy loss it is $\mathrm{diag}[\boldsymbol{z}\,(1-\boldsymbol{z})]$, positive semi-definite because $(\forall i)\; 0 < z_i < 1$. For softmax output with cross-entropy loss we have $H_{\mathcal{L}\circ\mathcal{M}} = \mathrm{diag}(\boldsymbol{z}) - \boldsymbol{z}\boldsymbol{z}'$, which is also positive semi-definite since $(\forall i)\; z_i > 0$ and $\sum_i z_i = 1$, and thus

$$\begin{aligned}
(\forall \boldsymbol{v} \in \mathbb{R}^o)\;\; \boldsymbol{v}'[\mathrm{diag}(\boldsymbol{z}) - \boldsymbol{z}\boldsymbol{z}']\,\boldsymbol{v} \;\; &= \sum_i z_i v_i^2 \;-\; (\sum_i z_i v_i)^2 \\
&= \sum_i z_i v_i^2 \;-\; 2\,(\sum_i z_i v_i)(\sum_j z_j v_j) \;+\; (\sum_j z_j v_j)^2 \\
&= \sum_i z_i (v_i - \sum_j z_j v_j)^2 \;\geq\; 0\,.
\end{aligned} \tag{6}$$

3 Fast Curvature Matrix-Vector Products

3.1 The Passes

We now describe algorithms that compute the product of F, G, or H with an arbitrary n-dimensional vector $\boldsymbol{v}$ in $O(n)$. They are all constructed from the same set of passes in which certain quantities are propagated through the network in either forward or reverse direction. For implementation purposes it should be noted that *automatic differentiation* software tools[1] can automatically produce these passes from a program implementing the basic forward pass f_0.

$\boldsymbol{f_0}$. This is the ordinary forward pass of a neural network, evaluating the function $\mathcal{F}(\boldsymbol{w})$ it implements by propagating activity forward through $\mathcal{F}$.

$\boldsymbol{r_1}$. The ordinary backward pass of a neural network, calculating $J'_{\mathcal{F}}\boldsymbol{u}$ by propagating $\boldsymbol{u}$ backwards through $\mathcal{F}$. Uses intermediate results from the f_0 pass.

$\boldsymbol{f_1}$. Following Pearlmutter [1], we define the *Gateaux* derivative

$$\mathcal{R}_{\boldsymbol{v}}(\mathcal{F}(\boldsymbol{w})) \equiv \left.\frac{\partial \mathcal{F}(\boldsymbol{w} + r\boldsymbol{v})}{\partial r}\right|_{r=0} = J_{\mathcal{F}}\,\boldsymbol{v} \tag{7}$$

which describes the effect on a function $\mathcal{F}(\boldsymbol{w})$ of a weight perturbation in the direction of $\boldsymbol{v}$. By pushing $\mathcal{R}_{\boldsymbol{v}}$ — which obeys the usual rules for differential operators — down into the equations of the forward pass f_0, one obtains an efficient procedure which calculates $J_{\mathcal{F}}\boldsymbol{v}$ from $\boldsymbol{v}$; see [1] for details and examples. This f_1 pass uses intermediate results from the f_0 pass.

$\boldsymbol{r_2}$. When the $\mathcal{R}_{\boldsymbol{v}}$ operator is applied to the r_1 pass for a scalar function $\mathcal{F}$, one obtains an efficient procedure for calculating the Hessian-vector product $H_{\mathcal{F}}\,\boldsymbol{v} = \mathcal{R}_{\boldsymbol{v}}(J'_{\mathcal{F}})$. Again, see [1] for details and examples. This r_2 pass uses intermediate results from the f_0, f_1, and r_1 passes.

3.2 The Algorithms

The first step in all three matrix-vector products is the computation of the gradient J' of our neural network model by standard backpropagation:

Gradient. J' is computed by an f_0 pass through the entire network ($\mathcal{N}$, $\mathcal{M}$, and $\mathcal{L}$), followed by an r_1 pass propagating $\boldsymbol{u} = 1$ back through the entire network ($\mathcal{L}$, $\mathcal{M}$, then $\mathcal{N}$). For matching loss functions there is a shortcut: since $J'_{\mathcal{L}\circ\mathcal{M}} = A\boldsymbol{z} + \boldsymbol{b}$, we can limit the forward pass to $\mathcal{N}$ and $\mathcal{M}$ (to compute $\boldsymbol{z}$), then r_1-propagate $\boldsymbol{u} = A\boldsymbol{z} + \boldsymbol{b}$ back through just $\mathcal{N}$.

Fisher information. To compute $F\boldsymbol{v} = J'J\boldsymbol{v}$, simply multiply the gradient J' by the inner product between J' and $\boldsymbol{v}$. If there is no random access to J' or $\boldsymbol{v}$ — *i.e.,* its elements can be accessed only through passes like the above — the

[1] See `http://www-unix.mcs.anl.gov/autodiff/`

scalar $J\boldsymbol{v}$ can instead be calculated by f_1-propagating $\boldsymbol{v}$ forward through the network. This step is also necessary for the other two matrix-vector products.

Hessian. After f_1-propagating $\boldsymbol{v}$ forward, r_2-propagate $\mathcal{R}_{\boldsymbol{v}}(1)=0$ back through the entire network to obtain $H\boldsymbol{v} = \mathcal{R}_{\boldsymbol{v}}(J')$ [1]. For matching loss functions, the shortcut is to f_1-propagate $\boldsymbol{v}$ through just $\mathcal{N}$ and $\mathcal{M}$ to obtain $\mathcal{R}_{\boldsymbol{v}}(\boldsymbol{z})$, then r_2-propagate $\mathcal{R}_{\boldsymbol{v}}(J'_{\mathcal{L}\circ\mathcal{M}}) = A\,\mathcal{R}_{\boldsymbol{v}}(\boldsymbol{z})$ back through $\mathcal{N}$.

Gauss-Newton. Following the f_1 pass, r_2-propagate $\mathcal{R}_{\boldsymbol{v}}(1) = 0$ back through $\mathcal{L}$ and $\mathcal{M}$ to obtain $\mathcal{R}_{\boldsymbol{v}}(J'_{\mathcal{L}\circ\mathcal{M}}) = H_{\mathcal{L}\circ\mathcal{M}} J_{\mathcal{N}} \boldsymbol{v}$, then r_1-propagate that back through $\mathcal{N}$, giving $G\boldsymbol{v}$. For matching loss functions we do not require an r_2 pass: since

$$G \;=\; J'_{\mathcal{N}} H_{\mathcal{L}\circ\mathcal{M}} J_{\mathcal{N}} \;=\; J'_{\mathcal{N}} J'_{\mathcal{M}} A' J_{\mathcal{N}}\,, \tag{8}$$

we can limit the f_1 pass to $\mathcal{N}$, multiply the result with A', then r_1-propagate it back through $\mathcal{M}$ and $\mathcal{N}$. Alternatively, one may compute the equivalent $G\boldsymbol{v} = J'_{\mathcal{N}} A J_{\mathcal{M}} J_{\mathcal{N}} \boldsymbol{v}$ by continuing the f_1 pass through $\mathcal{M}$, multiplying with A, and r_1-propagating back through $\mathcal{N}$.

Batch average. To calculate the product of a curvature matrix $\bar{C} \equiv \langle C \rangle_{\boldsymbol{x}}$ — where C is one of F, G, or H — with vector $\boldsymbol{v}$, average the instantaneous product $C\boldsymbol{v}$ over all input patterns $\boldsymbol{x}$ (and associated targets $\boldsymbol{z}^*$, if applicable) while holding $\boldsymbol{v}$ constant. For large training sets, or non-stationary streams of data, it is often preferable to estimate $\bar{C}\boldsymbol{v}$ by averaging over "mini-batches" of (typically) just 5–50 patterns.

3.3 Computational Cost

Table 1 summarizes the curvature matrix C corresponding to various gradient methods, the passes needed (for a matching loss function) to calculate both the gradient $\bar{J}' \equiv \langle J' \rangle_{\boldsymbol{x}}$ and the fast matrix-vector product $\bar{C}\boldsymbol{v}$, and the associated computational cost in terms of floating-point operations (flops) per weight and pattern in a multi-layer perceptron. These figures ignore certain optimizations — *e.g.*, not propagating gradients back to the inputs — and assume that any computation at the network's nodes is dwarfed by that required for the weights.

		Pass	f_0	r_1	f_1	r_2	**Cost**
Method		result:	$\mathcal{L}$	$J'\boldsymbol{u}$	$J\boldsymbol{v}$	$H\boldsymbol{v}$	(for $\bar{J}'$
$C =$	name	cost:	2	3	4	7	& $\bar{C}\boldsymbol{v}$)
I	steepest descent		√	√			6
F	natural gradient		√	√			10
G	Gauss-Newton		√	√√	√		14
H	Newton's method		√	√	√	√	18

Table 1. Passes needed to compute gradient $\bar{J}'$ and fast matrix-vector product $\bar{C}\boldsymbol{v}$, and associated cost (for a multi-layer perceptron) in flops per weight and pattern, for various choices of curvature matrix C.

4 Application to Stochastic Meta-Descent (SMD)

Algorithm. SMD [2–5] is a new, highly effective online algorithm for local learning rate adaptation. It updates the weights $\boldsymbol{w}$ by the simple gradient descent

$$\boldsymbol{w}_{t+1} \;=\; \boldsymbol{w}_t \;-\; \boldsymbol{p}_t \cdot J', \tag{9}$$

where $\cdot$ denotes element-wise multiplication, and J' the stochastic gradient. The vector $\boldsymbol{p}$ of local learning rates is adapted multiplicatively:

$$\boldsymbol{p}_t \;=\; \boldsymbol{p}_{t-1} \cdot \max(\tfrac{1}{2},\; 1 + \mu\, \boldsymbol{v}_t \cdot J')\,, \tag{10}$$

using a scalar meta-learning rate μ. This update minimizes the network's loss with respect to $\boldsymbol{p}$ by *exponentiated* gradient descent [9], but has been re-linearized so as to avoid the computationally expensive exponentiation operation [10]. The auxiliary vector $\boldsymbol{v}$ used in (10) is itself updated iteratively via

$$\boldsymbol{v}_{t+1} \;=\; \lambda\, \boldsymbol{v}_t \;+\; \boldsymbol{p}_t \cdot (J' \;-\; \lambda\, C \boldsymbol{v}_t)\,, \tag{11}$$

where C is the curvature matrix, and $0 \leq \lambda \leq 1$ a forgetting factor for nonstationary tasks. $C\boldsymbol{v}_t$ is computed via the fast algorithms described above.

Benchmark setup. We illustrate the behavior of SMD on the "four regions" benchmark [11]: a fully connected feedforward network $\mathcal{N}$ with two hidden layers of 10 tanh units each (Fig. 1, right) is to classify two continuous inputs in the range [-1,1] into four disjoint, non-convex regions (Fig. 1, left). We use the standard softmax output nonlinearity $\mathcal{M}$ with matching cross-entropy loss $\mathcal{L}$, meta-learning rate $\mu = 0.05$, initial learning rates $\boldsymbol{p}_0 = 0.1$, and uniformly random initial weights in the range [-0.3,0.3]. Training patterns are generated online by drawing independent, uniformly random input samples; they are presented in mini-batches of 10 patterns each. Since each pattern is seen only once, the empirical loss provides an unbiased estimate of generalization ability.

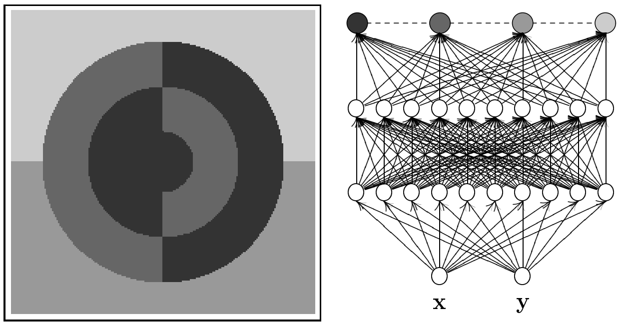

Fig. 1. The four regions task (left), and the network we trained on it (right).

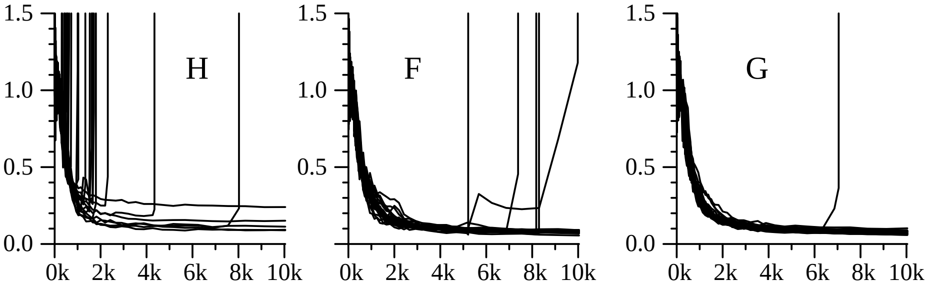

Fig. 2. Loss curves for 25 runs of SMD with $\lambda = 1$, when using the Hessian (left), the Fisher information (center), or the extended Gauss-Newton matrix (right) for C in Equation (11). Vertical spikes indicate divergence.

Curvature matrix. Fig. 2 shows loss curves for SMD with $\lambda = 1$ on the four regions problem, starting from 25 different random initial states, using the Hessian, Fisher information, and extended Gauss-Newton matrix, respectively, for C in Equation (11). With the Hessian (left), 80% of the runs diverge — most of them early on, when the risk that H is not positive definite is greatest. When we guarantee positive semi-definiteness by switching to the Fisher information matrix (center), the proportion of diverged runs drops to 20%; those runs that still diverge do so only relatively late. Finally, for our extended Gauss-Newton approximation (right) only a single run diverges, illustrating the benefit of retaining certain second-order terms while preserving positive semi-definiteness.

Stability. The residual tendency of SMD to occasionally diverge can be suppressed further by slightly lowering the λ parameter. By curtailing the memory of iteration (11), however, this can compromise the rapid convergence of SMD, resulting in a stability/performance tradeoff (Fig. 3):

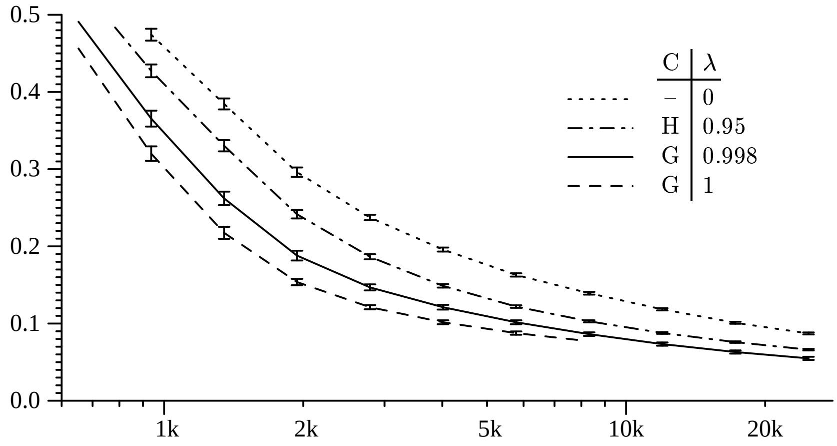

Fig. 3. Average loss over 25 runs of SMD for various combinations of curvature matrix C and forgetting factor λ. Memory ($\lambda \to 1$) is key to rapid convergence.

With the extended Gauss-Newton approximation, a small reduction of λ to 0.998 (solid line) is sufficient to prevent divergence, at a moderate cost in performance relative to $\lambda=1$ (dashed). When the Hessian is used, by contrast, λ must be set as low as 0.95 to maintain stability, and convergence is slowed much further (dash-dotted). Even so, this is still significantly faster than the degenerate case of $\lambda=0$ (dotted), which in effect implements IDD [12], the to our knowledge best competing online method for local learning rate adaptation.

From these experiments it appears that memory (*i.e.*, λ close to 1) is key to achieving the rapid convergence characteristic of SMD. We are now investigating other, more direct ways to keep iteration (11) under control, aiming to ensure the stability of SMD while maintaining its excellent performance at $\lambda = 1$.

Acknowledgment. We would like to thank Jenny Orr and Barak Pearlmutter for many helpful discussions, and the Swiss National Science Foundation for the financial support provided under grant number 2000–052678.97/1.

References

1. B. A. Pearlmutter, "Fast exact multiplication by the Hessian," *Neural Computation*, vol. 6, no. 1, pp. 147–160, 1994.
2. N. N. Schraudolph, "Local gain adaptation in stochastic gradient descent," in *Proc. 9th Int. Conf. Artificial Neural Networks*, pp. 569–574, IEE, London, 1999.
3. N. N. Schraudolph, "Online learning with adaptive local step sizes," in *Neural Nets – WIRN Vietri-99: Proc. 11th Italian Workshop on Neural Networks* (M. Marinaro and R. Tagliaferri, eds.), Perspectives in Neural Computing, (Vietri sul Mare, Salerno, Italy), pp. 151–156, Springer Verlag, Berlin, 1999.
4. N. N. Schraudolph, "Fast second-order gradient descent via O(n) curvature matrix-vector products," Tech. Rep. IDSIA-12-00, IDSIA, Galleria 2, CH-6928 Manno, Switzerland, 2000. Submitted to *Neural Computation*.
5. N. N. Schraudolph and X. Giannakopoulos, "Online independent component analysis with local learning rate adaptation," in *Adv. Neural Info. Proc. Systems* (S. A. Solla, T. K. Leen, and K.-R. Müller, eds.), vol. 12, pp. 789–795, The MIT Press, Cambridge, MA, 2000.
6. C. M. Bishop, *Neural Networks for Pattern Recognition*. Oxford: Clarendon, 1995.
7. S.-i. Amari, *Differential-Geometrical Methods in Statistics*, vol. 28 of *Lecture Notes in Statistics*. New York: Springer Verlag, 1985.
8. S.-i. Amari, "Natural gradient works efficiently in learning," *Neural Computation*, vol. 10, no. 2, pp. 251–276, 1998.
9. J. Kivinen and M. K. Warmuth, "Additive versus exponentiated gradient updates for linear prediction," in *Proc. 27th Annual ACM Symp. Theory of Computing*, (New York, NY), pp. 209–218, Association for Computing Machinery, 1995.
10. N. N. Schraudolph, "A fast, compact approximation of the exponential function," *Neural Computation*, vol. 11, no. 4, pp. 853–862, 1999.
11. S. Singhal and L. Wu, "Training multilayer perceptrons with the extended Kalman filter," in *Adv. Neural Info. Proc. Systems: Proc. 1988 Conf.* (D. S. Touretzky, ed.), pp. 133–140, Morgan Kaufmann, 1989.
12. M. E. Harmon and L. C. Baird III, "Multi-player residual advantage learning with general function approximation," Tech. Rep. WL-TR-1065, Wright Laboratory, WL/AACF, 2241 Avionics Circle, Wright-Patterson AFB, OH 45433-7308, 1996.